File No. S360-21
Order No. GC28-6514-7

OS

IBM

Systems Reference Library

IBM System/360 Operating System
Assembler Language

This publication contains specifications for the IBM System/360 Operating System Assembler Language (Levels E and F).

The assembler language is a symbolic programming language used to write programs for the IBM System/360. The language provides a convenient means for representing the machine instructions and related data necessary to program the IBM System/360. The IBM System/360 Operating System Assembler Program processes the language and provides auxiliary functions useful in the preparation and documentation of a program, and includes facilities for processing the assembler macro language.

Part I of this publication describes the assembler language.

Part II of this publication describes an extension of the assembler language -- the macro language -- used to define macro instructions.

PREFACE

This publication is a reference manual for the programmer using the assembler language and its features.

Part I of this publication presents information common to all parts of the language followed by specific information concerning the symbolic machine instruction codes and the assembler program functions provided for the programmer's use. Part II contains a description of the macro language and procedures for its use.

Appendixes A through J follow Part II. Appendixes A through F are associated with Parts I and II and present such items as a summary chart for constants, instruction listings, character set representations, and other aids to programming. Appendix G contains macro language summary charts, and Appendix H is a sample program. Appendix I is a features comparison chart of System/360 assemblers. Appendix J includes samples of macro definitions.

Knowledge of IBM System/360 machine operations, particularly storage addressing, data formats, and machine instruction formats and functions, is prerequisite to using this publication, as is experience with programming concepts and techniques or completion of basic courses of instruction in these areas. IBM System/360 machine operations are discussed in the publication IBM System/360: Principles of Operation, Form A22-6821. Information on program assembling, linkage editing, executing, interpreting listings, and assembler programming considerations is provided in IBM System/360 Operating System: Assembler (E) Programmer's Guide, Form C28-6595 and IBM System/360 Operating System: Assembler (F) Programmer's Guide, Form C26-3756.

The following publications are referred to in this publication:

IBM System/360 Operating System: Introduction, Form C28-6534

IBM System/360 Operating System: Linkage Editor and Loader, Form C28-6538

IBM System/360 Operating System: Supervi and Data Management Macro Instructions, Form C28-6647

IBM System/360 Operating System: Concepts and Facilities, Form C28-6535

IBM System/360 Operating System: Supervi and Data Management Services, Form C28-6646

IBM System/360 Operating System: Utilities, Form C28-6586

Seventh Edition (December, 1970)

This is a major revision of, and obsoletes, GC28-6514-6 and Technical Newsletter GN33-8072. The major changes are the addition of Assembler F support of the System/370 instructions and of weak external references (WXTRN).
Changes to the text and small changes to illustrations are indicated by a vertical line to the left of the change; changed or added illustrations are denoted by the symbol ● to the left of the caption.

This edition applies to release 20 of IBM System/360 Operating System and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually made to specifications herein; before using this publication in connection with the operations of IBM systems, consult the latest SRL Newsletter, Order No. GN20-0360, for the editions that are applicable and current.

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 readers' comments. If the form has been removed, comments may be addressed to IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden.

ILLUSTRATIONS

Figures

Tables

PART I -- THE ASSEMBLER LANGUAGE

SECTION 1: INTRODUCTION

SECTION 2: GENERAL INFORMATION

SECTION 3: ADDRESSING AND PROGRAM SECTIONING AND LINKING

SECTION 4: MACHINE INSTRUCTIONS

SECTION 5: ASSEMBLER INSTRUCTIONS

Computer programs may be expressed in machine language, i.e., language directly interpreted by the computer, or in a symbolic language, which is much more meaningful to the programmer. The symbolic language, however, must be translated into machine language before the computer can execute the program. This function is accomplished by a processing program.

Of the various 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 System/360. It enables the programmer to use all IBM System/360 machine functions, as if he were coding in System/360 machine language.

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

Compatibility

System/360 Operating System assemblers process source programs written in the Basic Programming Support/360 basic assembler language, the IBM 7090/7094 Support Package for IBM System/360 assembler language, the Basic Programming Support Assembler (8K Tape) language, the Basic Operating System Assembler (8K Disk) language, and the Disk and Tape Systems Assembler language, with the following exceptions:

1. The XFR assembler instruction is considered an invalid mnemonic operation code by Operating System/360 assemblers.

2. The assignment, size, and ordering of literal pools may differ among the assemblers.

Differences in the macro language for System/360 assemblers are described in Section 10 of this publication.

THE ASSEMBLER LANGUAGE

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

1. System/360 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 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 a macro instruction. A mnemonic symbol, supplied by the programmer, serves as the operation code of the instruction.

Machine Operation Codes

The assembler language provides mnemonic machine-instruction operation codes for all machine instructions in the IBM System/360 Universal Instruction Set and extended mnemonic operation codes for the conditional branch instruction.

Assembler Operation Codes

The assembler language also contains mnemonic assembler-instruction operation codes, used to specify auxiliary functions to be performed by the assembler. These are instructions to the assembler program itself and, with a few exceptions, result in the generation of no machine-language code by the assembler program.

Macro Instructions

The assembler language enables the programmer to define and use macro instructions.

Macro instructions are represented by an operation code which stands for a sequence of machine and/or assembler instructions. Macro instructions used in preparing an assembler language source program fall into two categories: system macro instructions, provided by IBM, which relate the object program to components of the operating system; and macro instructions created by the programmer specifically for use in the program at hand, or for incorporation in a library, available for future use.

Programmer-created macro instructions are used to simplify the writing of a program and to ensure that a standard sequence of instructions is used to accomplish a desired function. For instance, the logic of a program may require the same instruction sequence to be executed again and again. Rather than code this entire sequence each time it is needed, the programmer creates a macro instruction to represent the sequence and then, each time the sequence is needed, the programmer simply codes the macro instruction statement. During assembly, the sequence of instructions represented by the macro instruction is inserted in the object program.

Part II of this publication discusses the language and procedures for defining and using macro instructions.

THE ASSEMBLER PROGRAM

The assembler program, also referred to as the "assembler," processes the source statements written in the assembler language.

Basic Functions

Processing involves the translation of source statements into machine language, the assignment of storage locations to instructions and other elements of the program, and the performance of the auxiliary assembler functions designated by the programmer. The output of the assembler program is the object program, a machine-language translation of the source program. The assembler furnishes a printed listing of the source statements and object program statements and additional information useful to the programmer in analyzing his program, such as error indications. The object program is in the format required by the linkage editor component of Operating System/360. (See the linkage editor publication.)

The amount of main storage allocated to the assembler for use during processing determines the maximum number of certain language elements that may be present in the source program.

PROGRAMMER AIDS

The assembler provides auxiliary functions that assist the programmer in checking and documenting programs, in controlling address assignment, in segmenting a program, in data and symbol definition, in generating macro instructions, and in controlling the assembler itself. Mnemonic operation codes for these functions are provided in the language.

Variety in Data Representation: Decimal, binary, hexadecimal, or character representation of machine-language binary values may be employed by the programmer in writing source statements. The programmer selects the representation best suited to his purpose.

Base Register Address Calculation: As discussed in IBM System/360: Principles of Operation, the System/360 addressing scheme requires the designation of a base register (containing a base address value) and a displacement value in specifying a storage location. The assembler assumes the clerical burden of calculating 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 object programs produced by the assembler are in a format enabling relocation from the originally assigned storage area to any other suitable area.

Sectioning and Linking: The assembler language and program provide facilities for partitioning an assembly into one or more parts called control sections. Control sections may be added or deleted when loading the object program. Because control sections do not have to be loaded contiguously in storage, a sectioned program may be loaded and executed even though a continuous block of storage large enough to accommodate the entire program may not be available.

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 transfer of control between programs. A discussion of sectioning and linking is contained in Section 3 under the heading, "Program Sectioning and Linking."

Program Listings: A listing of the source program statements and the resulting object program statements may be produced by the assembler for each source program it assembles. The programmer can partly control the form and content of the listing.

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.

OPERATING SYSTEM RELATIONSHIPS

The assembler is a component of the IBM System/360 Operating System and, as such, functions under control of the operating system. The operating system provides the assembler with input/output, library, and other services needed in assembling a source program. In a like manner, the object program produced by the assembler will normally operate under control of the operating system and depend on it for input/output and other services. In writing the source program, the programmer must include statements requesting the desired functions from the operating system. These statements are discussed in the control program services publication. The introduction and the concepts and facilities publications provide further information on operating system relationships. Input/output considerations are discussed in the data management publication.

This section presents information about assembler language coding conventions and assembler source statement structure addressing.

ASSEMBLER LANGUAGE CODING CONVENTIONS

This subsection discusses the general coding conventions associated with use of the assembler language.

Coding Form

A source program is a sequence of source statements that are punched into cards. These statements may be written on the standard coding form, X28-6509 (shown in Figure 2-1), provided by IBM. One line of coding on the form is punched into one card. The standard card form, IBM 6509 (shown in Figure 2-2) can be used for punching source statements. The vertical columns on the form correspond to card columns. Space is provided on the form for program identification and instructions to keypunch operators.

The body of the form (Figure 2-1) is composed of two fields: the statement field, columns 1-71, and the identification-sequence field, columns 73-80. The identification-sequence field is not part of a statement and is discussed following the subsection "Statement Format."

The entries (i.e., coding) composing a statement occupy columns 1-71 of a line and, if needed, columns 16-71 of one or two successive continuation lines.

Continuation Lines

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

1. Write the statement up through column 71.
2. Enter a continuation character (not blank and not part of the coding) in column 72 of the line.
3. Continue the statement in column 16 of the next line, leaving columns 1 through 15 blank.
4. If the statement is not finished before column 71 of the second line, enter a continuation character in column 72, and continue in column 16 of the following line.
5. The statement has to be finished before column 71 of the third line, because the maximum number of continuation lines is two.
6. Macro instruction can be coded on as many lines as are needed.

These rules assume that normal source statement boundaries are used (see"Statement Boundaries" below).

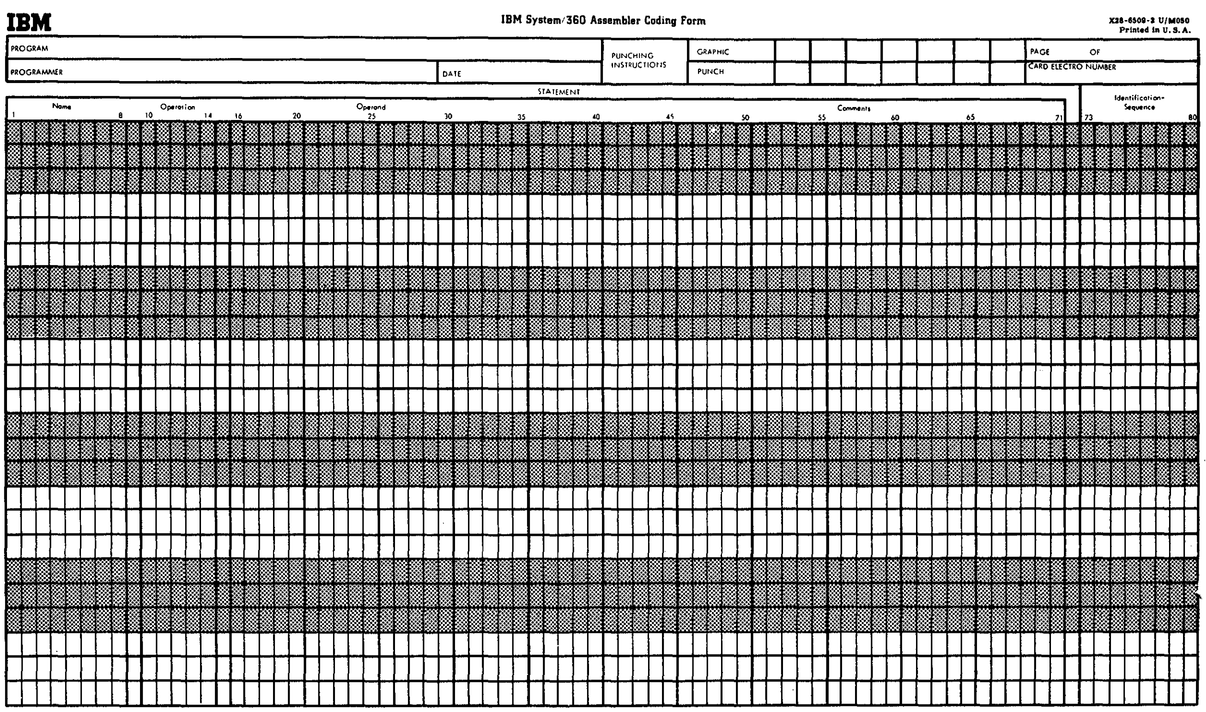

IBM
IBM System/360 Assembler Coding Form
X28-6509-3 U/M050
Printed in U.S.A.
PROGRAM
PROGRAMMER
DATE
PUNCHING INSTRUCTIONS
GRAPHIC
PUNCH
PAGE OF
CARD ELECTRO NUMBER
STATEMENT
Name
Operation
Operand
Comments
Identification-Sequence
1 8 10 14 16 20 25 30 35 40 45 50 55 60 65 71 73 80

Figure 2-1. Coding Form

Statement Boundaries

Source statements are normally contained in columns 1-71 of statement lines and columns 16-71 of any continuation lines. Therefore, columns 1, 71, and 16 are referred to as the "begin," "end," and "continue" columns, respectively. (This convention can be altered by use of the Input Format Control (ICTL) assembler instruction discussed later in this publication. The continuation character, if used, always immediately follows the "end" column.

Statement Format

Statements may consist of one to four entries in the statement field. They are, from left to right: a name entry, an operation entry, an operand entry, and a comments entry. These entries must be separated by one or more blanks, and must be written in the order stated.

The coding form (Figure 2-1) is ruled to provide an 8-character name field, a 5-character operation field, and a 56-character operand and/or comments field.

If desired, the programmer can disregard these boundaries and write the name, operation, operand, and comment entries in other positions, subject to the following rules:

1. The entries must not extend beyond statement boundaries within a line (either the conventional boundaries if no ICTL statement is given, or as designated by the programmer via the ICTL instruction).
2. The entries must be in proper sequence, as stated previously.
3. The entries must be separated by one or more blanks.
4. If used, a name entry must be written starting in the begin column.
5. The name and operation entries must be completed in the first line of the statement, including at least one blank following the operation entry.

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

Name Entry: The name entry is a symbol created by the programmer to identify a statement. A name entry is usually optional. The symbol must consist of eight characters or less, and be entered with the first character appearing in the begin column. The first character must be alphabetic. If the begin column is blank, the assembler program assumes no name has been entered. No blanks can appear in the symbol.

Operation Entry: The operation entry is the mnemonic operation code specifying the machine operation, assembler, or macro-instruction operation desired. An operation entry is mandatory and cannot appear in a continuation line. It must start at least one position to the right of the begin column. Valid mnemonic operation codes for machine and assembler operations are contained in Appendixes D and E of this publication. Valid operation codes consist of five characters or fewer for machine or assembler-instruction operation codes, and eight characters or fewer for macro-instruction operation codes. No blanks can appear within the operation entry.

Operand Entries: Operand entries identify and describe data to be acted upon by the instruction, by indicating such things as storage locations, masks, storage-area lengths, or types of data.

Depending on the needs of the instruction, one or more or no operands can be written. Operands are required for all machine instructions, but many assembler instructions require no operand.

Operands must be separated by commas, and no blanks can intervene between operands and the commas that separate them.

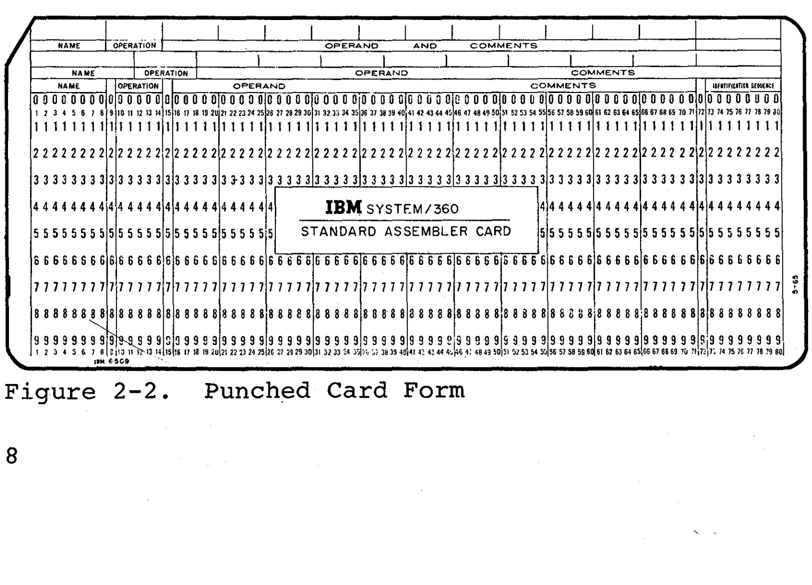


Figure 2-2. Punched Card Form

The first blank normally indicates the end of the operand field.

The operands cannot contain embedded blanks, except as follows:

> If character representation is used to specify a constant, a literal, or immediate data in an operand, the character string can contain blanks, e.g., C'A D'.

Comment Entries: Comments are descriptive items of information about the program that are shown on the program listing. All 256 valid characters (see Character Set in this section), including blanks can be used in writing a comment. The entry can follow the operand entry and must be separated from it by a blank; each line of comment entries cannot extend beyond the end column (column 71).

An entire statement field can be used for a comment by placing an asterisk in the begin column. Extensive comment entries can be written by using a series of lines with an asterisk in the begin column of each line or by using continuation lines. Comment entries cannot fall between a statement and its continuation line.

In statements where an optional operand entry is omitted but a comment entry is desired, the absence of the operand entry must be indicated by a comma preceded and followed by one or more blanks, as follows:

Name	Operation	Operand
	END	, COMMENT

For instructions that cannot contain an operand entry, this comma is not needed.

Note: Macro prototype statements and macro instructions without operands will not tolerate comments, even if a comma is coded as shown above.

For information on rules for the operand field of different assembler instructions, refer to the table in Appendix E.

Statement Example: The following example illustrates the use of name, operation, operand, and comment entries. A compare instruction has been named by the symbol COMP; the operation entry (CR) is the mnemonic operation code for a register-to-register compare operation, and the two operands (5,6) designate the two general registers whose contents are to be compared. The comment entry reminds the programmer that he is comparing "new sum" to "old" with this instruction.

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

Identification-Sequence Field

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

To aid in keeping source statements in order, the programmer can number the cards in this field. These characters are punched into their respective cards, and during assembly the programmer may request the assembler to verify this sequence by use of the Input Sequence Checking (ISEQ) assembler instruction. This instruction is discussed in Section 5, under Program Control Instructions.

Summary of Statement Format

The entries in a statement must always be separated by at least one blank and must be in the following order: name, operation, operand(s), comment(s).

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 is treated as a continuation line.

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

Character Set

Source statements are written using the following characters:

Letters A through Z, and $, #, @

Digits 0 through 9

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

These characters are represented by the card-punch combinations and internal bit configurations listed in Appendix A. In addition, any of the 256 punch combinations may be designated anywhere that characters may appear between paired apostrophes, in comments, and in macro instruction operands.

ASSEMBLER LANGUAGE STRUCTURE

The basic structure of the language can be stated as follows.

A source statement is composed of:

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

A name entry is:

- A symbol.

An operation entry is:

- A mnemonic 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.

Figure 2-3 depicts this structure. Terms shown in Figure 2-3 are classed as absolute or relocatable. Terms are absolute or relocatable, depending on the effect of program relocation upon them. Program relocation is the loading of the object program into storage locations other than those originally assigned by the assembler. A term is absolute if its value does not change upon relocation. A term is relocatable if its value changes upon relocation.

The following subsection "Terms and Expressions" discusses these items as outlined in Figure 2-3.

TERMS AND EXPRESSIONS

TERMS

Every term represents a value. This value may be assigned by the assembler (symbols, symbol length attribute, location counter reference) or 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 used to represent locations or arbitrary values. Symbols, through their use in name fields and in operands, provide the programmer with an efficient way to name and reference a program element. There are three types of symbols:

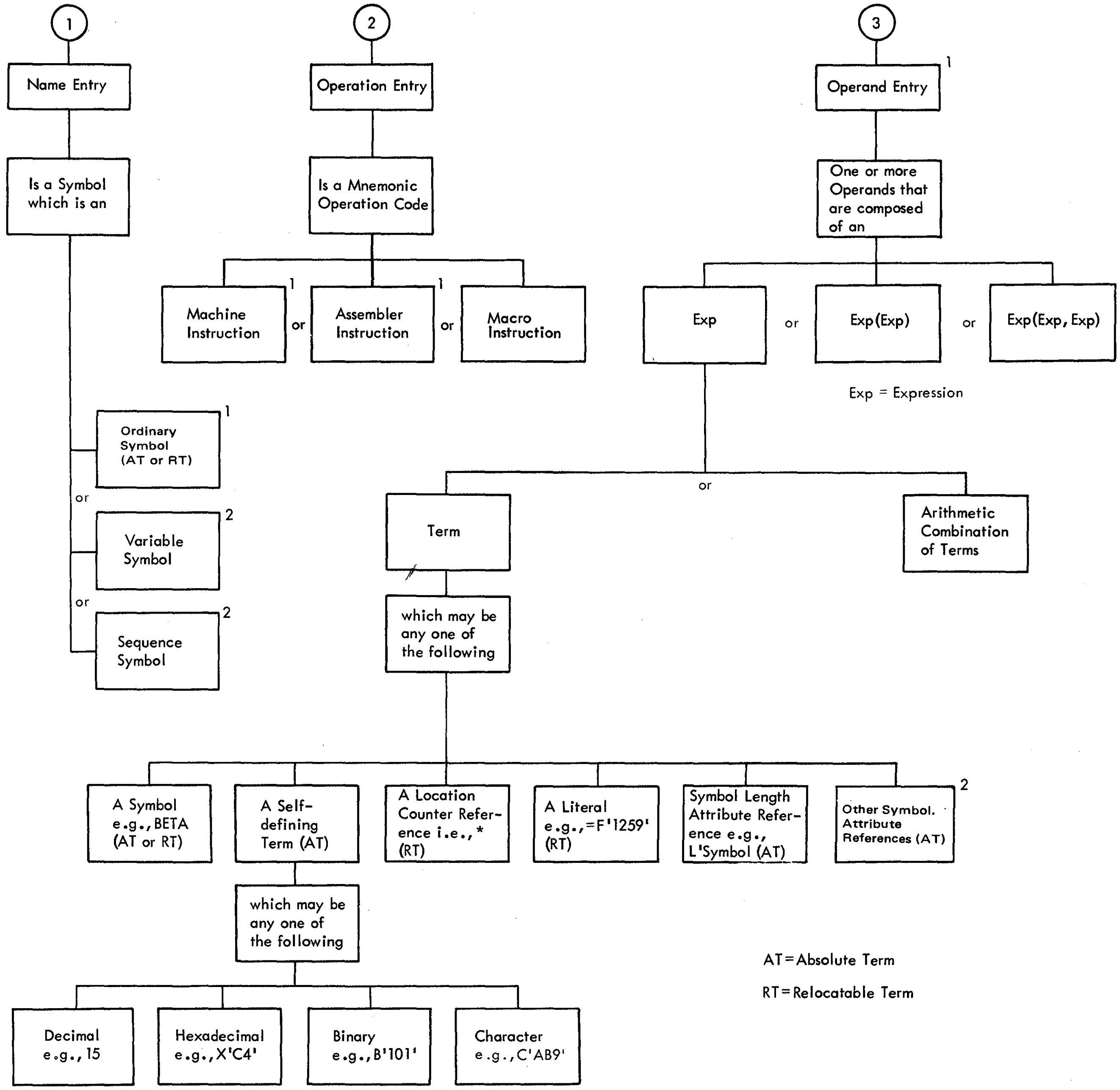
1. Ordinary symbols.
2. Variable symbols.
3. Sequence symbols.

Ordinary symbols, created by the programmer for use as a name entry and/or an operand, must conform to these rules:

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

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

3. No blanks are allowed in a symbol.



[1] May be generated by combination of variable symbols and assembler language characters. (Conditional assembly only)

[2] Conditional assembly only.

Figure 2-3. Assembler Language Structure -- Machine and Assembler Instructions

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

The following are valid symbols:

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

The following 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)

Variable symbols must begin with an ampersand (&) followed by one to seven letters and/or numbers, the first of which must be a letter. Variable symbols are used within the source program or macro definition to allow different values to be assigned to one symbol. A complete discussion of variable symbols appears in Section 6.

Sequence symbols consist of a period (.) followed by one to seven letters and/or numbers, the first of which must be a letter. Sequence symbols are used to indicate the position of statements within the source program or macro definition. Through their use the programmer can vary the sequence in which statements are processed by the assembler program. (See the complete discussion in Section 6.)

NOTE: Sequence symbols and variable symbols are used only for the macro language and conditional assembly. Programmers who do not use these features need not be concerned with these symbols.

DEFINING SYMBOLS: The assembler assigns a value to each symbol appearing as a name entry in a source statement. The values assigned to symbols naming 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 upon program relocation, the symbols naming them are considered relocatable terms.

A symbol used as a name entry in the Equate Symbol (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 (i.e., nonchanging) value, the symbol is considered a relocatable term or an absolute term, depending upon the value it is equated to.

The value of a symbol may not be negative and may not exceed 2^{24}-1.

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

Symbol definition also involves the assignment of a length attribute to the symbol. (The assembler maintains an internal table - the symbol table - in which the values and attributes of symbols are kept. When the assembler encounters a symbol in an operand, it refers to the table for the values associated with the symbol.) The length attribute of a symbol is the length, 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. Note that there are exceptions to this rule; for example, in the case where a symbol has been defined by an equate to location counter value (EQU *) or to a self-defining term, the length attribute of the symbol is 1. These and other exceptions are noted under the instructions involved. The length attribute is never affected by a duplication factor.

PREVIOUSLY DEFINED SYMBOLS: Some instructions require that a symbol appearing in the operand entry be previously defined. This simply means that the 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. That is, each symbol used as the name of a statement must be unique within that assembly. However, a symbol may be used in the name field more than once as a control section name (i.e., 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

A self-defining term is one whose value is inherent in the term. It is not assigned a value by the assembler. For example, the decimal self-defining term - 15 - represents a value of 15. The length attribute of a self-defining term is always 1.

There are four types of self-defining terms: decimal, hexadecimal, binary, and character. Use of these terms is spoken of as decimal, hexadecimal, binary, or character representation of the machine-language binary value or bit configuration they represent.

Self-defining terms are classed as absolute terms, since the values they represent do not change upon program relocation.

USING SELF-DEFINING TERMS: Self-defining terms are the means of specifying machine values or bit configurations without equating the values to symbol and using the symbols.

Self-defining terms may be used to specify such program elements as immediate data, masks, registers, addresses, and address increments. The type of term selected (decimal, hexadecimal, binary, or character) will depend on what is being specified.

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

Decimal Self-Defining Term: A decimal self-defining term is simply an unsigned decimal number written as a sequence of decimal digits. High-order zeros may be used (e.g., 007). Limitations on the value of the term depend on its use. For example, a decimal term that designates a general register should have a value between 0 and 15; one that represents an address should not exceed the size of storage. In any case, a decimal term may not consist of more than eight digits, or exceed 16,777,215 (2^{24}-1). A decimal self-defining term is assembled as its binary equivalent. Some examples of decimal self-defining terms are: 8, 147, 4092, and 00021.

Hexadecimal Self-defining Term: A hexadecimal self-defining term consists of one to six hexadecimal digits enclosed by apostrophes and preceded by the letter X: X'C49'.

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

The hexadecimal digits and their bit patterns are as follows:

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

A table for converting from hexadecimal representation to decimal representation is provided in Appendix B.

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, as follows: B'10001101'. This term would appear in storage as shown, occupying one byte. A binary term may have up to 24 bits represented.

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 three 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. In addition, any of the remainder of the 256 punch combinations may be designated in a character self-defining term. Examples of character self-defining terms are as follows:

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

Because of the use of apostrophes in the assembler language and ampersands in the macro language as syntactic characters, the following rule must be observed when using these characters in a character 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 eight-bit code equivalent (see Appendix A). The two apostrophes or ampersands that must be used to represent an apostrophe or ampersand within the character sequence are assembled as an apostrophe or ampersand.

Location Counter Reference

The 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 first 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. The location counter for each successively declared control section assigns locations in consecutively higher areas of storage. 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 control section will be assigned 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 they appear. The maximum value for the location counter is $2^{24}-1$.

The programmer may refer to the current value of the location counter at any place in a program by using an asterisk as a term in an operand. The asterisk represents the location of the first byte of currently available storage (i.e., after any required boundary adjustment). Using an asterisk as the operand 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 (i.e., 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 which requires the use of a predefined symbol, with the exception of the EQU and ORG assembler instructions.

Literals

A literal term is one of three basic ways to introduce data into a program. It is simply a constant preceded by an equal sign (=).

A literal represents data rather than a reference to data. The appearance of a literal in a statement directs the assembler program to assemble the data specified by the literal, store this data in a "literal pool," and place the address of the storage field containing the data in the operand field of the assembled statement.

Literals provide a means of entering constants (such as numbers for calculation, 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 is in contrast to using the DC 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.

A literal term cannot be combined with any other terms.

A literal cannot be used as the receiving field of an instruction that modifies storage.

A literal cannot be specified in a shift instruction or an I/O instruction (HIO, HDV, TIO, SIO, SIOF).

When a literal is contained in an instruction, it cannot specify an explicit base register or an explicit index register.

A literal cannot be specified in an address constant (see Section 5, DC--Define Constant).

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

NOTE: If a literal operand is a self-defining term (X,C,B, or decimal) and the equal sign (=) is omitted, the statement may assemble without error (See "Using Self-Defining Terms").

In general, literals can be used wherever a storage address is permitted as an operand. They cannot, however, be used in any assembler instruction that requires the use of a previously defined symbol. Literals are considered relocatable, because the address of the literal, rather than the literal itself, will be assembled in the statement that employs a literal. The assembler generates the literals, collects them, and places them in a specific area of storage, as explained in the subsection "The Literal Pool." A literal is not to be confused with the immediate data in an SI instruction. Immediate data is assembled into the instruction.

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

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in the operand of a DC assembler instruction. The major difference is that the literal must start with an equal sign (=), which indicates to the assembler that a literal follows. The reader is referred to the discussion of the DC assembler instruction operand format (Section 5) for the means of specifying a literal. 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'    --   a fixed-point number with
                 a length of four bytes.
=C'ABC'     --   a character literal.
```

The Literal Pool: The literals processed by the assembler are collected and placed in a special area called the literal pool, and the location of the literal, rather than the literal itself, is assembled in the statement employing 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.

The programmer may also specify that multiple literal pools be created. However, the sequence in which literals are ordered within the pool is controlled by the assembler. Further information on positioning the literal pool(s) is in 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, as in:

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 a storage field eight bytes in length and is assigned a length attribute of 8. B2 names a character constant two bytes in length and is assigned a length attribute of 2. The statement named 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.

The statement named 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.

Note: As previously stated, the length attribute of * is equal to the length of the instruction in which it appears, except in an EQU to *, in which case the length attribute is 1.

Terms in Parentheses

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

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

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

When the assembler program encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value which may be absolute or relocatable, depending on the combination of terms. This value 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. Five levels of parentheses are allowed; a level of parentheses is a left parenthesis and its corresponding right parenthesis. Parentheses which occur as part of an operand format do not count in this limit. An arithmetic combination of terms is evaluated as described in the next section "Expressions."

EXPRESSIONS

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

As shown in Figure 2-3, an expression is composed of a single term or an arithmetic combination of terms. The following are examples of valid expressions:

```
*                      BETA*10
AREA1+X'2D'            B'101'
*+32                   C'ABC'
N-25                   29
FIELD+332              L'FIELD
FIELD                  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 cannot start with an arithmetic operator, (+-/*). Therefore, the expression -A+BETA is invalid. However, the expression 0-A+BETA is valid.

2. An expression cannot contain two terms or two operators in succession.

3. An expression cannot consist of more than 16 terms.

4. An expression cannot have more than five levels of parentheses.

5. A multiterm expression cannot contain a literal.

Evaluation of Expressions

A single—term expression, e.g., 29, BETA, *, L'SYMBOL, takes on the value of the term involved.

A multiterm expression, e.g., BETA+10, ENTRY-EXIT, 25*10+A/B, is reduced to a single value, as follows:

1. Each term is evaluated.
2. Every expression is computed to 32 bits, and then truncated to the rightmost 24 bits.
3. Arithmetic operations are performed from left to right except that multiplication and division are done before addition and subtraction, e.g., A+B*C is evaluated as A+(B*C), not (A+B)*C. The computed result is the value of the expression.

4. Division always yields an integer result; any fractional portion of the result is dropped. E.g., 1/2*10 yields a zero result, whereas 10*1/2 yields 5.

5. Division by zero is permitted and yields a zero result.

Parenthesized multiterm subexpressions are processed before the rest of the terms in the expression, e.g., in the expression 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.

Negative values are carried in 2s complement form. Final values of expressions are the rightmost 24 bits of the results. Intermediate results have a range of -2^{31} through $2^{31}-1$. However, the value of an expression before truncation must be in the range -2^{24} through $2^{24}-1$ or the results will be meaningless. A negative result is considered to be a 3-byte positive value.

NOTE: In A-type address constants, the full 32-bit final expression result is truncated on the left to fit the specified or implied length of the constant.

Absolute and Relocatable Expressions

An expression is called absolute if its value is unaffected by program relocation.

An expression is called relocatable if its value depends upon program relocation.

The two types of expressions, absolute and relocatable, take on these characteristics from the term or terms composing them.

Absolute Expression: An absolute expression can be an absolute term or any arithmetic combination of absolute terms. An absolute term can be a non-relocatable symbol, any of the self-defining terms, or the length attribute reference. As indicated in Figure 2-3, all arithmetic operations are permitted between absolute terms.

An expression is absolute, even though it may contain relocatable terms (RT)--alone or in combination with absolute terms (AT)--under the following conditions.

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

2. The relocatable terms must be paired. Each pair of terms must have the same relocatability, i.e., they appear in the same control section in this assembly (see Program Sectioning and Linking, Section 3). Each pair must consist of terms with opposite signs. The paired terms do not have to be contiguous, e.g., RT+AT-RT.

3. No relocatable term can enter into a multiply or divide operation. Thus, RT-RT*10 is invalid. However, (RT-RT)*10 is valid.

The pairing of relocatable terms (with opposite signs and the same relocatability) cancels the effect of relocation since both symbols would be relocated by the same amount. 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 terms with the same relocatability. 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 evaluate as 35 (50-125+110=35).

An absolute expression reduces to a single absolute value.

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

A-Y+X
A
A*A
X-Y+A
*-Y (a reference to the location counter must be paired with another relocatable term from the same control section, i.e., with the same relocatability)

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

A relocatable expression can be a relocatable term. A relocatable expression can contain relocatable terms -- alone or in combination with absolute terms -- under the following conditions:

1. There must be an odd number of relocatable terms.
2. All the relocatable terms but one must be paired. Pairing is described in Absolute Expression.
3. The unpaired term must not be directly preceded by a minus sign.
4. No relocatable term can enter into a multiply or divide operation.

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

For example, in the expression 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, 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 IBM System/360 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 composed 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 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 in the machine.)

The programmer writes an explicit address by specifying the displacement and the base register number. In designating explicit addresses a base register may 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 it has been informed (1) what base registers are available to it and (2) what each contains. The programmer conveys this information to the assembler through the USING and DROP assembler instructions.

BASE REGISTER INSTRUCTIONS

The USING and DROP assembler 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.

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

Having the assembler determine 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 description of both instructions.

USING -- Use Base Address Register

The USING instruction indicates that one or more general registers are available for use as base registers. This instruction also states the base address values that the assembler may assume will be in the registers at object time. Note that a USING instruction does not load the registers specified. It is the programmer's responsibility to see that the specified base address values are placed into the registers. Suggested loading methods are described in the subsection "Programming with the USING Instruction." A reference to any name in a control section cannot occur in a machine instruction or an S-type address constant before the USING statement that makes that name addressable. The format of the USING instruction statement is:

Name	Operation	Operand
A sequence symbol or blank	USING	From 2-17 expressions of the form v,r1, r2,r3,...,r16

Operand v must be an absolute or relocatable expression. It may be a negative number whose absolute value does not exceed 2^{24}. No literals are permitted. Operand v specifies a value that the assembler can use as a base address. The other operands must be absolute expressions. The 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 the 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 then causes the assembler to assume that ALPHA+1000 is the value in register 9.

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

If the programmer has to refer to the first 4096 bytes of storage, he can use general register 0 as a base register subject to the following conditions:

1. The value of operand v must be either absolute or relocatable zero or simply relocatable, and
2. register 0 must be specified as operand r1.

The assembler assumes that register 0 contains zero. Therefore, regardless of the value of operand v, it calculates displacements as if operand v were absolute or relocatable zero. The assembler also assumes that subsequent registers specified in the same USING statement contain 4096, 8192, etc.

NOTE: If register 0 is used as a base register, the program is not relocatable, despite the fact that operand v may be relocatable. The program can be made relocatable by:

1. Replacing register 0 in the USING statement.
2. Loading the new register with a relocatable value.
3. Reassembling the program.

DROP -- Drop Base Register

The DROP instruction specifies a previously available register that may no longer be used as a base register. The format of the DROP instruction statement is as follows:

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

The expressions indicate general registers previously named in a USING statement that are now unavailable for base addressing. The following 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 being used 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.

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 specified in a USING instruction to have a relocatable value. A register with an absolute value is available only for absolute addresses. In either case, 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.

Name	Operation	Operand
BEGIN	BALR	2,0
	USING	*,2
FIRST	.	
	.	
	.	
LAST	.	
	END	BEGIN

In the preceding sequence, the BALR 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 BALR instruction. No other USING or load instructions are required if the location named LAST is within 4095 bytes of FIRST.

In Figure 3-1, the BALR and LM instructions 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 by altering the number of registers designated by the USING and LM instructions and the number of address constants specified in the DC instruction.

RELATIVE ADDRESSING

Relative addressing is the technique of addressing instructions and data areas by designating their location in relation to the location counter or to some symbolic location. This type of addressing is always in bytes, never in bits, words, or instructions. Thus, the expression *+4 specifies an address that is 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 of 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

Name	Operation	Operand
BEGIN	BALR	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 3-1. Multiple Base Register Assignment

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 under the heading "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 that can be relocated, independently of other coding, at load time without altering or impairing the operating logic of the program. It is normally identified by the CSECT instruction. However, if it is desired to specify a tentative starting location, the START instruction may be used to identify the first control section.

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 a load module.

The output from the assembler is called an object module. It contains data required for linkage editor processing. The external symbol dictionary, which is part of the object module, contains information the linkage editor needs in order to complete cross-referencing between control sections as it combines them into an object program. The linkage editor 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.

Whether the programmer writes an unsectioned program, a multisection program, or part of a multisection program, he still knows what eventually will be entered into storage because he has described storage symbolically. He may not know where each section appears in storage, but he does know what storage contains. There is no constant relationship between control sections. Thus, knowing the location of one control section does not make another control section addressable by relative addressing techniques.

The programmer must be aware that there is a limit to external symbol dictionary entries. The total number of control sections, dummy sections, unique symbols in EXTRN and WXTRN statements, V-type address constants, and external dummy sections must not exceed 255. Certain constants may cause a symbol to be counted twice: e.g., external symbols in V-type address constants (unless they are explicitly defined in an EXTRN or WXTRN statement), and external dummy sections implicitly defined by Q-type address constants and corresponding DSECT statements. EXTRN and WXTRN statements are described in this section; V-type and Q-type constants in Section 5 under "Operand Subfield 4: Constant."

Control Section Location Assignment

Control sections can be intermixed because the assembler provides a location counter for each control section. Locations are assigned to control sections as if the sections are placed in storage consecutively, in the same order as they first occur in the program. Each control section subsequent to the first begins at the next available double-word boundary.

FIRST CONTROL SECTION

The first control section of a program has the following special prcperties:

1. Its initial location counter value may be specified as an absolute value, if the START instruction is used.
2. It contains the literals of the program, unless their positioning has been altered by LTORG statements.

START -- Start Assembly

The START instruction may be used to give a name to the first (or only) control section of a program. It may also be used to specify an initial location counter value for the first control section of the program. The format of the START instruction statement is as follows:

Name	Operation	Operand
Any 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. This continues until a CSECT instruction identifying a different control section or a DSECT 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 whose value represents the address of the first byte of the control section. It has a length attribute of 1.

The assembler uses the self-defining term specified by the operand as the initial location counter value of the program. This value should be divisible by eight. For example, either of the following statements could be used to assign the name PROG2 to the first control section and to indicate an initial assembly location counter value of 2040. If the operand is omitted, the assembler sets the initial location counter value of the program at zero. The location counter is set at the next double-word boundary when the value of the START operand is not divisible by eight.

Name	Operation	Operand
PROG2	START	2040
PROG2	START	X'7F8'

Note:The START instruction must not be preceded by any code that will cause an unnamedccontrol section to be assembled. (See "Unnamed First Control Section" below.)

CSECT -- Identify Control Section

The CSECT instruction identifies the beginning or the continuation of a control section. The format of the CSECT instruction statement is as follows:

Name	Operation	Operand
Any symbol or blank	CSECT	Not used; should be 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 is encountered (i.e., another CSECT or a DSECT instruction).

The symbol in the name field is a valid relocatable symbol whose value represents the address of the first byte of the control section. It has a length attribute of 1.

Several CSECT statements with the same name may 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.

Unnamed First Control Section

All machine instructions and many assembler instructions have to belong to a control section. If such an instruction precedes the first CSECT instruction, the assembler will consider it to belong to an unnamed control section (also referred to as private code), which will be the first (or only) control section in the module.

The following instructions will not cause this to happen, since they do not have to belong to a control section:

Common Control Sections
Dummy Control Sections
Macro Definitions
Conditional Assembly Instructions
Comments
COPY (depends on the copied code)
DXD
EJECT
ENTRY
EXTRN
ICTL
ISEQ
OPSYN
PRINT
PUNCH
REPRO
SPACE
TITLE
WXTRN

No other assembler or machine instructions can precede a START instruction, since START, if used, must initiate the first control section in the program.

An involuntary unnamed control section at the beginning can cause trouble if literals are used. Then the programmer must be aware of the fact, that unless he codes an LTORG statement in each control section where he uses literals, literals will be assembled in the first control section, which will in this case be the involuntary section. If that control section does not establish addressability (through USING), an addressability error will be the result. Therefore statements like EQU should not be placed before the first CSECT or the START instruction.

Resumption of an unnamed control section at later points can be accomplished through unnamed CSECT statements. A program can contain only one unnamed control section.

Of course, it is possible to write a program that does not contain CSECT or START statements. It will then be assembled as one unnamed control section.

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 an area of storage without actually reserving the storage. (It is assumed that the storage is reserved either by some other part of this assembly or else 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. The format of the DSECT instruction statement is as follows:

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

The symbol in the name field is a valid relocatable symbol whose value represents the first byte of the section. It has 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. All assembler language instructions may occur within dummy sections.

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

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 zero at the beginning of the dummy section, and 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 specifying both a general register that the assembler can assign to the machine-instructions as a base register and a value from the dummy section that the assembler may assume the register contains.
2. Ensures that the same register is loaded with the actual address of the storage area.

The values assigned to symbols defined in a dummy section are relative to the initial statement of the section. Thus, all machine-instructions 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 following coding. Assume that two independent assemblies (assembly 1 and assembly 2) have been loaded and are to be executed as a single overall program. Assembly 1 is an input routine that places a record in a specified area of storage, places the address of the input area containing the record in general register 3, and branches to assembly 2. Assembly 2 processes the record. The coding shown in the example is from assembly 2.

The input area is described in assembly 2 by the DSECT control section named INAREA. Portions of the input area (i.e., 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, and that general register 3 is assumed to contain the address of INAREA.

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

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

The programmer must ensure that a section of code in his program is actually described by the dummy section which references it. Consider the following example, which illustrates how a dummy section should not be addressed:

Name	Operation	Operand
TEST	CSECT	
	.	
	.	
	CNOP	2,4
HALF	DS	CL2
FULL	DS	F
	.	
	.	
	END	
AREA	DSECT	
HALF	DS	CL2
FULL	DS	F

Note that in the dummy section AREA, two bytes are skipped between HALF and FULL in order to align FULL on a fullword boundary. In the control section TEST, however, the CNOP instruction causes two bytes to be skipped. Thus FULL is properly aligned without skipping any bytes between HALF and FULL.

When the programmer addresses the dummy section, the location of FULL (relative to the location of HALF) will not be the same as the location of FULL in the control section.

Note: To correct this example change the CNOP instruction to CNOP 0,4.

EXTERNAL DUMMY SECTIONS (ASSEMBLER F ONLY)

External dummy sections facilitate communication between programs by allowing the programmer to define work areas in several different programs and then at execution to combine them into one block of storage accessible to each program. Several different programs may be assembled together, each with one or more external dummy sections and after the linkage editor processes these programs, the programmer cal allocate storage for the dummy sections in one block. External dummy sections are defined through the use of the DXD instruction or a DSECT in combination with a Q-type DCIinstruction. In order to allocate the correct amount of storage when the program is executed, the programmer must use the CXD instruction, described below, within one of the programs.

DXD -- DEFINE EXTERNAL DUMMY SECTION

The DXD instruction (also referred to as a Pseudo Register) defines an external dummy section; when the assembler encounters a DXD instruction, it computes the amount of storage required and the alignment and passes this information to the linkage editor which will compute the total length of the external dummy sections. The format for the DXD instruction is:

Name	Operation	Operand
A symbol	DXD	Duplication factor, type, length, constant

The symbol in the name field is a symbol that usually appears as a Q-type constant in the operand field of a DC statement later in the program. It has a length attribute of 1. The operand form and alignment are the same as that described for the DS instruction. If more than one external dummy section with the same name is encountered by the linkage editor, it uses the largest section in computing total length; if two or more identically named external dummy sections have different boundary alignments, the linkage editor uses the most restrictive alignment in computing total length. An external dummy section is generated by a Q-type address constant which references a DSECT name.

CXD - CUMULATIVE LENGTH EXTERNAL DUMMY SECTION

The CXD instruction allocates a four-byte full-word aligned area in storage which will contain the sum of the lengths of all external dummy sections when the program is executed. This sum is supplied by the linkage editor. The instruction format is:

Name	Operation	Operand
Any 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 a 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	
	.	
	.	
	DC	Q(ALPHA)
	DC	Q(BETA)
	.	
	.	

ROUTINE B

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

ROUTINE C

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

Each of the three routines is requesting an amount of work area. Routine A wants 2 double words and 4 full words. Routine B wants 5 double words and 10 full words. Routine C wants 4 half words. At the time these routines are brought into storage the sum of the individual lengths will be placed in the location of the CXD instruction labeled OMEGA. Routine A can then allocate the amount of storage that is specified in the CXD location.

COM -- DEFINE BLANK COMMON CONTROL SECTION

The COM assembler instruction identifies and reserves a common area of storage that may be referred to by independent assemblies that have been linked and loaded for execution as one overall program.

Appearances of a COM statement after the initial one indicate the resumption of the blank common control section.

When several assemblies are loaded, each designating a common control section, the amount of storage reserved is equal to the longest common control section. The format is:

Name	Operation	Operand
A sequence symbol or blank	COM	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.

It is necessary to establish addressability relative to a named statement within COM since the COM statement itself cannot have a name. In the following example, addressability to the common area of storage is established relative to the named statement XYZ.

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

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

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

SYMBOLIC LINKAGES

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

In the module where the linkage symbol is defined (i.e., used as a name), it must also be identified to the assembler by means of the ENTRY assembler instruction unless the symbol is the name of a CSECT or START statement. It is identified as a symbol that names an entry point, which means that another module may use that symbol in order to effect a branch operation or a data reference. The assembler places this information in the control dictionary.

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

Another way to obtain symbolic linkages, is by using the V-type address constant. The subsection "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 or WXTRN statement. The V-type address constant may be used for external branch references (i.e., for effecting branches to other programs). It may not be used for external data references (i.e., for referring to data in other programs).

ENTRY -- IDENTIFY ENTRY-POINT SYMBOL

The ENTRY instruction identifies linkage symbols that are defined in one source module and referenced by other modules.

Name	Operation	Operand
A sequence symbol or blank	ENTRY	One or more relocatable symbols, separated by commas, that also appear as statement names

A source module may contain a maximum of 100 ENTRY symbols. ENTRY symbols which are not defined (not appearing as statement names), although invalid, will also count towards this maximum of 100 ENTRY symbols.

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 or in a blank common control section. The following example identifies the statements named SINE and COSINE as entry points to the program.

Name	Operation	Operand
	ENTRY	SINE,COSINE

Note: Labels of START and CSECT statements are automatically treated as entry points to a module. Thus they need not be identified by ENTRY statements.

EXTRN -- IDENTIFY EXTERNAL SYMBOL

The EXTRN instruction identifies linkage symbols used by one source module but identified in another module. Each external symbol must be identified. This includes symbols that refer to control section names. The format of the EXTRN statement is:

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

The symbols in the operand field may not appear as the name of statements in the module where the EXTRN statement is. The length attribute of an external symbol is 1.

The following example identifies three external symbols. They are used as operands in the module where they appear, but they are defined in some other module.

Name	Operation	Operand
	EXTRN	RATEBL,PAYCALC
	EXTRN	WITHCALC

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

Note 1: A V-type address constant does not have to be identified by an EXTRN statement.

Note 2: When external symbols are used in an expression they may not be paired. Each external symbol 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.

2. Load the constant into a general register and branch to the control section via the register.

For example, to link to the control section named SINE, the preceding coding might be used.

Name	Operation	Operand
MAINPROG	CSECT	
BEGIN	BALR	2,0
	USING	*,2
	.	
	.	
	L	3,VCON
	BALR	1,3
	.	
	.	
VCON	DC	V(SINE)
	END	BEGIN

An external symbol naming data may be referred to as follows:

1. Identify the external symbol with the EXTRN instruction, and create an address constant from the symbol.
2. Load the constant into a general register, and use the register for base addressing.

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

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

The total number of control sections, dummy sections, external symbols and external dummy sections must not exceed 255. Certain constants may cause a symbol to be counted twice: external symbols in V-type address constants (unless they are explicitly defined in an EXTRN or WXTRN statement), and external dummy sections implicitly defined by Q-type address constants and corresponding DSECT statements. (EXTRN and WXTRN statements are discussed in this section; V-type constants in Section 5 under the DC assembler instruction.)

WXTRN -- IDENTIFY WEAK EXTERNAL SYMBOL

The WXTRN statement has the same format as the EXTRN statement. It is used to identify weak external references. The only difference between a weak (WXTRN) and a strong (EXTRN or V-type constant) external reference is that the automatic library call mechanism of the linkage editor or loader is not effective for symbols that are identified in WXTRN statements.

The automatic library call mechanism searches the call library for any unresolved external references. If it finds any of these references, it includes the module where the reference occurs in the load module produced by the linkage editor or loader. Refer to IBM System/360 Operating System: Linkage Editor and Loader, GC28-6538 for a full description of the automatic library call mechanism.

The format of the WXTRN instruction is:

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

Note: If a V-type address constant is identified by a WXTRN instruction, the automatic library call mechanism is suppressed for it.

This section discusses the coding of the machine-instructions represented in the assembler language. The reader is reminded that the functions of each machine-instruction are discussed in the principles of operation manual (see Preface).

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. Appendix C illustrates machine format for the five classes of instructions. A mnemonic operation code is written in the operation field, and one or more operands are written in the operand field. Comments may be appended to a machine-instruction statement as previously explained in Section 1.

Any machine-instruction statement may be named by a symbol, which 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, as follows:

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 half-word boundaries. If any statement that causes information to be assembled requires alignment, the bytes skipped are filled with hexadecimal zeros. 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, as follows:

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

2. Double-shift, full-word 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, and 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 follows: 40(5). In the RX format, both an index register subfield and a base register subfield are written as follows: 40(3,5). In the SS format, both a length subfield and a base register subfield are written as follows: 40(21,5).

Appendix C shows two types of addressing formats for RX, RS, SI, and SS instructions. 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 the following symbolic operands:

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

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

In order to use implied addresses, the following 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 currently 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 zero, 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 4-1. 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, while an explicit address is used for the other.

Table 4-1. Address Specification Details

Type	Explicit Address	Implied Address
RX	D2(X2,B2)	S2(X2)
	D2(,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 separate operands. Parentheses must enclose a subfield or subfields, and a comma must 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, the following 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  32(,5),FIELD2 (implied length)
```

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 has been defined as consisting of one term or a series of arithmetically combined terms.) Refer to Appendix C for a detailed description of field requirements.

Note: Blanks may not appear in an operand unless provided by a character self-defining term or a character literal. Thus, 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 of the following:

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. The value of L'* is the length of the instruction in all non-literal machine instruction operands and in the CCW assembler instruction. In all other uses its value will be 1.

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 zero is desired, the length may be stated as zero or one.

To summarize, the length required in an SS instruction may be specified explicitly by the formats shown in the first column of Table 4-2 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, while an explicit length is used for the other.

Table 4-2. 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 (shown in Appendix D) are designed to be easily remembered codes that indicate 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. For example, A represents Add , and MV represents Move. The function may be further defined by a modifier. For example, the 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, full-word 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 are not considered here. For discussion of these considerations and for examples of coding sequences that illustrate them, the reader is referred to Section 3, "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(,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)

Whereas 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, whereas the BETA instructions and GAMMA4 specify implied addresses. GAMMA2 specifies a displacement of zero. 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 machine-instruction. 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 Figure 4-1, together with their machine-instruction equivalents. Unless otherwise noted, all extended mnemonics shown are for instructions in the RX 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, like the machine-instruction list, shows explicit address formats only. Each address can also be specified as an implied address.

Extended Code		Meaning	Machine-Instruction
B	D2(X2,B2)	Branch Unconditional	BC 15,D2(X2,B2)
BR	R2	Branch Unconditional (RR format)	BCR 15,R2
NOP	D2(X2,B2)	No Operation	BC 0,D2(X2,B2)
NOPR	R2	No Operation (RR format)	BCR 0,R2
	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 Instructions		
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)
BNP	D2(X2,B2)	Branch on Not Plus	BC 13,D2(X2,B2)
BNM	D2(X2,B2)	Branch on Not Minus	BC 11,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)
BNO	D2(X2,B2)	Branch if Not Ones	BC 14,D2(X2,B2)

Figure 4-1. Extended Mnemonic Codes

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(,6)
	BL	GO(3)
	BL	GO
	BR	4

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 last instruction is an unconditional branch to the address contained in register 4.

Just as machine instructions are used to request the computer to perform a sequence of operations during program execution time, so assembler instructions are requests to the assembler to perform certain operations during the assembly. Assembler-instruction statements, in contrast to machine-instruction statements, do not usually 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.

The following is a list of assembler instructions.

Symbol Definition Instruction
EQU - Equate Symbol

Operation Code Definition Instruction
OPSYN - Equate Operation Code (Assembler F only)

Data Definition Instructions
DC - Define Constant
DS - Define Storage
CCW - Define Channel Command Word

* Program Sectioning and Linking Instructions
START - Start Assembly
CSECT - Identify Control Section
CXD - Cumulative Length of External Dummy Section (Assembler F only)
DSECT - Identify Dummy Section
DXD - Define External Dummy Section (Assembler F only)
ENTRY - Identify Entry-Point Symbol
EXTRN - Identify External Symbol
WXTRN - Identify Weak External Symbol (Assembler F only)
COM - Identify Blank Common Control Section

* Base Register Instructions
USING - Use Base Address Register
DROP - Drop Base Address Register

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

* Discussed in Section 3.

SYMBOL DEFINITION INSTRUCTION

PUNCH - Punch a Card
REPRO - Reproduce Following Card

EQU -- EQUATE SYMBOL

The EQU instruction is used to define a symbol by assigning to it the length, value, and relocatability attributes of an expression in the operand field. The format of the EQU instruction statement is as follows:

Name	Operation	Operand
A variable symbol or ordinary symbol	EQU	An expression

The expression in the operand field can be absolute or relocatable. Any symbols appearing in the expression must be previously defined.

The symbol in the name field is given the same length, value, and relocatibility attributes as the expression in the operand field. The length attribute of the symbol is that of the leftmost (or only) term of the expression. In the case of EQU to * or to a self-defining term, the length attribute is 1. The value attribute of the symbol is the value of the expression.

The EQU instruction is used to equate symbols to register numbers, immediate data, or other arbitrary values. The following examples illustrate how this can be done:

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

To reduce programming time, the programmer can equate symbols to frequently used expressions and then use the symbols as operands in place 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. Note, however, that ALPHA, BETA, and GAMMA must all be previously defined. If the final result of the expression is negative, it is treated as if it were positive, i.e., the low-order 24 bits of the 2's complement is used.

The assembler assigns a length attribute of 1 in an EQU to * statement.

OPERATION CODE DEFINITION INSTRUCTION

OPSYN -- EQUATE OPERATION CODE (ASSEMBLER F ONLY)

The OPSYN instruction is used to define a machine mnemonic or extended mnemonic operation code as equivalent to another operation code. It is also used to prevent the assembler from recognizing an operation code. The OPSYN instruction has two formats:

Name	Operation	Operand
Any ordinary symbol, except an assembler operation code	OPSYN	A machine instruction mnemonic code, an extended mnemonic code, or an operation code defined by a previous OPSYN instruction

In this format, the OPSYN instruction assigns all the properties of the operation code in the operand field to the symbol in the name field. The symbol in the name field can be a previously defined machine or extended mnemonic operation code. In this case, the latest definition takes precedence.

Name	Operation	Operand
A machine or extended mnemonic operation code	OPSYN	Blank

In this format, the OPSYN instruction prevents the assembler from recognizing the operation code in the name field.

Only ICTL and OPSYN instructions may precede an OPSYN instruction.

Additional information on use of the OPSYN instruction is contained in IBM System/360 Operating System Assembler (F) Programmer's Guide (GC26-3756).

DATA DEFINITION INSTRUCTIONS

There are three data definition instruction statements: Define Constant (DC), Define Storage (DS), and Define Channel Command Word (CCW).

These statements are used to enter data constants into storage, to define and reserve areas of storage, and to specify the contents of channel command words. The statements can be named by symbols so that other program statements can refer to the generated fields. The DC instruction is presented first and discussed in more detail than the DS instruction because the DS instruction is written in the same format as the DC instruction and can specify some or all of the information that the DC instruction provides. Only the function and treatment of the statements vary.

DC -- DEFINE CONSTANT

The DC instruction is used to provide constant data in storage. It can specify one constant or a series of constants. A variety of constants can 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 as follows:

Name	Operation	Operand
Any symbol or blank	DC	One or more operands in the format described below, each separated by a comma

Each operand consists of four subfields: the first three describe the constant, and the fourth subfield provides the nominal value(s) for the constant(s). The first and third subfields can be omitted, but the second and fourth must be specified. Note that nominal value(s) for more than one constant can be specified in the fourth subfield for most types of constants. Each constant so specified must be of the same type; the descriptive subfields that precede the nominal value apply to all of them. No blanks can occur within any of the subfields (unless provided as characters in a character constant or a character self-defining term), nor can they occur between the subfields of an operand. Similarly, blanks cannot occur between operands and the commas that separate them when multiple operands are being specified.

The subfields of each DC operand are written in the following sequence:

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

Although the constants specified within one operand must have the same characteristics, each operand can specify a different type of constant. 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 (e.g., SYMBOL+2) can 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 in order 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 any incrementing is necessary) before the address value is established. Thus, the symbol naming the constant will not receive a value attribute that is the location of a skipped byte.

Any bytes skipped in aligning statements that do not cause information to be assembled are not zeroed. Bytes skipped to align a DC statement are zeroed; bytes skipped to align a DS statement are not zeroed.

Appendix F summarizes, in chart form, the information concerning constants that is presented in this section.

LITERAL DEFINITIONS: The reader is reminded that the discussion of literals as machine-instruction operands (in Section 2) referred him 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 being:

1. The literal is preceded by an equal sign.
2. Multiple operands may not be specified.
3. Unsigned decimal self-defining terms must be used to express the duplication factor and length modifier values.
4. The duplication factor may not be zero.
5. S-type address constants may not be specified.
6. Signed or unsigned decimal self-defining terms must be used to express scale and exponent modifiers.
7. Q-type address constants may not be specified in literals.

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

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 full word
H	Fixed-point	Signed, fixed-point binary format; normally a half word
E	Floating-point	Short floating-point format; normally a full word
D	Floating-point	Long floating-point format; normally a double word
L	Floating-point	Extended floating-point format; normally two double words (Assembler F only)
P	Decimal	Packed decimal format
Z	Decimal	Zoned decimal format
A	Address	Value of address; normally a full word
Y	Address	Value of address; normally a half word
S	Address	Base register and displacement value; a half word
V	Address	Space reserved for external symbol addresses; each address normally a full word
Q	Address	Space reserved for dummy section offset (Assembler F only)

Figure 5-1. Type Codes for Constants

Operand Subfield 1: Duplication Factor

The duplication factor may be omitted. If specified, it causes the constant(s) to be generated the number of times indicated by the factor. The factor may be specified either by an unsigned decimal self-defining term or by a positive absolute expression that is enclosed by parentheses. The duplication factor is applied after the constant is assembled. All symbols in the expression must be previously defined.

Note that a duplication factor of zero is permitted except in a literal and achieves the same result as it would in a DS instruction. A DC instruction with a zero duplication factor will not produce control dictionary entries. See "Forcing Alignment" under "DS -- Define Storage."

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 operand.

Operand Subfield 2: Type

The type subfield defines the type of constant being specified. From the type specification, the assembler determines how it is to interpret the constant and translate it into the appropriate machine format. The type is specified by a single-letter code as shown in Figure 5-1.

Further information about these constants is provided in the discussion of the constants themselves under "Operand Subfield 4: Constant."

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 self-defining term or a positive absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. The value of n represents the number of bytes of storage that are assembled for the constant. The maximum value permitted for the length modifiers supplied for the various types of constants is summarized in Appendix F. This table also 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. However, no boundary alignment will be provided when a length modifier is given.

Use of a length modifier may cause truncation. For example,

```
DC C'ABCDXYZ'
```

will generate a 7-byte constant, whereas

```
DC CL6'ABCDXYZ'
```

will generate a 6-byte constant and cause Z to be lost. Truncation of C, X, B, Z, A, Y, and P constants is not flagged as an error. However, F, H, E, D, and L constants will be flagged if significant bits are lost. Finally, each type of constant has an imposed or natural length modifier range limit. Appendix F shows which constants can be flagged for truncation of significant digits. It also shows the allowable length modifier range for each constant.

Bit-Length Specification: The length of a constant, in bits, is specified by L.n, where n is specified as stated above and represents the number of bits in storage into which the constant is to be assembled. The value of n may exceed eight 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 immediately follow in the same statement, the remainder of the last byte used is filled with zeros. This leaves the location counter set at the next byte boundary. Figure 5-2 shows a fixed-point constant with a specified bit-length of 13, as coded, and as it would appear in storage. Note that the constant has been padded on the left to bring it to its designated 13-bit length.

As coded:

Name	Operation	Operand
BLCON	DC	FL.13'579'

In storage:

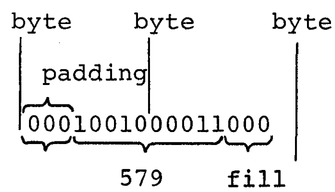


Figure 5-2. Bit-Length Specification (Single Constant)

The implied length of BLCON is two bytes. A reference to BLCON would cause the entire two bytes to be referenced.

When bit-length specification is used in association with multiple constants (see Operand Subfield 4: Constant following), each succeeding constant in the list is assembled starting at the next available bit. Figure 5-3 illustrates this.

As coded:

Name	Operation	Operand
BLMCON	DC	FL.10'161,21,57'

In storage:

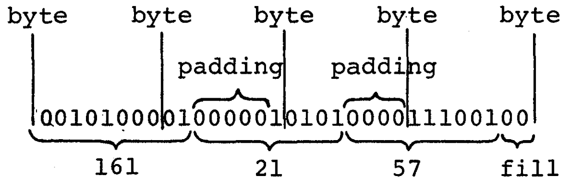


Figure 5-3. Bit-Length Specification (Multiple Constants)

The symbol used as a name entry in a DC assembler instruction takes on the length attribute of the first constant in the list; therefore the implied length of BLMCON in Figure 5-3 is two bytes.

If duplication is specified, filling occurs once at the end of the field occupied by the duplicated constant(s).

When bit-length specification is used in association with multiple operands, assembly of the constant(s) in each succeeding operand starts at the next available bit. Figure 5-4 illustrates this.

As coded:

Name	Operation	Operand
BLMOCON	DC	FL.7'9',CL.10'AB',XL.14'C4'

In storage:

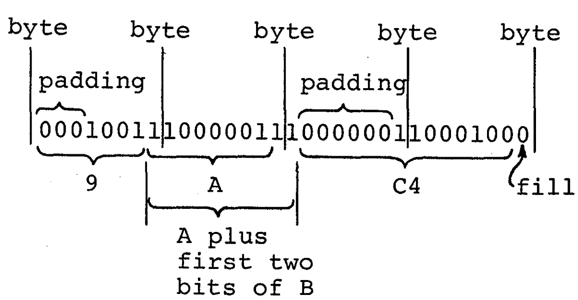


Figure 5-4. Bit-Length Specification (Multiple Operands)

In Figure 5-4, three different types of constants have been specified, one to an operand. Note that the character constant 'AB' which normally would occupy 16 bits is truncated on the right to fit the 10-bit field designated. Note that filling occurs only at the end of the field occupied by all the constants.

Scale Modifier: This modifier is written as Sn, where n is either a decimal value or an absolute expression enclosed by parentheses. All symbols 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 F.

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

Scale Modifier for Fixed-Point Constants: the scale modifier specifies the power of two 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 two 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 the following: (1) the number of binary positions to be occupied by the fractional portion of the binary number, or (2) the number of binary positions to be deleted from the integral portion of the binary number. A positive scale of x shifts 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 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 amount 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, i.e., contains hexadecimal zeros 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.

<u>EXPONENT MODIFIER:</u> 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 or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed.

An exponent modifier may be used with fixed-point (F, H) and floating-point (E,D,L) 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 "Operand Subfield 4: Constant." The exponent modifier affects each constant in the operand, whereas 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.

The range for the exponent modifier is -85 through +75. However, if there is an exponent in the constant itself (see "Floating-Point Constants -- E, D, and L" under "Operand Subfield 4: Constant") the sum of that exponent and the exponent modifier must be within the range -85 - +75. Thus, an exponent modifier of -40 together with an exponent of -47 would not be permitted.

One further limitation is that the value specified must be contained in the implied length of the constant. Refer to "Floating Point Arithmetic" in <u>IBM System/360 Principles of Operation</u> (GA22-6821).

<u>Operand Subfield 4: Constant</u>

This subfield supplies the constant (or constants) described by the subfields that precede it. A data constant (any type except A, Y, S, Q and V) is enclosed by apostrophes. An address constant (type A, Y, S, Q, or V) 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 (i.e., apostrophes or parentheses). Thus, the format for specifying the constant(s) is one of the following:

Single <u>Constant</u>	Multiple <u>Constants*</u>
'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, no boundary alignment is performed. If an operand specifies more than one constant, any necessary alignment applies to the first constant only. Thus, for an operand that provides five full-word constants, the first would be aligned on a full-word boundary, and the rest would automatically fall on full-word boundaries.

The total storage requirement of an operand is the product of the length times the number of constants in the operand times 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 that is 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 of the constant types and provides examples.

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

Special consideration must be given to representing apostrophes and ampersands as characters. Each single apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. 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 the following example, the length attribute of FIELD is 12:

Name	Operation	Operand
FIELD	DC	C'TOTAL IS 110'

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

Name	Operation	Operand
FIELD	DC	CL15'TOTAL IS 110'

In the next example, the length attribute of FIELD is 12, although 13 characters appear in the operand. The two ampersands count as only one byte.

Name	Operation	Operand
FIELD	DC	C'TOTAL IS &&10'

Note that in the next example, a length of four has been specified, but there are five characters in the constant.

Name	Operation	Operand
FIELD	DC	3CL4'ABCDE'

The generated constant would be:

ABCDABCDABCD

On the other hand, if the length had been specified as six instead of four, 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, which are 0-9 and A-F. Only one hexadecimal constant can 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 zero, while 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 zero is added to an odd number of digits). If a length modifier is given, the constant is handled as follows:

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 zeros.

An eight-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 1's:

Name	Operation	Operand
	DS	0F
TEST	DC	X'FF00FF00'

The DS instruction sets the location counter to a full word-boundary. (See DS--Define Symbol.)

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

Name	Operation	Operand
	IC	5,=X'FF'

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

Name	Operation	Operand
ALPHACON	DC	3XL2'A6F4E'

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

0A6F4E0A6F4E0A6F4E

Binary Constant -- B: A binary constant is written using 1's and 0's enclosed in apostrophes. Only one binary constant can be specified in an operand. Duplication and length can 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.

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

Name	Operation	Operand
BCON	DC	B'11011101'
BTRUNC	DC	BL1'100100011'
BPAD	DC	BL1'101'

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

00100011

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

00000101

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

1. The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, 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 Subfield 3: Modifiers.

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal self-defining term 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 power of 10 that it specifies before the constant is converted to its binary form. The exponent may exceed the permissible range for exponents, provided that the sum of the exponent and the exponent modifier does not exceed that range.

The number is converted to a binary number, and scaling is performed if specified. The binary number is then rounded and assembled into the proper field, according to the specified or implied length. The resulting number will not differ from the exact value by more than one in the last place. If the value of the number exceeds the length specified or implied, the sign is lost, the necessary leftmost bits are truncated to the length of the field, and the value is then assembled into the whole field. Any duplication factor that is present is applied after the constant is assembled. A negative number is carried in 2's complement form.

An implied length of four bytes is assumed for a full-word (F) and two bytes for a half-word (H), and the constant is aligned to the proper full-word or half-word if a length is not specified. However, any length up to and including eight bytes can 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}

A field of three full-words 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 full-word 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 two-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'-25.46'

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

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. Notice that 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, D, and L: A floating-point constant is written as a decimal number. As an option a decimal exponent may follow. 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 as follows:

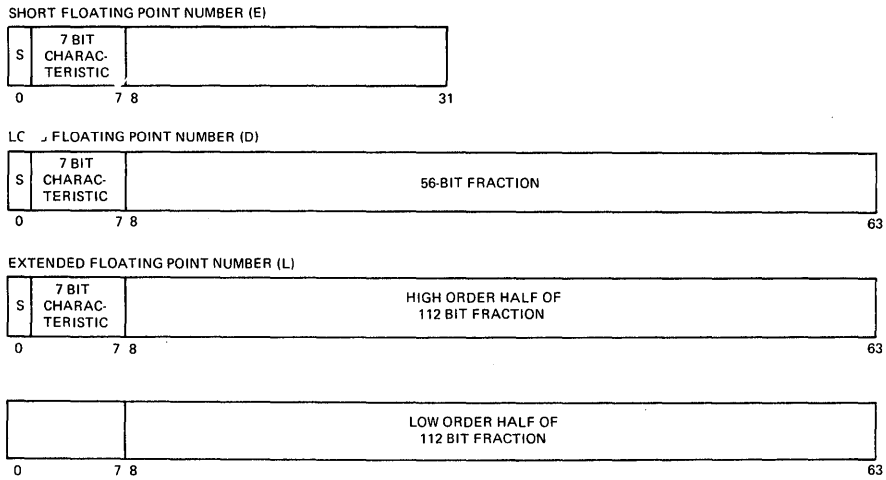


Figure 5-5. Floating-Point External Formats

1. The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, 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. If an unsigned exponent is specified, a plus sign is assumed. The range of the exponent is explained under "Exponent Modifier" above.

The external format for a floating-point number has two parts: the portion containing the exponent, which is sometimes called the characteristic, followed by 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. Figure 5-5 shows the external format of the three types of floating-point constants.

The type L constant resembles two contiguous type D constants. In the type L constant the sign of the second double word is the same as the sign of the first. The characteristic of the second double word is equal to the characteristic of the first minus 14, modulo 128. For information on use of the type L constant see the Assembler (F) Programmer's Guide.

For example, the constant 27.35E2 represents the number 27.35 times 10 to the 2nd. Represented as a fraction, it would be .2735 times 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, as explained under "Operand Subfield 3: Modifiers." Thus, the exponent is also altered before being translated into machine format.

In machine format a floating-point number also has two parts, the signed exponent and signed fraction. The quantity expressed by this number is the product of the fraction and the number 16 raised to the power of the exponent.

The exponent is translated into its binary equivalent in excess 64 binary notation and the fraction is converted to a binary number. Scaling is performed if specified; if not, the fraction is normalized (leading hexadecimal zeros are removed). Rounding of the fraction is then performed according to the specified or implied length, and the number is stored in the proper field. The resulting number will not differ from the exact value by more than one in the last place. 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. Negative fractions are carried in true representation, not in the 2s complement form.

An implied length of four bytes is assumed for a short (E) constant and eight bytes for a long (D) constant. An implied length of 16 bytes is assumed for an extended (L) constant. The constant is aligned at the proper word (E) or double word (D and L) boundary if a length is not specified. However, any length up to and including eight bytes (E and D) or 16 bytes (L) can be specified by a length modifier. In this case, no boundary alignment occurs.

Any of the following statements could be used to specify 46.415 as a positive, full-word, 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 double-word 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 may be omitted. Scaling and exponent modifiers may not be specified for decimal constants. The maximum length of a decimal constant is 16 bytes. No word boundary alignment is performed.

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. Furthermore, 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 (i.e., shift it for purposes of alignment).

If zoned decimal format is specified (Z), each decimal digit is translated into one byte. The translation is done according to the character set shown in Appendix A. The rightmost byte contains the sign as well as 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 zeros 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 zero bits for packed decimals). If a length modifier is given, the constant is handled as follows:

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 zero is placed in each added byte. For packed decimals, the bits of each added byte are set to zero.

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 follow.

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 (i.e., 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'

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

Name	Operation	Operand
	UNPK	OUTAREA,=PL2'+25'

Address Constants: An address constant is a storage address that is translated into a constant. Address constants can be used for initializing base registers to facilitate the addressing of storage. Furthermore, they provide a means of communicating between control sections of a multisection program. However, storage addressing and control section communication are also dependent on the use of the USING assembler instruction and the loading of registers. Coding examples that illustrate these considerations are provided in Section 3 under "Programming with the Using Instruction."

An address constant, unlike other types of constants, is enclosed in parentheses. If two or more address constants are specified in an operand, they are separated by commas, and the entire sequence is enclosed by parentheses. There are five types of address constants: A, Y, S, Q and V. A relocatable address constant may not be specified with bit lengths.

Complex Relocatable Expressions: A complex relocatable expression can only be used to specify an A-type or Y-type address constant. These expressions contain two or more unpaired relocatable terms and/or negative relocatable terms in addition to any absolute or paired relocatable terms that may be present. A complex relocatable expression might consist of external symbols and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the address constant.

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 to 32 bits as explained in Section 2 with one exception: the maximum value of the expression may be 2^{31}-1. The value is then truncated on the left, if necessary, to the specified or implied length of the field and assembled into the rightmost bits of the field. The implied length of an A-type constant is four bytes, and alignment is to a full-word boundary unless a length is specified, in which case no alignment will occur. The length that may be specified depends on the type of expression used for the constant; a length of .1-4 bytes may be used for an absolute expression, while a length of only 3 or 4 may be used for a relocatable or complex relocatable expression.

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

Name	Oper- ation	Operand
ACON	DC	A(108,LOP,END-STRT,*+4096)
	LM	4,7,=A(108,LOP,END-STRT,*+4096)

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

Y-Type Address Constant: A Y-type address constant has much in common with the A-type constant. It too is specified as an absolute, relocatable, or complex relocatable expression. The value of the expression is also calculated to 32 bits as explained in Section 2. However, the maximum value of the expression may be only 2^{15}-1. The value is then truncated, if necessary, to the specified or implied length of the field and assembled into the right-most bits of the field. The implied length of a Y-type constant is two bytes, and alignment is to a half-word boundary unless a length is specified, in which case no alignment will occur. The maximum length of a Y-type address constant is two bytes. If length specification is used, a length of two bytes may be designated for a relocatable or complex expression and .1 to 2 bytes for an absolute expression.

Warning: Specification of relocatable Y-type address constants should be avoided in programs destined to be executed on machines having more than 32,767 bytes of storage capacity. In any case Y-type relocatable address constants should not be used in programs to be executed under Operating System/360 control.

S-Type Address Constant: The S-type address constant is used to store an address in base-displacement form.

The constant may be specified in two ways:

1. As an absolute or relocatable expression, e.g., 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 converted by the assembler into the proper base register and displacement value. An S-type constant is assembled as a half word and aligned on a half-word 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 not be specified as literals.

Q-Type Address Constant (Assembler F only): 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 external dummy sections to access the desired section. The constant is specified as a relocatable symbol which has been previously defined in a DXD or DSECT statement. The implied length of a Q-type address constant is four bytes and boundary alignment is to a full word; a length of 1-4 bytes may be specified. No bit length specification is permitted in a Q-type constant. In the following example the constant VALUE has been previously defined in a DXD or DSECT statement. To access VALUE the value of A is added to the base address of the block of storage allocated for external dummy sections. Q-type address constants may not be specified in literals.

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 that is used for effecting branches to other programs. The constant may not be used for external data references within an overlay program. The constant is specified as one relocatable symbol, which need not be identified by an EXTRN statement. Whatever symbol is used is assumed to be an external symbol by virtue of the fact that it is supplied in a V-type address constant.

To suppress the automatic library call mechanism of the linkage editor for a constant identified in a V-type address constant, the programmer can identify it in a WXTRN statement (Assembler F only).

Note that specifying 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 full word. A length modifier may be used to specify a length of either three or four bytes, in which case no such boundary alignment occurs. In the following example, 12 bytes will be reserved, because there are three symbols. The value of each assembled constant will be zero until the program is loaded. It must be emphasized that a V-type address constant of length less than 4 can and will be processed by the Assembler but cannot be handled by the Linkage Editor.

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

DS -- DEFINE STORAGE

The DS instruction is used to reserve areas of storage and to assign 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
Any symbol or blank	DS	One or more operands, separated by commas, written in the format described in the following text

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

1. The specification of data (subfield 4) is optional in a DS operand, but it is mandatory in a DC operand. If the constant is specified, it must be valid.
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, all he needs to do is 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 for 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 zero.

Each field type (e.g., hexadecimal, character, floating-point) is associated with certain characteristics (these are summarized in Appendix F). The associated characteristics will determine which field-type code the programmer selects for the DS operand and what other information he adds, notably a length specification or a duplication factor. For example, the E floating-point field and the F fixed-point field both have an implied length of four bytes. The leftmost byte is aligned to a full-word boundary. Thus, either code could be specified if it were desired to reserve four bytes of storage aligned to a full-word boundary. To obtain a length of eight bytes, one could specify either the E or F field type with a length modifier of eight. 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, if used, would have to be accompanied by a length modifier, unless just one byte is to be reserved. Although no alignment occurs, the use of C and X field types permits greater latitude in length specifications, the maximum for either type being 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, either the length must be specified for the C, X, P, Z, or B field types, or else 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 as follows:

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 a SS machine-instruction operand.

Additional examples of DS statements are shown below:

Name	Operation	Operand
ONE	DS	CL80(one 80-byte field, length attribute of 80
TWO	DS	80C(80 one-byte fields, length attribute of one
THREE	DS	6F(six full words, length attribute of four)
FOUR	DS	D(one double word, length attribute of eight)
FIVE	DS	4H(four half-words, length attribute of two)

Note: A DS statement causes the storage area to be reserved but not set to zeros. 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 double-word, full-word, or half-word boundary by using the appropriate field type (e.g., D, F, or H) with a duplication factor of zero. 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 double-word boundary and then reserve storage space for a 128-byte field (whose leftmost byte would be on a double-word boundary).

Name	Operation	Operand
	DS	0D
AREA	DS	CL128

DEFINING FIELDS OF AN AREA: A DS instruction with a duplication factor of zero 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 the following format:

Positions 5-10	Payroll Number
Positions 11-30	Employee Name
Positions 31-36	Date
Positions 47-54	Gross Wages
Positions 55-62	Withholding Tax

The following example illustrates how DS instructions might be used to assign a name to the record area, then define the fields of the area and allocate the storage for them. Note that 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 six-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

CCW -- DEFINE CHANNEL COMMAND WORD

The CCW instruction provides a convenient way to define and generate an eight-byte channel command word aligned at a double-word boundary. CCW will cause any bytes skipped to be zeroed. The internal machine format of a channel command word is shown in Table 5-1.

Table 5-1. 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 zero
6	40-47	Set to zero
7-8	48-63	Count

The format of the CCW instruction statement is:

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

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

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

2. An expression specifying the data address. This value is treated as a 3-byte A-type constant. The value of this expression is in bytes 2-4.

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

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

The following is an example of a CCW statement:

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

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

32-35	36-39
0100	1000

If there is a symbol in the name field of the CCW instruction, it is assigned the address value of the leftmost byte of the channel command word. The length attribute of the symbol is 8.

LISTING CONTROL INSTRUCTIONS

The listing control instructions are used to identify an assembly listing and assembly output cards, to provide blank lines in an assembly listing, and to designate how much detail is to be included in an assembly listing. In no case are instructions or constants generated in the object program. Listing control statements with the exception of PRINT are not printed in the listing.

NOTE: TITLE, SPACE, and EJECT statements will not appear in the source listing unless the statement is continued onto another card. Then the first card of the statement is printed. However, any of these three types of statements, if generated as macro instruction expansion, will never be listed regardless of continuation.

TITLE -- IDENTIFY ASSEMBLY OUTPUT

The TITLE instruction enables the programmer to identify the assembly listing and assembly output cards. The format of the TITLE instruction statement is as follows:

Name	Operation	Operand
Special, sequence or variable symbol or blank	TITLE	A sequence of characters, enclosed in apostrophes

The name field may contain a special symbol of from one to four alphabetic or numeric characters in any combination. The contents of the name field are punched into columns 73-76 of all the output cards for the program except those produced by the PUNCH and REPRO assembler instructions. Only the first TITLE statement in a program may have a special symbol or a variable symbol in the name field. The name field of all subsequent TITLE statements must contain either a sequence symbol or a blank.

The operand field may contain up to 100 characters enclosed in apostrophes. Special consideration must be given to representing apostrophes and ampersands as characters. Each single apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage. The contents of the operand field are printed at the top of each page of the assembly listing.

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

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

Name	Operation	Operand
PGM1	TITLE	'FIRST HEADING'

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

If the following statement occurs later in the same program:

Name	Operation	Operand
	TITLE	'A NEW HEADING'

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

<u>Note:</u> The sequence number of the cards in the output deck is contained in columns 77-80.

EJECT -- START NEW PAGE

The EJECT instruction 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 of the EJECT instruction statement is as follows:

Name	Operation	Operand
A sequence symbol or blank	EJECT	Not used; should be blank

If the line before the EJECT statement appears at the bottom of a page, the EJECT statement has no effect. Two EJECT statements may be used in succession to obtain a blank page. A TITLE instruction followed immediately by an EJECT instruction will produce a page with nothing but the operand entry (if any) of the TITLE instruction. Text following the EJECT instruction will begin at the top of the next page.

SPACE -- SPACE LISTING

The SPACE instruction is used to insert one or more blank lines in the listing. The format of the SPACE instruction statement is as follows:

Name	Operation	Operand
A sequence symbol or blank	SPACE	A decimal value or blank

A decimal value is used to specify 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 is used to control printing of the assembly listing. The format of the PRINT instruction statement is:

Name	Operation	Operand
A sequence symbol or blank	PRINT	One to three operands

The one to three operands may include an operand from each of the following groups in any sequence:

1. ON - A listing is printed.

   OFF - No listing is printed.

2. GEN - All statements generated by macro-instructions are printed.

   NOGEN - Statements generated by macro-instructions are not printed with the exception of MNOTE which will print regardless of NOGEN. However, the macro-instruction itself will appear in the listing.

3. DATA - Constants are printed out in full in the listing.

   NODATA - Only the leftmost eight bytes are printed on the listing.

A program may contain any number of PRINT statements. A PRINT statement controls the printing of the assembly listing until another PRINT statement is encountered. Each option remains in effect until the corresponding opposite option is specified.

Until the first PRINT statement (if any) is encountered, the following is assumed:

Name	Operation	Operand
	PRINT	ON, NODATA, GEN

For example, if the statement:

Name	Operation	Operand
	DC	XL256'00'

appears in a program, 256 bytes of zeros are assembled. If the statement:

Name	Operation	Operand
	PRINT	DATA

is the last PRINT statement to appear before the DC statement, all 256 bytes of zeros are printed in the assembly listing. However, if:

Name	Operation	Operand
	PRINT	NODATA

is the last PRINT statement to appear before the DC statement, only eight bytes of zeros are printed in the assembly listing.

Whenever an operand is omitted, it is assumed to be unchanged and continues according to its last specification.

The hierarchy of print control statements is:

1. ON and OFF
2. GEN and NOGEN
3. DATA and NODATA

Thus with the following statement nothing would be printed.

Name	Operation	Operand
	PRINT	OFF, DATA, GEN

PROGRAM CONTROL INSTRUCTIONS

The program control instructions are used to specify the end of an assembly, to set the location counter to a value or word boundary, to insert previously written coding in the program, to specify the placement of literals in storage, to check the sequence of input cards, to indicate statement format, and to punch a card. Except for the CNOP and COPY instructions, none of these assembler instructions generate instructions or constants in the object program.

ICTL -- INPUT FORMAT CONTROL

The ICTL instruction allows the programmer to alter the normal format of his source program statements. The ICTL statement must precede all other statements in the source program and may be used only once. The format of the ICTL instruction statement is as follows:

Name	Operation	Operand
Blank	ICTL	1-3 decimal self-defining 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 within 1-40, inclusive. Operand e specifies the end column of the source statement. The end column, when specified, must be within 41-80, inclusive; when not specified, it is assumed to be 71. The end column must not be less than the begin column +5. 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 within 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 all statements are contained on a single card. The operand forms b,,c and b, are invalid.

If no ICTL statement is used in the source program, the assembler assumes that 1, 71, and 16 are the begin, end, and continue columns, respectively.

The 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

The ISEQ instruction is used to check the sequence of input cards. (A sequence error is considered serious, but the assembly is not terminated.) The format of the ISEQ instruction statement is as follows:

Name	Operation	Operand
Blank	ISEQ	Two decimal self-defining values of the form l,r; or blank

The operands l and r, respectively, 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 l. 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 eight-bit internal collating sequence. (See Appendix A.) Each card checked must be higher than the preceding card.

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

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

PUNCH -- PUNCH A CARD

The PUNCH assembler-instruction causes the data in the operand to be punched into a card. One PUNCH statement produces one punched card. As many PUNCH statements may be used as are necessary. The format is:

Name	Operation	Operand
A sequence symbol or blank	PUNCH	1 to 80 characters enclosed in apostrophes

Using character representation, the operand is written as a string of up to 80 characters enclosed in apostrophes. All characters, including blank, are valid. The position immediately to the right of the left apostrophe is regarded as column one of the card to be punched. Substitution is performed for variable symbols in the operand. 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.

PUNCH statements may occur anywhere within a program, except before macro definitions. They may occur within a macro definition but not between the end of a macro definition and the beginning of the next macro definition. If a PUNCH statement occurs before the first control section, the resultant card will precede all other cards in the object program card deck; otherwise the card will be punched in place. No sequence number or identification is punched in the card.

REPRO -- REPRODUCE FOLLOWING CARD

The REPRO assembler-instruction causes data on the following statement line to be punched into a card. The data is not processed; it is punched in a card, and no substitution is performed for variable symbols. No sequence number or identification is punched on the card. One REPRO instruction produces one punched card. The REPRO instruction may not appear before a macro definition. REPRO statements that occur before all statements composing the first or only control section will punch cards which precede all other cards of the object deck. The format is:

Name	Operation	Operand
A sequence symbol or blank	REPRO	Blank

The line to be reproduced may contain any combination of up to 80 valid characters. Characters may be entered starting in column 1 and continuing through column 80 of the line. Column 1 of the line corresponds to column 1 of the card to be punched.

ORG -- SET LOCATION COUNTER

The ORG instruction is used to alter the setting of the location counter for the current control section. The format of the ORG instruction statement is:

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

Any symbols in the expression must have been previously defined. The unpaired relocatable symbol must be defined in the same control section in which the ORG statement appears.

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

An ORG statement cannot be used to specify a location below the beginning of the control section in which it appears. The following is invalid if it appears less than 500 bytes from the beginning of the current control section since it will give the location counter a value larger than it can handle.

Name	Operation	Operand
	ORG	*-500

If it is desired to reset the location counter to the next available byte in the current control section, the following statement would 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 control section, an ORG statement with an omitted operand can then be used to terminate the effects of such statements and restore the location counter to its highest setting.

Note: Through use of the ORG statement two instructions may be given the same location counter values. In such a case the second instruction will not always eliminate the effects of the first instruction. Consider the following example:

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

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

LTORG -- BEGIN LITERAL POOL

The LTORG instruction causes all literals since the previous LTORG (or start of the program) to be assembled at appropriate boundaries starting at the first doubleword boundary following the LTORG statement. If no literals follow the LTORG statement, alignment of the next instruction (which is not a LTORG instruction) will occur. Bytes skipped are not zeroed. The format of the LTORG instruction statement 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.

The literal pool is organized into four segments within which the literals are stored in order of appearance, dependent on the divisibility properties of their object lengths (dup factor times total explicit or implied length). The first segment contains all literals whose object length is a multiple of eight. Those remaining literals with lengths divisible by four are stored in the second segment. The third segment holds the remaining even-length literals. Any literals left over have odd lengths and are stored in the fourth segment.

Since each literal pool begins at a double-word boundary, this guarantees that all segment one literals are double-word, segment two full-word, and segment three half-word aligned, with no space wasted except, possibly, at the pool origin.

Literals from the following statement are in the pool, in the segments indicated by the circled numbers, where 8 means multiple of eight, etc.,

```
MVC  A(12),=3F'1'   4
 AD  2,=D'1'          8
 SD  3,=2F'1,2'     8
 IC  2,=XL1'1'        1
 AD  2,=D'2'        8
```

Special Addressing Consideration

Any literals used after the last LTORG statement in a program are placed at the end of the first control section. If there are no LTORG statements in a program, 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 usage in subsequent control sections. If the programmer does not wish to reserve a register for this purpose, he may place a LTORG statement at the end of each control section thereby ensuring that all literals appearing in that section are 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.

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

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

XL3'0'
        Both are stored
HL3'0'

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

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

CNOP -- CONDITIONAL NO OPERATION

The CNOP instruction allows the programmer to align an instruction at a specific half-word boundary. If any bytes must be skipped in order 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).

The CNOP instruction ensures the alignment of the location counter setting to a half-word, word, or double-word 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 of which uses two bytes.

The format of the CNOP instruction statement is as follows:

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

Double Word							
Word				Word			
Half Word		Half Word		Half Word		Half Word	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0,4 0,8		2,4 2,8		0,4 4,8		2,4 6,8	

Figure 5-6. CNOP Alignment

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

b,w	Specifies
0,4	Beginning of a word
2,4	Middle of a word
0,8	Beginning of a double word
2,8	Second half word of a double word
4,8	Middle (third half word) of a double word
6,8	Fourth half word of a double word

Figure 5-6 shows the position in a double word that each of these pairs specifies. Note that both 0,4 and 2,4 specify two locations in a double word.

Assume that the location counter is currently aligned at a double-word boundary. Then the CNOP instruction in this sequence:

Name	Operation	Operand
	CNOP	0,8
	BALR	2,14

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

Name	Operation	Operand
	CNOP	6,8
	BALR	2,14

causes three branch-on-conditions (no-operations) to be generated, thus aligning the BALR instruction at the last half-word in a double word as follows:

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

After the BALR instruction is generated, the location counter is at a double-word 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. The format of the COPY instruction statement is as follows:

Name	Operation	Operand
Blank	COPY	One symbol

The operand is a symbol that identifies a partitioned data set member to be copied from either the system macro library or a user library concatenated to it. Inserting code in the library to be copied later is performed by the IEUPDAT or IEUPDTE routines, details of which are covered in the S/360 Operating System Utilities, Form C28-6586.

The assembler inserts the requested coding immediately after the COPY statement is encountered. The requested coding may not contain any COPY, END, ICTL, ISEQ, MACRO, or MEND statements.

If identical COPY statements are encountered, the coding they request is brought into the program each time. All statements included in the program via COPY are processed using the standard format regardless of any ICTL instructions in the program. (For a further discussion of COPY see Section 7.)

END -- END ASSEMBLY

The END instruction 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. A literal may not be used. If an external symbol is used in the expression, the value of the expression must be 0.

The format of the END instruction statement is as follows:

Name	Operation	Operand
Blank	END	A relocatable expression or blank

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

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

NOTE: Editing errors in system macro definitions (macro definitions included in a macro library) are discovered when the macro definitions are read from the macro library. This occurs after the END statement has been read. They will therefore be flagged after the END statement. If the programmer does not know which of his system macros caused an error, it is necessary to punch all system macro definitions used in the program, including inner macro definitions, and insert them in the source program as programmer macro definitions, since programmer macro definitions are flagged in-line. To aid in debugging it is advisable to test all macro definitions as programmer macro definitions before incorporating them in the library as system macro definitions.

PART II -- THE MACRO LANGUAGE

The Operating System/360 macro language is an extension of the Operating System/360 assembler language. It provides a convenient way to generate a desired sequence of assembler language statements many times in one or more programs. The macro-definition is written only once, and a single statement, a macro instruction statement, is written each time a programmer wants to generate the desired sequence of statements.

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

An additional facility, called conditional assembly, allows one to code 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 assembly. The conditional assembly facility may be used without using macro instruction statements.

THE MACRO INSTRUCTION STATEMENT

A macro instruction statement (hereafter called a macro instruction) is a source program statement. 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.

Macro instructions can be tested by placing them before the assembly cards of a test program.

Three types of macro instructions may be written. They are positional, keyword, and mixed-mode macro instructions. Positional macro instructions permit the programmer to write the operands of a macro instruction in a fixed order. Keyword macro instructions permit the programmer to write the operands of a macro instruction in a variable order. Mixed-mode macro instructions permit the programmer to use the features of both positional and keyword macro instructions in the same macro instruction.

THE MACRO DEFINITION

A macro definition is a set of statements that provides the assembler with: (1) the mnemonic operation code and the format of the macro instruction, and (2) 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, COPY statements, MEXIT, MNOTE, or conditional assembly instructions, and a macro definition trailer statement.

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

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

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

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

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

The MNOTE instruction 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, i.e., among the assembler language statements in the program.

THE MACRO LIBRARY

The same macro definition may be made available to more than one source program by placing the macro definition in the macro library. The macro library is a collection of macro definitions that can be used by all the assembler language programs in an installation. Once a macro definition has been placed in the macro library it may be used by writing its corresponding macro instruction in a source program. Macro definitions must be in the system macro library under the same name as the prototype. The procedure for placing macro definitions in the macro library is described in the Utilities publication.

SYSTEM AND PROGRAMMER MACRO DEFINITIONS

A macro definition included in a source deck is called a programmer macro definition. One residing in a macro library is called a system macro definition. There is no difference in function. If a programmer macro is included in a macro library it becomes a system macro definition, and if a system macro definition is punched and included in a source deck it becomes a programmer macro definition.

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

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

SYSTEM MACRO INSTRUCTIONS

The macro instructions that correspond to macro definitions prepared by IBM are called system macro instructions. System macro instructions are described in IBM System/360 Operating System Supervisor and Data Management Macro Instructions (GC28-6647).

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. Conditional assembly instructions, however, may be used to vary the number and format of the generated statements.

VARIABLE SYMBOLS

A variable symbol is a type of symbol that 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 statements are replaced with the values assigned to them. By changing the values assigned to variable symbols the programmer can vary 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. The SET symbols are further broken down into SETA symbols, SETB symbols, 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 by means 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 (i.e., 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.

ORGANIZATION OF THIS PART OF THE PUBLICATION

Sections 7 and 8 describe the basic rules for preparing macro definitions and for writing macro instructions.

Section 9 describes the rules for writing conditional assembly instructions.

Section 10 describes additional features of the macro language, including rules for defining global SET symbols, preparing keyword and mixed-mode macro definitions, and writing keyword and mixed-mode macro instructions.

Appendix G contains a reference summary of the entire macro language.

Examples of the features of the language appear throughout the remainder of the publication. These examples illustrate the use of particular features. However, they are not meant to show the full versatility of these features.

A macro definition consists of:

1. A macro definition header statement.

2. A macro instruction prototype statement.

3. Zero or more model statements, COPY statements, MEXIT, MNOTE, or conditional assembly instructions.

4. A macro definition trailer statement.

Except for MEXIT, MNOTE, and conditional assembly instructions, this section of the publication describes all of the 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 definitions appearing in a source program must appear before all PUNCH and REPRO statements and all statements which pertain to the first control section. Specifically, only the listing control instructions (EJECT, PRINT, SPACE, and TITLE), OPSYN, ICTL, and ISEQ instructions, and comments statements can occur before the macro definitions. All but the ICTL and OPSYN instruction can appear between macro definitions if there is more than one definition in the source program. Conditional assembly, substitution, and sequence symbols cannot be used in front of or between macro definitions.

A macro definition cannot appear within a macro definition and the maximum number of continuation cards for a macro definition statement is two.

MACRO -- MACRO DEFINITION HEADER

The macro definition header statement indicates the beginning of a macro definition. It must be the first statement in every macro definition. The format of this statement is:

Name	Operation	Operand
Blank	MACRO	Blank

MEND -- MACRO DEFINITION TRAILER

The macro definition trailer statement indicates the end of a macro definition. It can appear only once within a macro definition and must be the last statement in every macro definition. The format of this statement is:

Name	Operation	Operand
A sequence symbol or blank	MEND	Blank

MACRO INSTRUCTION PROTOTYPE

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

Name	Operation	Operand
A symbolic parameter or blank	A symbol	One or more symbolic parameters separated by commas, or blank

The symbolic parameters are used in the macro definition to represent the name field and operands of the corresponding macro instruction. A description of symbolic parameters appears under "Symbolic Parameters."

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 this 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 a machine or assembler instruction as listed in Appendix G.

The operand field may contain 0 to 200 symbolic parameters separated by commas. If there are no symbolic parameters, comments may not appear.

The following is an example of 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 assembler language statements. The normal format is described in Part I of this publication. The alternate format described here allows the programmer to write an operand on each line, and allows the interspersing of operands and comments in the statement.

In the alternate format, as in the normal format, 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 formats may be used in the same prototype statement.

The rules for using the alternate statement format 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 the next line starts after the continue column, the information entered on the next line is considered comments, and the operand field is considered terminated. Any subsequent continuation lines are considered comments.

Note: A prototype statement may be written on as many continuation lines as necessary. When using normal format, the operands of a prototype statement must begin on the first statement line or in the continue column of the second line.

The following examples illustrate: (1) the normal statement format, (2) the alternate statement format, and (3) the combination of both statement formats.

Name	Operation	Operand Comments	
NAME1	OP1	OPERAND1,OPERAND2,OPERAN D3 THIS IS THE NORMAL STATEMENT FORMAT	X X
NAME2	OP2	OPERAND1, THIS IS THE AL OPERAND2,OPERAND3, TERNA TE STATEMENT FORMAT	X X
NAME3	OP3	OPERAND1, THIS IS A COMB OPERAND2,OPERAND3,OPERAN D4,OPERAND5 INATION OF BOTH STATEMENT FORMATS	X X X

MODEL STATEMENTS

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

The fields in the model statement must correspond to the fields in the generated statement. It is not possible to generate blanks to separate statement fields.

Model statement fields must follow the rules for paired apostrophes, ampersands, and blanks as macro instruction operands (see "Macro Instruction Operands" in Section 8).

Though model statements must follow the normal continuation card conventions, statements generated from model statements may have more than two continuation lines. Substituted statements may not have blanks in any field except between paired apostrophes. They may not have leading blanks in the name or operand fields.

Name Field

The name field may be blank or it may contain an ordinary symbol, a variable symbol, or a sequence symbol. It may also contain an ordinary symbol concatenated with a variable symbol or a variable symbol concatenated with one or more other variable symbols.

Variable symbols may not appear in the name field of ACTR, COPY, END, ICTL, ISEQ, or OPSYN statements. The characters * and .* may not be substituted for a variable symbol.

Operation Field

The operation field may contain a machine instruction, an assembler instruction listed in Section 5 (except END, ICTL, ISEQ, OPSYN, or PRINT), a macro instruction, or variable symbol. It may also contain an ordinary symbol concatenated with a variable symbol or a variable symbol concatenated with one or more other variable symbols.

Variable symbols may not be used to generate

- Macro instructions,
- Assembler instructions appearing outside of Section 5, or
- END, ICTL, ISEQ, OPSYN, PRINT, or REPRO instructions.

Variable symbols may also be used outside of macro definitions to generate mnemonic operation codes with the preceding restrictions.

The use of COPY instructions is described under "COPY Statements".
Variable symbols in the line following a REPRO instruction, will not be replaced by their values.

Operand Field

The operand field may contain ordinary symbols or variable symbols. However, variable symbols may not be used in the operand field of COPY, END, ICTL, ISEQ, or OPSYN instructions.

Comments Field

The comments field may contain any combination of characters. No substitution is performed for variable symbols appearing in the comments field. Only generated statements will be printed in the listing.

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 through 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.

The following are valid symbolic parameters:

&READER	&LOOP2
&A23456	&N
&X4F2	&$4

The following are invalid symbolic parameters:

CARDAREA	(first character is not an ampersand)
&256B	(first character after ampersand is not a letter)
&AREA2456	(more than seven characters after the ampersand)
&BCD%34	(contains a special character other than initial ampersand)
&IN AREA	(contains a special character, i.e., 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, FIELDA, and FIELDB, respectively. If the preceding macro instruction was used in a source program, the following 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 different from those 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

The 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 (i.e., ST and L) in the model statements, the character that corresponds to &TY (i.e., D) is concatenated with the other characters to form the operation fields of the generated statements.

The 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, respectively. Since &P is followed by &FROM in the second model statement, the characters that correspond to them (i.e., 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 wishes 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 a special character other than a left parenthesis or another period that follows it.

If a symbolic parameter is immediately followed by a period, then 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. The characters FIELD of the macro instruction correspond to &P. Since &P is to be concatenated with a letter (i.e., 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 comments statements in a macro-definition which 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
```

NOTE: To get a truly representative sampling of the various language components used effectively in writing macro instructions the programmer may list all or selected macro instructions from the SYS1.GENLIB or the SYS1.MACLIB by using the IEBPTPCH system utility covered in the S/360 Operating System, Utilities manual, C28-6586.

COPY STATEMENTS

COPY statements may be used to copy model statements and MEXIT, MNOTE, and conditional assembly instructions into a macro-definition, just as they 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 operand is a symbol that identifies a partitioned data set member to be copied from either the system macro library or a user library concatenated to it. The symbol must not be the same as the operation mnemonic of a definition in the macro library. Any statement that may be used in a macro definition may be part of the copied coding, except MACRO, MEND, COPY, and prototype statements.

When considering statement positions within a program the code included by a COPY instruction statement should be considered rather than the COPY itself. For example if a COPY statement in a macro-definition brings in global and local definition statements, it may appear immediately after the prototype statement. However, since global definition statements must come before local definition statements, if global and local definition statements are also specified explicitly in the macro definition which contains the COPY, the COPY must occur between the explicit global definition statements and the explicit local definition statements.

The format of a macro instruction is:

Name	Operation	Operand
Any symbol or blank	Mnemonic operation code	0-200 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 operation code must be the same as the mnemonic operation code of a macro definition in the source program or in the macro library.

The macro definition with the same mnemonic operation code 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 operation code, the macro definition in the source program is used.

The placement and order of the operands in the macro instruction is 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 that 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. A quoted string is any sequence of characters that begins and ends with an apostrophe and contains an even number 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.

A quoted string ends with the first even-numbered apostrophe that is not immediately followed by another apostrophe.

The first and last apostrophes of a quoted string are called paired apostrophes. The following example contains two quoted strings. The first and fourth and the fifth and sixth apostrophes are each paired apostrophes.

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

An apostrophe not within a quoted string, immediately followed by a letter, and immediately preceded by the letter L (when L is preceded by any special character other than an ampersand), is not considered in determining paired apostrophes. For instance, in the following example, the apostrophe is not considered.

```
L'SYMBOL
'AL'SYMBOL' is an invalid operand.
```

Paired Parentheses: There must be an equal number of left and right parentheses. The nth left parenthesis must appear to the left of the nth right parenthesis.

Paired parentheses are a left parenthesis and a following right parenthesis without any other parentheses intervening. 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 each paired parentheses.

```
(A(B)C)D(E)
```

A parenthesis that appears between paired apostrophes is not considered in determining paired parentheses. For instance, in the following example the middle parenthesis is not considered.

```
(')')
```

Equal Signs: An equal sign can only occur as the first character in an operand or between paired apostrophes or paired parentheses. The following examples illustrate these rules.

```
=F'32'
'C=D'
E(F=G)
```

Ampersands: Except as noted under "Inner Macro Instructions," each sequence of consecutive ampersands must be an even number of ampersands. The following example illustrates this rule.

&&123&&&&

Commas: A comma indicates the end of an operand, unless it is placed between paired apostrophes or paired parentheses. The following example illustrates this rule.

(A,B)C','

Blanks: Except as noted under "Statement Format," a blank indicates the end of the operand field, unless it is placed between paired apostrophes. The following example illustrates this rule.

'A B C'

The following 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 ,'

The following 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 paired apostrophes)
'ONE' IS '1'	(blank not placed between paired apostrophes)

STATEMENT FORMAT

Macro instructions may be written using the same alternate format that can be used to write prototype statements. If this format is used, a blank does not always indicate the end of the operand field. The alternate format is described in Section 7, under the subsection "Macro Instruction Prototype."

OMITTED OPERANDS

If an operand that appears in the prototype statement is omitted from the macro instruction, then the comma that would have separated it from the next operand must be present. If the last operand(s) is omitted from a macro instruction, then the comma(s) separating the last operand(s) from the next previous operand may be omitted.

The following example shows a macro instruction preceded by its corresponding prototype statement. The macro instruction operands that correspond to the third and sixth operands of the prototype statement are omitted in this example.

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, i.e., 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 was omitted from the macro instruction, the second statement below would be generated from the model statement.

Name	Operation	Operand
	MVC	THERE&C.25,THIS
	MVC	THERE25,THIS

OPERAND SUBLISTS

A sublist may occur as the operand of a macro instruction.

Sublists provide the programmer with a convenient way to refer to a collection of macro instruction operands as a single operand, or 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 operand of the sublist may be written on a separate line; the macro instruction may be written on as many lines as necessary.

If &P1 is a symbolic parameter in a prototype statement, and the corresponding operand of a macro-instruction is a sub-list, then &P1(n) may be used in a model statement to refer to the nth operand of the sublist, where n may have a value greater than or equal to 1. N may be specified as a decimal integer or any arithmetic expression allowed in a SETA instruction. (The SETA instruction is described in Section 9.) If the nth operand is omitted, then &P1(n) would refer to a null character value.

If the sublist notation is used but the operand is not a sublist, then &P1(1) refers to the operand and &P1(2), &P1(3),... refer to a null character value. If an operand has the form ( ), it is treated as a character string and not as a sublist.

For example, consider the following macro-definition, macro-instruction, and generated statements.

	Name	Operation	Operand
Header		MACRO	
Prototype		ADD	&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		ADD	(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 of the model statements. For example, &NUM(1) refers to the first operand in the sublist corresponding to symbolic parameter &NUM. The first operand of the sublist 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 of the symbolic parameter, e.g., &NUM(1). 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 above example.

	Name	Operation	Operand
Prototype		ADD	&NUM,&REG,&AREA
Model		L	&REG,&NUM.(1)
Macro		ADD	(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 does not appear 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. Macro instructions used as model statements are called inner macro instructions.

A macro instruction that is not used as a model statement is referred to as an outer macro instruction.

The rule for inner macro instruction parameters is the same as that for outer macro instructions. Any symbolic parameters used in an inner macro instruction are replaced by the corresponding characters of the outer macro instruction. An operand of an outer macro instruction sublist cannot be passed as a sublist to an inner macro instruction.

The macro definition corresponding to an inner macro instruction is used to generate the statements that replace the inner macro instruction.

The ADD 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. The characters (X,Y,Z) and J of the macro instruction correspond to &S and &T, respectively. Therefore, these characters replace the symbolic parameters in the operand field of the inner macro instruction.

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 generated statements have been generated for 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		ADD	&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

Further relevant limitations and differences between inner and outer macro instructions will be covered under the pertinent sections on sequence symbols, attributes, etc.

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 called a first level macro instruction. Each of the inner macro instructions is called 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 macro instructions are called third level macro instructions, etc.

The number of levels of macro instructions that may be used depends upon the complexity of the macro definition and the amount of storage available.

The conditional assembly instructions allow the programmer to: (1) define and assign values to SET symbols that can be used to vary parts of generated statements, and (2) 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.

There are 13 conditional assembly instructions, 10 of which are described in this section. The other three conditional assembly instructions -- GBLA, GBLB, and GBLC -- are described in Section 10. The instructions described in this section are:

LCLA	SETA	AIF	ANOP
LCLB	SETB	AGO	
LCLC	SETC	ACTR	

The primary use of the conditional assembly instructions is in macro-definitions. However, all of them may be used in an assembler language source program.

Where the use of an instruction outside macro-definitions differs from its use within macro-definitions, the difference is described in the subsequent text.

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 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 symbols or macro-instruction operands to determine which statements are to be processed. The ACTR instruction may be used to vary the maximum number of AIF and AGO branches.

Examples illustrating the use of conditional assembly instruction are included throughout this section. 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. The symbolic parameters discussed in Section 7 are another type of variable symbol. SET symbols differ from symbolic parameters in three ways: (1) where they can be used in an assembler language source program, (2) how they are assigned values, and (3) whether or not the values assigned to them can be changed.

Symbolic parameters can only be used in macro-definitions, whereas SET symbols can be used inside and outside macro-definitions.

Symbolic parameters are assigned values when the programmer writes a macro-instruction, whereas SET symbols are assigned values when the programmer 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. SET symbols may be assigned new values by means of 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

The SETA, SETB, and SETC instructions may be used to change the values assigned to SETA, SETB, and SETC symbols, respectively. When a SET symbol appears in the name, operation, or operand field of a model statement, the current value of the SET symbol (i.e., the last value assigned to it) 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 following illustrates this rule.

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

If the statement above is a prototype statement, then &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. If such is the 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 (either SET symbol or symbolic parameter) in another definition. Similarly, if &A is a variable symbol (SET symbol or symbolic parameter) 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 concatenating symbolic parameters with other characters are in Section 7 under the subsection "Symbolic Parameters."

Variable symbols in macro-instructions are replaced by the values assigned to them, immediately prior to the start of 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 and to symbols in the program. These attributes may be referred to only in conditional assembly instructions or expressions.

There are six kinds of attributes. They are: type, length, scaling, integer, count, and number. Each kind of attribute is discussed in the paragraphs that follow.

If an outer macro-instruction operand is a symbol before substitution, then the attributes of the operand are the same as the corresponding attributes of the symbol. The symbol must appear in the name field of an assembler language statement or in the operand field of an EXTRN statement in the program. The statement must be outside macro-definitions and must not contain any variable symbols.

If an inner macro-instruction operand is a symbolic parameter, then the attributes of the operand are the same as the attributes of the corresponding outer macro instruction operand. A symbol appearing as an inner macro instruction operand is not assigned the same attributes as the same symbol appearing as an outer macro instruction operand.

If a macro-instruction operand is a sublist, the programmer may refer to the attributes of either the sublist or each operand in the sublist. The type, length, scaling, and integer attributes of a sublist are the same as the corresponding attributes of the first operand in the sublist.

All the attributes of macro-instruction operands may be referred to in conditional assembly instructions within macro-definitions. However, only the type, length, scaling, and integer attributes of symbols may be referred to in conditional assembly instructions outside macro-definitions. Symbols appearing in the name field of generated statements are not assigned attributes.

Each attribute has a notation associated with it. The notations are:

Attribute	Notation
Type	T'
Length	L'
Scaling	S'
Integer	I'
Count	K'
Number	N'

The programmer may refer to an attribute in the following ways:

1. In a statement that is outside macro definitions, he may write the notation for the attribute immediately followed by a symbol. (e.g., T'NAME refers to the type 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 characters in the macro-instruction that correspond 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.)

Type Attribute (T')

The type attribute of a macro instruction operand, or a symbol is a letter.

The following letters are used for symbols that name DC and DS statements and for outer macro instruction operands that are symbols that name DC or DS statements.

A	A-type address constant, implied length, aligned, (also in CXD statement)
B	Binary constant.
C	Character constant.
D	Long floating-point constant, implied length, aligned.
E	Short floating-point constant, implied length, aligned.
F	Full-word fixed-point constant, implied length, aligned.
G	Fixed-point constant, explicit length.
H	Half-word fixed-point constant, implied length, aligned.
K	Floating-point constant, explicit length.
L	Extended floating-point constant, implied length, aligned
P	Packed decimal constant.
Q	Q-type address constant, implied length, aligned.
R	A-, S-, Q-, 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.

The following letters are used for symbols (and outer macro instruction operands that are symbols) that name statements other than DC or DS statements, or that appear in the operand field of an EXTRN statement.

I	Machine instruction
J	Control section name
M	Macro instruction
T	EXTRN symbol
W	CCW instruction
$	WXTRN symbol

The following letters are used for inner and outer macro instruction operands only.

N	Self-defining term
O	Omitted operand

The following letter is used for inner and outer macro instruction operands that cannot be assigned any of the above letters. This includes inner macro instruction operands that are symbols.

This letter is also assigned to symbols that name EQU and LTORG statements, to any symbols occurring more than once in the name field of source statements, and to all symbols naming statements with expressions as modifiers.

U   Undefined

The attributes of A, B, C and D are undefined in the following example:

Name	Operation	Operand
A	DC	3FL(AA-BB)'75'
B	DC	(AA-BB)F'15'
C	DC	&X'1'
D	DC	FL(3-2)'1'

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

The length, scaling, and integer attributes of macro instruction operands, and symbols are numeric values.

The length attribute of a symbol (or of a macro instruction operand that is a symbol) is as described in Part I of this publication. Reference to the length attribute of a variable symbol is illegal except for symbolic parameters in SETA, SETB and AIF statements. If the basic L' attribute is desired, it may be obtained as follows:

```
&A  SETC   'Z'
&B  SETC   'L'''
    MVC    &A.(&B&A),X
```

After generation, this would result in

```
    MVC    Z(L'Z),X
```

Conditional assembly instructions must not refer to the length attributes of symbols or macro instruction operands whose type attributes are the letters M, N, O, T, W, or $.

Scaling and integer attributes are provided for symbols that name fixed-point, floating-point, and decimal fields.

Fixed and Floating Point: The scaling attribute of a fixed-point or floating-point number is the value given by the scale modifier. The integer attribute is a function of the scale and length attributes of the number.

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 the number of decimal digits to the left of the assumed decimal point after the number is assembled.

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,L, 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 programmer may refer to the count attribute of macro instruction operands only.

The value of the count attribute is equal to the number of characters in the macro instruction operand. It includes all characters in the operand, but does not include the delimiting commas. The count attribute of an omitted operand is zero. These rules are illustrated by the following examples:

Operand	Count Attribute
ALPHA	5
(JUNE,JULY,AUGUST)	18
2(10,12)	8
A(2)	4
'A''B'	6
' '	3
''	2

If a macro instruction operand contains variable symbols, the characters that replace the variable symbols, rather than the variable symbols, 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 part 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 one plus the number of commas that indicate the end of an operand in the sublist.

The following examples illustrate this 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 one. If the macro instruction operand is omitted, the number attribute is zero.

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 one; i.e., 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: The integer attribute of a Type D or E floating-point number is equal to two times the difference between the length attribute of the number and one, minus the scaling attribute; i.e., I'=2*(L'-1)-S'.

Because of its low order characteristic, the integer attribute of a Type L constant with a length greater than 8 bytes is two less than the value indicated in the formula above. The integer attribute of a Type L constant with a length of 8 bytes or less is the same as the value indicated in the formula above.

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 one; i.e., 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; i.e., I'=L'-S'.

Each of the following statements defines a decimal field. The length attribute of FIRST is 2, the scaling attribute is 2, and the integer attribute is 1. The length attribute of SECOND is 3, the scaling attribute is 0, and the integer attribute is 3. The length attribute of THIRD is 4, the scaling attribute is 2, and the integer attribute is 2. The length attribute of FOURTH is 3, the scaling attribute is 2, and the integer attribute is 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

The name field of a statement may contain a sequence symbol. Sequence symbols 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 is considered to be local to a macro definition.

A sequence symbol may be used in the name field of any statement that does not contain a symbol or SET symbol except a prototype statement, a MACRO, LCLA, LCLB, LCLC, GBLA, GBLB, GBLC, ACTR, ICTL, ISEQ, or COPY instruction.

A sequence symbol consists of a period followed by one through seven letters and/or digits, the first of which must be a letter.

The following are valid sequence symbols:

```
.READER   .A23456
.LOOP2    .X4F2
.N        .S4
```

The following are invalid sequence symbols:

```
CARDAREA  (first character is not
            a period)
.246B     (first character after
            period is not a letter)
.AREA2456 (more than seven characters
            after period)
.BCD%84   (contains a special character
            other than initial period)
.IN AREA  (contains a special
            character, i.e., 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.

The following example illustrates this 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 (statement 1) and the first model statement (statement 2). In the macro-instruction (statement 3) a sequence symbol (.SYM) corresponds to the symbolic parameter &NAME. &NAME is not replaced by .SYM, and, therefore, the generated statement (statement 4) does not contain an entry in the name field.

LCLA, LCLB, LCLC -- DEFINE LOCAL SET SYMBOLS

The format of these instructions is:

Name	Operation	Operand
Blank	LCLA, LCLB, or LCLC	One or more variable symbols, that are to be used as 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, respectively. The SETA, SETB, and SETC symbols are assigned the initial values of 0, 0, and null character value, respectively.

The programmer should not define any SET symbol whose first four characters are &SYS.

All LCLA, LCLB, or LCLC instructions in a macro definition must appear immediately after the prototype statement and GBLA, GBLB or GBLC instructions. All LCLA, LCLB, or LCLC instructions outside macro definitions must appear after all macro definitions in the source program, after all GBLA, GBLB, and GBLC instructions outside macro definitions, before all conditional assembly instructions, and PUNCH and REPRO statements outside macro definitions, and before the first control section of the program.

SETA -- SET ARITHMETIC

The SETA instruction may be used to assign an arithmetic value to a SETA symbol. The format of this instruction is:

Name	Operation	Operand
A SETA symbol	SETA	An 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 $+2^{31}-1$, respectively.

The expression may consist of one term or an arithmetic combination of terms. The terms that may be used alone or in combination with each other are self-defining terms, variable symbols, and the length, scaling, integer, count, and number attributes. Self-defining terms are described in Part I of this publication.

Note: A SETC variable symbol may appear in a SETA expression only if the value of the SETC variable is one to eight decimal digits. The 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 + (addition), - (subtraction), * (multiplication), and / (division).

An expression may not contain two terms or two operators in succession, nor may it begin with an operator.

The following are valid operand fields of SETA instructions:

```
&AREA+X'2D'    I'&N/25
&BETA*10       &EXIT-S'&ENTRY+1
L'&HERE+32     29
```

The following are invalid operand fields of SETA instructions:

```
&AREAX'C'      (two terms in succession)
&FIELD+-       (two operators in succession)
-&DELTA*2      (begins with an operator)
*+32           (begins with an operator;
                 two operators in succession)
NAME/15        (NAME is not a 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.

The following evaluation procedure is used:

1. Each term is given its numerical value.
2. The arithmetic operations are performed moving from left to right. However, multiplication and/or 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. Only five levels of parentheses are allowed and an expression may not consist of more than 16 terms. Parentheses required for sublist notation, substring notation, and subscript notation count toward this limit.

The following are examples of SETA instruction operand fields that contain parenthesized sequences of terms.

```
(L'&HERE+32)*29
&AREA+X'2D'/(&EXIT-S'&ENTRY+1)
&BETA*10*(I'&N/25/(&EXIT-S'&ENTRY+1))
```

The parenthesized portion or portions of an arithmetic expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first.

Using SETA Symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic expression. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is converted to an unsigned integer, with leading zeros removed. If the value is zero, it is converted to a single zero.

The following example illustrates this 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 to the SETA symbols &A and &B the arithmetic values +10 and +12, respectively. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2. When &C is used in statement 5, the arithmetic value -2 is converted to the unsigned integer 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the unsigned integer 8.

The following example shows how the value assigned to a SETA symbol may be changed in a macro definition.

	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 the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the unsigned integer 5. Statement 3 assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the unsigned integer 8, instead of 5.

A SETA symbol may be used with a symbolic parameter to refer to an operand in an operand sublist. If a SETA symbol is used for this purpose it must have been assigned a positive value.

Any expression that may be used in the operand field of a SETA instruction may be used to refer to an operand in an operand sublist.

Sublists are described in 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 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 (statement 1). The corresponding characters, (A,B,C,D,E), of the macro-instruction (statement 4) are a sublist. Statement 2 assigns to &LAST the arithmetic value +5, which is equal to the number of operands in the sublist. Therefore, in statement 3, &NUMBER(&LAST) is replaced by the fifth operand of the sublist.

SETC -- SET CHARACTER

The SETC instruction is used to assign a character value to a SETC symbol. The format of this instruction is:

Name	Operation	Operand
A SETC symbol	SETC	One operand, of the form 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 zeros removed. If the value is zero, one decimal zero is used.

Type Attribute

The character value assigned to a SETC symbol may be a type attribute. If the type attribute is used, it must appear alone in the operand field. The following example assigns to the SETC symbol &TYPE the letter that is the type attribute 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 (up to 255) characters enclosed in apostrophes.

The first eight characters in a character value enclosed in apostrophes in the operand field are assigned to the SETC symbol in the name field. The maximum size character value that can be assigned to a SETC symbol is eight characters.

Evaluation of Character Expressions: The following statement assigns the character value AB%4 to the SETC symbol &ALPHA:

Name	Operation	Operand
&ALPHA	SETC	'AB%4'

More than one character expression may be concatenated into a single character expression by placing a period between the terminating apostrophe of one character expression and the opening apostrophe of the next character 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.

The following statement assigns the character value L'SYMBOL to the SETC symbol &LENGTH.

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. Both ampersands become part of the character value assigned to the SETC symbol. They are not replaced by a single ampersand.

The following statement assigns the character 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. Substring character values 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: (1) the character value itself, and (2) the part of the character value he wants to assign to the SETC symbol. The combination of (1) and (2) in the operand field of a SETC instruction is called a substring notation. The character value that is assigned to the SETC symbol in the name field is called a substring character value.

Substring notation consists of a character expression, immediately followed by two arithmetic expressions that are separated from each other by a comma and are enclosed in parentheses. The two arithmetic 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. If a substring asks for more characters than are in the character string only the characters in the string will be assigned.

The maximum size substring character value that can be assigned to a SETC symbol is eight characters. The maximum size character expression the substring character value can be chosen from is 255 characters. If a value greater than 8 is specified, the leftmost 8 characters will be used.

The following are valid substring notations:

```
'&ALPHA'(2,5)
'AB%4'(&AREA+2,1)
'&ALPHA.RST'(6,&A)
'ABC&GAMMA'(&A,&AREA+2)
```

The following are invalid substring notations:

```
'&BETA' (4,6)
      (blanks between character value
        and arithmetic expressions)
'L''SYMBOL'(142-&XYZ)
      (only one arithmetic expression)
'AB%4&ALPHA'(8 &FIELD*2)
      (arithmetic expressions
        not separated by a 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.

For example, consider the following 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 statements 2 and 3, &PREFIX is replaced by FIELD.

The following example shows how the value assigned to a SETC symbol may be changed in a macro definition.

	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.

The following example illustrates the use of a substring notation 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.

Concatentating 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, the two may be concatenated by placing a period between the terminating apostrophe of the character expression and the opening apostrophe of the substring notation.

For example, if &ALPHA has been assigned the character value AB%4, and &BETA has been assigned the character value ABCDEF, then the following statement assigns &GAMMA the character value AB%4BCD.

Name	Operation	Operand
&GAMMA	SETC	'&ALPHA'.'&BETA'(2,3)

If a substring notation precedes a character expression or another substring notation, the two may be concatenated by writing the opening apostrophe of the second item immediately after the closing parenthesis of the substring notation.

The programmer may optionally place a period between the closing parenthesis of a substring notation and the opening apostrophe of the next item in the operand field.

If &ALPHA has been assigned the character value AB%4, and &ABC has been assigned the character value 5RS, either of the following statements may be used to assign &WORD the character value AB%45RS.

Name	Operation	Operand
&WORD	SETC	'&ALPHA'(1,4)'&ABC'
&WORD	SETC	'&ALPHA'(1,4)'&ABC'(1,3)

If a SETC symbol is used in the operand field of a SETA instruction, the character value assigned to the SETC symbol must be one to eight decimal digits.

If a SETA symbol is used in the operand field of a SETC statement, the arithmetic value is converted to an unsigned integer with leading zeros removed. If the value is zero, it is converted to a single zero.

SETB -- SET BINARY

The SETB instruction may be used to assign the binary value 0 or 1 to a SETB symbol. The format of this instruction is:

Name	Operation	Operand
A SETB symbol	SETB	A 0 or a 1 enclosed or not enclosed in parentheses, or a logical expression enclosed in parentheses

The operand field may contain a 0 or a 1 or a logical expression enclosed in parentheses. A logical expression is evaluated to determine if it is true or false; the SETB symbol in the name field is then assigned the binary value 1 or 0 corresponding to true or false, respectively.

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 used to combine the terms of an expression are AND, OR, and NOT.

An expression may not contain two terms in succession. 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 the operator NOT. It may not begin with the 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), LE (less than or equal), and GE (greater than or equal).

Any expression that may be used in the operand field of a SETA instruction, may be used as an arithmetic expression in the operand field of a SETB instruction. Anything 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. This includes 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 unequal size, then the smaller one will always compare less than the larger one.

The relational and logical operators must be immediately preceded and followed by at least one blank or other special character. Each relation may or may not be enclosed in parentheses. If a relation is not enclosed in parentheses, it must be separated from the logical operators by at least one blank or other special character.

The following 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)
('&C'EQ'MB')
(O)
```

The following 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)
                (the NOT operator must be
                 preceded by AND or OR)
(AND T'&P12 EQ 'F')
                (expression begins with AND)
```

Evaluation of Logical Expressions

The following procedure is used to evaluate a logical expression in the operand field of a SETB instruction:

1. Each term (i.e., arithmetic relation, character relation, or SETB symbol) is evaluated and given its logical value (true or false).
2. The logical operations are performed moving from left to right. However, NOTs are performed before ANDs, and ANDs are performed 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.

The following are examples of SETB instruction operand fields that contain parenthesized sequences of terms.

```
(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 the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first. Five levels of parentheses are permissible.

Using SETB Symbols

The logical value assigned to a SETB symbol is used for the SETB symbol appearing in the operand field of an AIF instruction or another SETB instruction.

If a SETB symbol is used in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of AIF and SETB instructions, the binary values 1 (true) and 0 (false) are converted to the arithmetic values +1 and +0, respectively.

If a SETB symbol is used in the operand field of a SETC instruction, in character relations in the operand fields of AIF and SETB instructions, or in any other statement, the binary values 1 (true) and 0 (false), are converted to the character values 1 and 0, respectively.

The following example illustrates these rules. It is assumed that L'&TO EQ 4 is true, and S'&TO EQ 0 is false.

	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. Therefore, the character value 0 is substituted for &B2 in statement 4.

AIF -- CONDITIONAL BRANCH

The AIF instruction is used to conditionally alter the sequence in which source program statements or macro-definition statements are processed by the assembler. The assembler assigns a maximum count of 4096 AIF and AGO branches that may be executed in the source program or in a macro-definition. When a macro-definition calls an inner macro-definition, the current value of the count is saved and a new count of 4096 is set up for the inner macro-definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored. The format of this instruction is:

Name	Operation	Operand
A sequence symbol or blank	AIF	A logical expression enclosed in parentheses, immediately followed by a 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 the expression is true, the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is 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, then the sequence symbol in the operand field must appear in the name field of a statement in the definition. If an AIF instruction appears outside macro-definitions, then the sequence symbol in the operand field must appear in the name field of a statement outside macro-definitions.

The following are valid operand fields of AIF instructions:

```
(&AREA+X'2D' GT 29).READER
(T'&P12 EQ 'F').THERE
('&FIELD3' EQ'').NO3
```

The following 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
                      expression and se-
                      quence symbol)
```

The following macro-definition may be used to generate the statements needed to move a full-word 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 has the value true if the type attributes of the two macro-instruction operands are not equal. If the type attributes are equal, the expression has the logical value false.

Therefore, if the type attributes are not equal, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attributes are equal, statement 2 (the next sequential statement) is processed.

The logical expression in the operand field of statement 2 has the value true if the type attribute of the first macro-instruction operand is not the letter F. If the type attribute is the letter F, the expression has the logical value false.

Therefore, if the type attribute is not the letter F, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attribute is the letter F, statement 3 (the next sequential statement) is processed.

AGO -- UNCONDITIONAL BRANCH

The AGO instruction is used to unconditionally alter the sequence in which source program or macro-definition statements are processed by the assembler. The assembler assigns a maximum count of 4096 AIF and AGO branches that may be executed in the source program or in a macro-definition. When a macro-definition calls an inner macro-definition, the current value of the count is saved and a new count of 4096 is set up for the inner macro-definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored. The format of this instruction is:

Name	Operation	Operand
A sequence symbol or blank	AGO	A sequence symbol

The statement named by the sequence symbol in the operand field is the next statement processed by the assembler.

The statement named by the sequence symbol may precede or follow the AGO instruction.

If an AGO instruction is part of a macro-definition, then the sequence symbol in the operand field must appear in the name field of a statement that is in that definition. If an AGO instruction appears outside macro-definitions, then the sequence symbol in the operand field must appear in the name field of a statement outside macro-definitions.

The following example 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 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler.

Statement 2 is used to indicate to the assembler that the next statement to be processed is statement 4 (the statement named by sequence symbol .END).

ACTR -- CONDITIONAL ASSEMBLY LOOP COUNTER

The ACTR instruction is used to assign a maximum count (different from the standard count of 4096) to the number of AGO and AIF branches executed within a macro-definition or within the source program. The format of this instruction is as follows:

Name	Operation	Operand
Blank	ACTR	Any valid SETA expression

This statement, which can only occur immediately after the global and local declarations, causes a counter to be set to the value in the operand field. The counter is checked for zero or a negative value; if it is not zero or negative, it is decremented by one each time an AGO or AIF branch is executed. If the count is zero before decrementing, the assembler will take one of two actions:

1. If processing is being performed inside a macro definition, the entire nest of macro definitions will be terminated and the next source statement will be processed.
2. If the source program is being processed, an END card will be generated.

An ACTR instruction in a macro-definition affects only that definition; it has no effect on the number of AIF and AGO branches that may be executed in macro-definitions called.

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
A sequence symbol	ANOP	Blank

If the programmer wants to use an AIF or AGO instruction 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 the programmer 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, the programmer must place an ANOP instruction before the statement and then branch to the ANOP instruction. This has the same effect as branching to the statement immediately after the ANOP instruction.

The following example 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 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, statement 4 should be processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (statement 3) must be placed before statement 4.

Then, if the type attribute of the first operand is the letter F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE retains its initial null character value because the SETC instruction is not processed. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, the statement following the ANOP instruction.

CONDITIONAL ASSEMBLY ELEMENTS

The following chart summarizes the elements that can be used in each conditional assembly instruction. Each row in this chart indicates which elements can be used in a single conditional assembly instruction. Each column is used to indicate 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, and 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[3]		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
ACTR	O	O	O	O[3]		O	O	O	O	O	

[1] Only in character relations
[2] Only in arithmetic relations
[3] Only if one to eight decimal digits

Abbreviations

N	is Name	L'	is Length Attribute	K'	is Count Attribute
O	is Operand	S'	is Scaling Attribute	N'	is Number Attribute
S.P.	is Symbolic Parameter	I'	is Integer Attribute	S.S.	is Sequence Symbol

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 System/360 macro definitions.

MEXIT -- MACRO DEFINITION EXIT

The MEXIT instruction is used to indicate to the assembler that it should terminate processing of a macro-definition. The format of this instruction is:

Name	Operation	Operand
A sequence symbol or blank	MEXIT	Blank

The MEXIT instruction may only be used in a macro-definition.

If the assembler processes an MEXIT instruction that is 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 an MEXIT instruction that is in a macro-definition corresponding to a second or third level macro-instruction, the next statement processed by the assembler is the next statement after the second or third level macro-instruction instruction in the macro-definition, respectively.

MEXIT should not be confused with MEND. MEND 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.

The following example 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 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, the assembler processes the remainder of the macro-definition starting with statement 3. If the type attribute is not the letter F, the next statement processed by the assembler is statement 2. Statement 2 indicates to the assembler that it is to terminate processing of the macro-definition.

MNOTE -- REQUEST FOR ERROR MESSAGE

The MNOTE instruction may be used to request the assembler to generate an error message. The format of this instruction is:

Name	Operation	Operand
A sequence symbol, variable symbol or blank	MNOTE	A severity code, followed by a comma, followed by any combination of characters enclosed in apostrophes

The operand of the MNOTE instruction may also be written using one of the following forms:

Operand
severity-code, 'message'
,'message'
'message'

The MNOTE instruction may only be used in a macro-definition. Variable symbols may be used to generate the MNOTE mnemonic operation code, the severity code, and the message.

The severity code may be a decimal integer from 0 through 255 or an asterisk. If it is omitted, 1 is assumed. The severity code indicates the severity of the error, a higher severity code indicating a more serious error.

When MNOTE * occurs, the statement in the operand field will be printed as a comment.

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. If any variable symbols are used in the operand field of an MNOTE instruction, they 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.

The following example illustrates the use of the MNOTE instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
		MNOTE	*,'MOVE MACRO GEN'
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 is used to determine 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. If they are not, statement 4 is the next statement processed by the assembler. Statement 4 causes an error message indicating the type attributes are not the same to be printed in the source program listing.

Statement 2 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the attribute is not the letter F, statement 5 is the next statement processed by the assembler. Statement 5 causes an error message indicating the type attribute is not F to be printed in the source program listing.

GLOBAL AND LOCAL VARIABLE SYMBOLS

The following are local variable symbols:

1. Symbolic parameters.
2. Local SET symbols.
3. 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. Hereinafter, SET symbols defined by LCLA, LCLB, and LCLC instructions will be called local SET symbols.

Global SET symbols communicate values between statements in one or more macro-definitions and statements outside macro-definitions. 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 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 as a global SET symbol outside macro-definitions.

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 are 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 format of these instructions is:

Name	Operation	Operand
Blank	GBLA, GBLB, or GBLC	One or more variable symbols that are to be used as SET symbols, separated by commas

The GBLA, GBLB, and GBLC instructions define global SETA, SETB, and SETC symbols, respectively, 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 the 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 immediately follow the prototype statement, or another GBLA, GBLB, or GBLC instruction. GBLA, GBLB, and GBLC instructions outside macro-definitions must appear after all macro-definitions in the source program, before all conditional assembly instructions and PUNCH and REPRO statements outside macro-definitions, and before the first control section of the program.

All GBLA, GBLB, and GBLC instructions in a macro-definition must appear before all LCLA, LCLB, and LCLC instructions in that macro-definition. All GBLA, GBLB, and GBLC instructions outside macro-definitions must appear before all LCLA, LCLB, and LCLC instructions outside macro-definitions.

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: This example illustrates how the same SET symbol can be used to communicate (1) values between statements in the same macro-definitions, and (2) different values between statements 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

&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 (which is in the macro definition) does not affect the value used for &A in statements 5 and 6 (which are outside macro definitions). Moreover, the use of LOADA between statements 5 and 6 will alter &A from its previous value as a local symbol within that macro definition since the first act of the macro definition is to LCLA &A to zero.

Example 2: This example illustrates how a SET symbol can be used to communicate values between statements that are part of a macro-definition and statements outside macro-definitions.

	Name	Operation	Operand
		MACRO	
	&NAME	LOADA	
1		GBLA	&A
2	&NAME	LR	15,&A
3	&A	SETA	&A+1
		MEND	
4		GBLA	&A
	FIRST	LOADA	
5		LR	15,&A
		LOADA	
6		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 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 global SETA symbol in the macro-definition and outside macro-definitions, it is the same SETA symbol in both cases. Therefore, statement 3 (which is in the macro-definition) affects the value used for &A in statements 5 and 6 (which are outside macro-definitions).

Example 3: This example illustrates how the same SET symbol can be used to communicate: (1) values between statements in one macro-definition, and (2) different values between statements in a different macro-definition.

&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 macro-definition. Therefore, statement 3 (which is in one macro-definition) does not affect the value used for &A in statement 5 (which is 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: This example illustrates how a SET symbol can be used to communicate values between statements that are part of two different macro-definitions.

	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

&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).

Since &A is a global SETA symbol in each macro-definition, it is the same SETA symbol in each macro-definition. Therefore, statement 3 (which is in one macro-definition) affects the value used for &A in statement 5 (which is in the other macro-definition). Similarly, statement 6 affects the value used for &A in statement 2.

Example 5: This example illustrates how the same SET symbol can be used to communicate: (1) values between statements in two different macro-definitions, and (2) different values between statements outside macro-definitions.

	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

	Name	Operation	Operand
	FIRST	LR	15,0
		LR	15,1
		LR	15,0
		LR	15,2
		LR	15,3
		LR	15,0
		END	FIRST

&A is defined as a global SETA symbol in two different macro-definitions (statements 1 and 4), but it is defined as a local SETA symbol 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 symbol in each macro-definition, it is the same SETA symbol in each macro-definition. However, since &A is a local SETA symbol outside macro-definitions, it is a different SETA symbol outside macro-definitions.

Therefore, statement 3 (which is in one macro-definition) affects the value used for &A in statement 5 (which is in the other macro-definition), but it does not affect the value used for &A in statements 8 and 9 (which are 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.

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 nonsubscripted 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 SET 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 may not be 0 or negative.

The following are valid subscripted SET symbols.

```
&READER(17)
&A23456(&S4)
&X4F2(25+&A2)
```

The following 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 nonsubscripted 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 a SETA, SETB, or SETC symbol is 2500.

A subscripted SET symbol may be used only if the declaration was subscripted; a nonsubscripted SET symbol may be used only if the declaration had no subscript.

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, &PSW and &TSW each have 230 binary variables.

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 part of a macro-definition, (and &A was defined as a SETA symbol in the same definition), the following statements could be part 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 nonsubscripted SETA symbol &A. Statements 2 and 3 then assign the binary value 0 to subscripted SETB symbols &PSW(5) and &TSW(9), respectively. 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, respectively.

SYSTEM VARIABLE SYMBOLS

System variable symbols are local variable symbols that are assigned values automatically by the assembler. There are three system variable symbols: &SYSNDX, &SYSECT, and &SYSLIST. System variable symbols 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 instructions.

&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 four-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 four-digit number of the macro-instruction being processed, including leading zeros.

If &SYSNDX appears in arithmetic expressions (e.g., 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 example in the next column illustrates these rules. It is assumed that the first macro-instruction processed, OUTER 1, is the 106th macro-instruction processed by the assembler.

Statement 7 is the 106th macro-instruction processed. Therefore, &SYSNDX is assigned the number 0106 for that macro-instruction. The number 0106 is substituted for &SYSNDX when it is used in statements 4 and 6. Statement 4 is used to assign the character value 0106 to the SETC symbol &NDXNUM. Statement 6 is used to create the unique name B0106.

	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'

Statement 5 is the 107th macro-instruction processed. Therefore, &SYSNDX is assigned the number 0107 for that macro-instruction. The number 0107 is substituted for &SYSNDX when it is used in statements 1 and 3. The number 0106 is substituted for the global SETC symbol &NDXNUM in statement 2.

Statement 8 is the 108th macro-instruction processed. Therefore,each occurrence of &SYSNDX is replaced by the number 0108. For example, statement 6 is used to create the unique name B0108.

When statement 5 is used to process the 108th macro-instruction, statement 5 becomes the 109th macro-instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0109. For example, statement 1 is used to create the unique name A0109.

&SYSECT -- Current Control Section

The system variable symbol &SYSECT may be used to represent the name of the control section in which a macro-instruction appears. For each inner and outer macro-instruction processed by the assembler, &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 for &SYSECT for 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.

Statement 8 is the last CSECT, DSECT, or START statement processed before statement 9 is processed. Therefore, &SYSECT is assigned the value MAINPROG for macro-instruction OUTER1 in statement 9. MAINPROG is substituted for &SYSECT when it appears in statement 6.

Statement 3 is the last CSECT, DSECT, or START statement processed before statement 4 is processed. Therefore, &SYSECT is assigned the value CSOUT1 for macro-instruction INNER in statement 4. CSOUT1 is substituted for &SYSECT when it appears in statement 2.

Statement 1 is used to generate a CSECT statement for statement 4. This is the last CSECT, DSECT, or START statement that appears before statement 5. Therefore, &SYSECT is assigned the value INA for macro-instruction INNER in statement 5. INA is substituted for &SYSECT when it appears in statement 2.

	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 1 is used to generate a CSECT statement for statement 5. This is the last CSECT, DSECT, or START statement that appears before statement 10. Therefore, &SYSECT is assigned the value INB for macro instruction OUTER2 in statement 10. INB is substituted for &SYSECT when it appears in statement 7.

&SYSLIST -- Macro Instruction Operand

The system variable symbol &SYSLIST provides the programmer with an alternative to symbolic parameters for referring to positional macro instruction operands.

&SYSLIST and symbolic parameters may be used in the same macro definition.

&SYSLIST(n) may be used to refer to the nth positional 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 allowed in the operand field of a SETA statement. M may be equal to or greater than 1 and N has a range of 1 to 200.

The type, length, scaling, integer, and count attributes of &SYSLIST(n) and &SYSLIST(n,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 positional 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 zero; if the nth operand is not a sublist, N' is one.

The following procedure is used to evaluate N'&SYSLIST:

1. A sublist is considered to be one operar
2. The count includes specifically omitted (by means of commas) operands.

Examples:

Macro Instruction	N'&SYSLIST
MAC K1=DS	0
MAC ,K1=DC	1
MAC FULL,,F,('1','2'),K1=DC	4
MAC ,	2
MAC	0

Attributes are discussed in Section 7 under "Attributes."

KEYWORD MACRO DEFINITIONS AND INSTRUCTIONS

Keyword macro definitions provide the programmer with an alternate way of preparing macro definitions.

A keyword macro definition enables a programmer to reduce the number of operands in each macro instruction that corresponds to the definition, and to write the operands in any order.

The macro instructions that correspond to the macro definitions described in Section 7 (hereinafter called positional macro instructions and positional macro definitions, respectively) require the operands to 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 that appear 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 only write 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
A symbolic parameter or blank	A symbol	One or more operands of the 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. This value must not include a keyword.

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 are detailed in Section 8.

The following are valid keyword prototype operands.

```
&READER=
&LOOP2=SYMBOL
&S4==F'4096'
```

The following 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
A symbol, sequence symbol, or blank	Mnemonic operation code	Zero or more operands of the form described below, separated by commas

Each operand consists of a keyword immediately followed by an equal sign and an optional value which may not include a keyword. Anything that may be used as an 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 are 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 that appears 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.

The following are valid keyword macro instruction operands.

```
LOOP2=SYMBOL
S4==F'4096'
TO=
```

The following 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 does not have to 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.

<u>Note:</u> If a standard value is a self-defining term the type attribute assigned to the standard value is the letter N. If a standard value is omitted the type attribute assigned to the standard value is the letter O. All other standard values are assigned the type attribute U.

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

Statement 1 assigns the standard values 2 and S to the symbolic parameters &R and &A, respectively. Statement 6 assigns the values FA, FB, and THERE to the keywords T, F, and A, respectively. 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 (statement 1), and the corresponding characters (HERE) of the macro-instruction (statement 6) are a symbol, &N is replaced by HERE in statement 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, of statement 1, and statement 6 contains the keyword (T) that corresponds to &T, the value assigned to T (FA) replaces &T in statement 4. Similarly, FB and THERE replace &F and &A in statement 3 and in statements 2 and 5, respectively. 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 statements 2, 3, 4, and 5.

<u>Operand Sublists:</u> The value assigned to a keyword and the standard value assigned to a symbolic parameter may be an operand sublist. Anything that may be used as an 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 are detailed in Section 8 under "Operand Sublists."

<u>Keyword Inner Macro Instructions:</u> 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 the same way as positional macro definitions, except that the prototype statement is written differently. If &SYSLIST is used, it refers only to the positional operands in the prototype. Subscripting past the last positional parameter will yield an empty string and a type attribute of "O". The rules for preparing positional macro definitions are in Section 7.

Mixed-Mode Prototype

The format of this statement is:

Name	Operation	Operand
A symbolic parameter or blank	A symbol	Two or more operands of the form described below, separated by commas

The operands must be valid operands of positional and keyword prototype statements. All the positional operands must precede the first keyword operand. The rules for forming positional operands are discussed in Section 7, under "Macro-Instruction Prototype." The rules for forming keyword operands are discussed above under "Keyword 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 of a mixed-mode macro-instruction is:

Name	Operation	Operand
A symbol, sequence symbol, or blank	Mnemonic operation code	Zero or more operands of the form described below, separated by commas

The operand field consists of two parts. The first part corresponds to the positional prototype operands. This part of the operand field is written in the same way that the operand field of a positional macro-instruction is written. The rules for writing positional macro-instructions are in Section 8.

The second part of the operand field corresponds to the keyword prototype operands. This part of the operand field is written in the same way that the operand field of a keyword macro-instruction is written. The rules for writing keyword macro-instructions are described above under "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 (statement 1) contains three positional operands (&TY,&P, and &R) and two keyword operands (&TO and &F). In the macro instruction (statement 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 the other System/360 assemblers having macro language facilities may be used with the Operating System/360 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 Operating System/360 assembler the same way that the AIF and AGO instructions are processed. AIFB and AGOB instructions will cause the count set up by the ACTR instructions to be decremented in exactly the same way as the AGO and AIF instructions.

APPENDIXES

APPENDIX A: CHARACTER CODES

APPENDIX B: HEXADECIMAL-DECIMAL NUMBER CONVERSION TABLE

APPENDIX C: MACHINE-INSTRUCTION FORMAT

APPENDIX D: MACHINE-INSTRUCTION MNEMONIC OPERATION CODES

APPENDIX E: ASSEMBLER INSTRUCTIONS

APPENDIX F: SUMMARY OF CONSTANTS

APPENDIX G: MACRO LANGUAGE SUMMARY

APPENDIX H: SAMPLE PROGRAM

APPENDIX I: ASSEMBLER LANGUAGES--FEATURES COMPARISON CHART

APPENDIX J: SAMPLE MACRO DEFINITIONS

8-Bit BCD Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
00000000	12,0,9,8,1	0	00	
00000001	12,9,1	1	01	
00000010	12,9,2	2	02	
00000011	12,9,3	3	03	
00000100	12,9,4	4	04	
00000101	12,9,5	5	05	
00000110	12,9,6	6	06	
00000111	12,9,7	7	07	
00001000	12,9,8	8	08	
00001001	12,9,8,1	9	09	
00001010	12,9,8,2	10	0A	
00001011	12,9,8,3	11	0B	
00001100	12,9,8,4	12	0C	
00001101	12,9,8,5	13	0D	
00001110	12,9,8,6	14	0E	
00001111	12,9,8,7	15	0F	
00010000	12,11,9,8,1	16	10	
00010001	11,9,1	17	11	
00010010	11,9,2	18	12	
00010011	11,9,3	19	13	
00010100	11,9,4	20	14	
00010101	11,9,5	21	15	
00010110	11,9,6	22	16	
00010111	11,9,7	23	17	
00011000	11,9,8	24	18	
00011001	11,9,8,1	25	19	
00011010	11,9,8,2	26	1A	
00011011	11,9,8,3	27	1B	
00011100	11,9,8,4	28	1C	
00011101	11,9,8,5	29	1D	
00011110	11,9,8,6	30	1E	
00011111	11,9,8,7	31	1F	
00100000	11,0,9,8,1	32	20	
00100001	0,9,1	33	21	
00100010	0,9,2	34	22	
00100011	0,9,3	35	23	
00100100	0,9,4	36	24	
00100101	0,9,5	37	25	
00100110	0,9,6	38	26	
00100111	0,9,7	39	27	
00101000	0,9,8	40	28	
00101001	0,9,8,1	41	29	
00101010	0,9,8,2	42	2A	
00101011	0,9,8,3	43	2B	
00101100	0,9,8,4	44	2C	
00101101	0,9,8,5	45	2D	
00101110	0,9,8,6	46	2E	
00101111	0,9,8,7	47	2F	
00110000	12,11,0,9,8,1	48	30	
00110001	9,1	49	31	
00110010	9,2	50	32	

8-Bit BCD Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
00110011	9,3	51	33	
00110100	9,4	52	34	
00110101	9,5	53	35	
00110110	9,6	54	36	
00110111	9,7	55	37	
00111000	9,8	56	38	
00111001	9,8,1	57	39	
00111010	9,8,2	58	3A	
00111011	9,8,3	59	3B	
00111100	9,8,4	60	3C	
00111101	9,8,5	61	3D	
00111110	9,8,6	62	3E	
00111111	9,8,7	63	3F	
01000000		64	40	blank
01000001	12,0,9,1	65	41	
01000010	12,0,9,2	66	42	
01000011	12,0,9,3	67	43	
01000100	12,0,9,4	68	44	
01000101	12,0,9,5	69	45	
01000110	12,0,9,6	70	46	
01000111	12,0,9,7	71	47	
01001000	12,0,9,8	72	48	
01001001	12,8,1	73	49	
01001010	12,8,2	74	4A	
01001011	12,8,3	75	4B	. (period)
01001100	12,8,4	76	4C	<
01001101	12,8,5	77	4D	(
01001110	12,8,6	78	4E	+
01001111	12,8,7	79	4F	
01010000	12	80	50	&
01010001	12,11,9,1	81	51	
01010010	12,11,9,2	82	52	
01010011	12,11,9,3	83	53	
01010100	12,11,9,4	84	54	
01010101	12,11,9,5	85	55	
01010110	12,11,9,6	86	56	
01010111	12,11,9,7	87	57	
01011000	12,11,9,8	88	58	
01011001	11,8,1	89	59	
01011010	11,8,2	90	5A	
01011011	11,8,3	91	5B	$
01011100	11,8,4	92	5C	*
01011101	11,8,5	93	5D	)
01011110	11,8,6	94	5E	
01011111	11,8,7	95	5F	
01100000	11	96	60	-
01100001	0,1	97	61	/
01100010	11,0,9,2	98	62	
01100011	11,0,9,3	99	63	
01100100	11,0,9,4	100	64	
01100101	11,0,9,5	101	65	
01100110	11,0,9,6	102	66	
01100111	11,0,9,7	103	67	
01101000	11,0,9,8	104	68	
01101001	0,8,1	105	69	
01101010	12,11	106	6A	
01101011	0,8,3	107	6B	(comma)

8-Bit BCD Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
01101100	0,8,4	108	6C	%
01101101	0,8,5	109	6D	
01101110	0,8,6	110	6E	
01101111	0,8,7	111	6F	
01110000	12,11,0	112	70	
01110001	12,11,0,9,1	113	71	
01110010	12,11,0,9,2	114	72	
01110011	12,11,0,9,3	115	73	
01110100	12,11,0,9,4	116	74	
01110101	12,11,0,9,5	117	75	
01110110	12,11,0,9,6	118	76	
01110111	12,11,0,9,7	119	77	
01111000	12,11,0,9,8	120	78	
01111001	8,1	121	79	
01111010	8,2	122	7A	
01111011	8,3	123	7B	#
01111100	8,4	124	7C	@
01111101	8,5	125	7D	' (apostrophe)
01111110	8,6	126	7E	=
01111111	8,7	127	7F	
10000000	12,0,8,1	128	80	
10000001	12,0,1	129	81	
10000010	12,0,2	130	82	
10000011	12,0,3	131	83	
10000100	12,0,4	132	84	
10000101	12,0,5	133	85	
10000110	12,0,6	134	86	
10000111	12,0,7	135	87	
10001000	12,0,8	136	88	
10001001	12,0,9	137	89	
10001010	12,0,8,2	138	8A	
10001011	12,0,8,3	139	8B	
10001100	12,0,8,4	140	8C	
10001101	12,0,8,5	141	8D	
10001110	12,0,8,6	142	8E	
10001111	12,0,8,7	143	8F	
10010000	12,11,8,1	144	90	
10010001	12,11,1	145	91	
10010010	12,11,2	146	92	
10010011	12,11,3	147	93	
10010100	12,11,4	148	94	
10010101	12,11,5	149	95	
10010110	12,11,6	150	96	
10010111	12,11,7	151	97	
10011000	12,11,8	152	98	
10011001	12,11,9	153	99	
10011010	12,11,8,2	154	9A	
10011011	12,11,8,3	155	9B	
10011100	12,11,8,4	156	9C	
10011101	12,11,8,5	157	9D	
10011110	12,11,8,6	158	9E	
10011111	12,11,8,7	159	9F	
10100000	11,0,8,1	160	A0	
10100001	11,0,1	161	A1	
10100010	11,0,2	162	A2	
10100011	11,0,3	163	A3	
10100100	11,0,4	164	A4	

8-Bit BCD Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
10100101	11,0,5	165	A5	
10100110	11,0,6	166	A6	
10100111	11,0,7	167	A7	
10101000	11,0,8	168	A8	
10101001	11,0,9	169	A9	
10101010	11,0,8,2	170	AA	
10101011	11,0,8,3	171	AB	
10101100	11,0,8,4	172	AC	
10101101	11,0,8,5	173	AD	
10101110	11,0,8,6	174	AE	
10101111	11,0,8,7	175	AF	
10110000	12,11,0,8,1	176	B0	
10110001	12,11,0,1	177	B1	
10110010	12,11,0,2	178	B2	
10110011	12,11,0,3	179	B3	
10110100	12,11,0,4	180	B4	
10110101	12,11,0,5	181	B5	
10110110	12,11,0,6	182	B6	
10110111	12,11,0,7	183	B7	
10111000	12,11,0,8	184	B8	
10111001	12,11,0,9	185	B9	
10111010	12,11,0,8,2	186	BA	
10111011	12,11,0,8,3	187	BB	
10111100	12,11,0,8,4	188	BC	
10111101	12,11,0,8,5	189	BD	
10111110	12,11,0,8,6	190	BE	
10111111	12,11,0,8,7	191	BF	
11000000	12,0	192	C0	
11000001	12,1	193	C1	A
11000010	12,2	194	C2	B
11000011	12,3	195	C3	C
11000100	12,4	196	C4	D
11000101	12,5	197	C5	E
11000110	12,6	198	C6	F
11000111	12,7	199	C7	G
11001000	12,8	200	C8	H
11001001	12,9	201	C9	I
11001010	12,0,9,8,2	202	CA	
11001011	12,0,9,8,3	203	CB	
11001100	12,0,9,8,4	204	CC	
11001101	12,0,9,8,5	205	CD	
11001110	12,0,9,8,6	206	CE	
11001111	12,0,9,8,7	207	CF	
11010000	11,0	208	D0	
11010001	11,1	209	D1	J
11010010	11,2	210	D2	K
11010011	11,3	211	D3	L
11010100	11,4	212	D4	M
11010101	11,5	213	D5	N
11010110	11,6	214	D6	O
11010111	11,7	215	D7	P
11011000	11,8	216	D8	Q
11011001	11,9	217	D9	R
11011010	12,11,9,8,2	218	DA	
11011011	12,11,9,8,3	219	DB	
11011100	12,11,9,8,4	220	DC	
11011101	12,11,9,8,5	221	DD	

8-Bit BCD Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
11011110	12,11,9,8,6	222	DE	
11011111	12,11,9,8,7	223	DF	
11100000	0,8,2	224	E0	
11100001	11,0,9,1	225	E1	
11100010	0,2	226	E2	S
11100011	0,3	227	E3	T
11100100	0,4	228	E4	U
11100101	0,5	229	E5	V
11100110	0,6	230	E6	W
11100111	0,7	231	E7	X
11101000	0,8	232	E8	Y
11101001	0,9	233	E9	Z
11101010	11,0,9,8,2	234	EA	
11101011	11,0,9,8,3	235	EB	
11101100	11,0,9,8,4	236	EC	
11101101	11,0,9,8,5	237	ED	
11101110	11,0,9,8,6	238	EE	
11101111	11,0,9,8,7	239	EF	
11110000	0	240	F0	0
11110001	1	241	F1	1
11110010	2	242	F2	2
11110011	3	243	F3	3
11110100	4	244	F4	4
11110101	5	245	F5	5
11110110	6	246	F6	6
11110111	7	247	F7	7
11111000	8	248	F8	8
11111001	9	249	F9	9
11111010	12,11,0,9,8,2	250	FA	
11111011	12,11,0,9,8,3	251	FB	
11111100	12,11,0,9,8,4	252	FC	
11111101	12,11,0,9,8,5	253	FD	
11111110	12,11,0,9,8,6	254	FE	
11111111	12,11,0,9,8,7	255	FF	

Special Graphic Characters

¢	Cent Sign	*	Asterisk	>	Greater-than Sign
.	Period, Decimal Point	)	Right Parenthesis	?	Question Mark
<	Less-than Sign	;	Semicolon	:	Colon
(	Left Parenthesis	¬	Logical NOT	#	Number Sign
+	Plus Sign	-	Minus Sign, Hyphen	@	At Sign
|	Vertical Bar, Logical OR	/	Slash	'	Prime, Apostrophe
&	Ampersand	,	Comma	=	Equal Sign
!	Exclamation Point	%	Percent	"	Quotation Mark
$	Dollar Sign	_	Underscore		

Examples	Type	Bit Pattern Bit Positions 01 23 4567	Hole Pattern Zone Punches	Hole Pattern Digit Punches
PF	Control Character	00 00 0100	12 -9 -	4
%	Special Graphic	01 10 1100	0 -	8 - 4
R	Upper Case	11 01 1001	11 -	9
a	Lower Case	10 00 0001	12 -0 -	1
	Control Character, function not yet assigned	00 11 0000	12 - 11 - 0 -9 -	8 - 1

The table in this appendix provides for direct conversion of decimal and hexadecimal numbers in these ranges:

Hexadecimal	Decimal
000 to FFF	0000 to 4095

Decimal numbers (0000-4095) are given within the 5-part table. The first two characters (high-order) of hexadecimal numbers (000-FFF) are given in the lefthand column of the table; the third character (x) is arranged across the top of each part of the table.

To find the decimal equivalent of the hexadecimal number 0C9, look for 0C in the left column, and across that row under the column for x = 9. The decimal number is 0201.

To convert from decimal to hexadecimal, look up the decimal number within the table and read the hexadecimal number by a combination of the hex characters in the left column, and the value for x at the top of the column containing the decimal number. For example, the decimal number 123 has the hexadecimal equivalent of 07B; the decimal number 1478 has the hexadecimal equivalent of 5C6.

For numbers outside the range of the table, add the following values to the table

Hexadecimal	Decimal
1000	4096
2000	8192
3000	12288
4000	16384
5000	20480
6000	24576
7000	28672
8000	32768
9000	36864
A000	40960
B000	45056
C000	49152
D000	53248
E000	57344
F000	61440

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00x	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01x	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02x	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03x	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04x	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05x	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06x	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07x	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08x	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09x	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0Ax	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0Bx	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0Cx	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0Dx	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0Ex	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0Fx	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10x	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11x	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12x	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13x	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14x	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15x	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16x	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17x	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18x	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19x	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1Ax	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1Bx	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1Cx	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1Dx	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1Ex	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1Fx	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20x	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21x	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22x	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23x	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24x	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25x	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26x	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27x	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28x	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29x	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2Ax	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2Bx	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2Cx	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2Dx	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2Ex	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2Fx	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30x	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31x	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32x	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33x	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34x	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35x	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36x	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37x	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38x	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39x	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3Ax	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3Bx	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3Cx	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3Dx	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3Ex	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3Fx	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40x	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41x	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42x	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43x	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44x	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45x	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46x	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47x	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48x	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49x	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4Ax	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4Bx	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4Cx	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4Dx	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4Ex	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4Fx	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50x	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51x	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52x	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53x	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54x	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55x	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56x	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57x	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58x	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59x	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5Ax	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5Bx	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5Cx	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5Dx	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5Ex	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5Fx	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60x	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61x	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62x	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63x	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64x	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65x	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66x	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67x	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68x	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69x	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6Ax	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6Bx	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6Cx	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6Dx	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6Ex	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6Fx	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70x	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71x	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72x	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73x	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74x	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75x	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76x	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77x	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78x	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79x	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7Ax	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7Bx	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7Cx	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7Dx	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7Ex	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7Fx	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80x	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81x	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82x	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83x	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84x	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85x	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86x	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87x	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88x	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89x	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8Ax	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8Bx	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8Cx	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8Dx	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8Ex	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8Fx	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90x	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91x	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92x	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93x	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94x	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95x	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96x	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97x	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98x	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99x	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9Ax	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9Bx	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9Cx	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9Dx	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9Ex	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9Fx	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0x	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1x	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2x	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3x	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4x	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5x	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6x	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7x	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8x	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9x	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AAx	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
ABx	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
ACx	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
ADx	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AEx	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AFx	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0x	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1x	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2x	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3x	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4x	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5x	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6x	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7x	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8x	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9x	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BAx	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BBx	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BCx	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BDx	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BEx	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BFx	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0x	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1x	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2x	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3x	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4x	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5x	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6x	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7x	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8x	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9x	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CAx	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CBx	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CCx	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CDx	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CEx	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CFx	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0x	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1x	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2x	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3x	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4x	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5x	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6x	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7x	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8x	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9x	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DAx	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DBx	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DCx	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DDx	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DEx	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DFx	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0x	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1x	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2x	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3x	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4x	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5x	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6x	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7x	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8x	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9x	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EAx	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EBx	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
ECx	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
EDx	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EEx	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EFx	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0x	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1x	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2x	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3x	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4x	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5x	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6x	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7x	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8x	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9x	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FAx	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FBx	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FCx	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FDx	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FEx	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FFx	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
RR	8 4 4 Operation Code R1 R2	R1,R2	All RR instructions except BCR,SPM, and SVC
	8 4 4 Operation Code M1 R2	M1,R2	BCR
	8 4 Operation Code R1	R1	SPM
	8 8 Operation Code I	I (See Notes 1,6,8, and 9)	SVC
RX	8 4 4 4 12 Operation Code R1 X2 B2 D2	R1,D2(X2,B2) R1,D2(,B2) R1,S2(X2) R1,S2	All RX instructions except BC
	8 4 4 4 12 Operation Code M1 X2 B2 D2	M1,D2(X2,B2) M1,D2(,B2) M1,S2(X2) M1,S2 (See Notes 1,6,8, and 9)	BC
RS	8 4 4 4 12 Operation Code R1 R3 B2 D2	R1,R3,D2(B2) R1,R3,S2	BXH,BXLE,LM,STM,LCL,STCL
	8 4 4 4 12 Operation Code R1 B2 D2	R1,D2(B2) R1,S2	All shift instructions
	8 4 4 4 12 Operation Code R1 M3 B2 D2	R1,M3,D2(B2) R1,M3,S2 (See Notes 1-3,7, 8,and 9)	ICM,STCM,CLM

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
SI	8 Operation Code | 8 I2 | 4 B1 | 12 D1	D1(B1,I2) S1,I2	All SI instructions except those listed for other SI formats
	8 Operation Code | | 4 B1 | 12 D1	D1(B1) S1	LPSW,SSM,TIO,TCH, TS
	16 Two-byte Operation Code | 4 B1 | 12 D1	D1(B1) S1 (See Notes 2,3,6, 7,8, and 10)	SCK,STCK,STIPD,SIOF, STIDC,SIO,HIO,HDV
SS	8 Operation Code | 4 L1 | 4 L2 | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L1,B1),D2(L2,B2) S1(L1),S2(L2)	PACK,UNPK,MVO,AP, CP,DP,MP,SP,ZAP
	8 Operation Code | 8 L | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L,B1),D2(B2) S1(L),S2	NC,OC,XC,CLC,MVC,MVN, MVZ,TR,TRT,ED,EDMK
	8 Operation Code | 4 L1 | 4 I3 | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L1,B1),D2(B2),I3 S1(L1),S2,I3 S1,S2,I3 (See Notes 2,3,5,6, 7 and 10)	SRP

Notes for Appendix C:

1. R1, R2, and R3 are absolute expressions that specify general or floating-point registers. The general register numbers are 0 through 15; floating-point register numbers are 0, 2, 4, and 6.

2. D1 and D2 are absolute expressions that specify displacements. A value of 0 - 4095 may be specified.

3. B1 and B2 are absolute expressions that specify base registers. Register numbers are 0 - 15.

4. X2 is an absolute expression that specifies an index register. Register numbers are 0 - 15.

5. L, L1, and L2 are absolute expressions that specify field lengths. An L expression can specify a value of 1 - 256. L1 and L2 expressions can specify a value of 1 - 16. In all cases, the assembled value will be one less than the specified value.

6. I, I2, and I3 are absolute expressions that provide immediate data. The value of I and I2 may be 0 - 255. The value of I3 may be 0 - 9.

7. S1 and S2 are absolute or relocatable expressions that specify an address.

8. RR, RS, and SI instruction fields that are blank under BASIC MACHINE FORMAT are not examined during instruction execution. The fields are not written in the symbolic operand, but are assembled as binary zeros.

9. M1 and M3 specify a 4-bit mask.

10. In IBM System/370 the SIO, HIO, HDV and SIOF operation codes occupy one byte and the low order bit of the second byte. In all other systems the HIO and SIO operation codes occupy only the first byte of the instruction.

This appendix contains a table of the mnemonic operation codes for all machine instructions that can be represented in assembler language, including extended mnemonic operation codes. It is in alphabetic order by instruction. Indicated for each instruction are both the mnemonic and machine operation codes, explicit and implicit operand formats, program interruptions possible, and condition code set.

The column headings in this appendix and the information each column provides follow.

Instruction: This column contains the name of the instruction associated with the mnemonic operation code.

Mnemonic Operation Code: This column gives the mnemonic operation code for the machine instruction. This is written in the operation field when coding the instruction.

Machine Operation Code: This column contains the hexadecimal equivalent of the actual machine operation code. The operation code will appear in this form in most storage dumps and when displayed on the system control panel. For extended mnemonics, this column also contains the mnemonic code of the instruction from which the extended mnemonic is derived.

Operand Format: This column shows the symbolic format of the operand field in both explicit and implicit form. For both forms, R1, R2, and R3 indicate general registers in operands one, two, and three respectively. X2 indicates a general register used as an index register in the second operand. Instructions which require an index register (X2) but are not to be indexed are shown with a 0 replacing X2. L, L1, and L2 indicate lengths for either operand, operand one, and operand two respectively. M1 and M3 indicate a four-bit mask in operand, one and three, respectively. I, I2, and I3 indicate immediate data eight bits long (I and I2) or four bits long (I3).

For the explicit format, D1 and D2 indicate a displacement and B1 and B2 indicate a base register for operands one and two.

For the implicit format, D1, B1 and D2, B2 are replaced by S1 and S2 which indicate a storage address in operands one and two.

Type of Instruction: This column gives the basic machine format of the instruction (RR, RX, SI, or SS). If an instruction is included in a special feature or is an extended mnemonic this is also indicated.

Program Interruptions Possible: This column indicates the possible program interruptions for this instruction. The abbreviations used are: A - Addressing, S - Specification, Ov - Overflow, P - Protection, Op - Operation (if feature is not installed), and Other - other interruptions which are listed. The type of overflow is indicated by: D - Decimal, E - Exponent, or F - Fixed Point.

Condition Code Set: The condition codes set as a result of this instruction are indicated in this column. (See legend following the table).

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
Add	A	5A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add	AR	1A	R1,R2	
Add Decimal	AP	FA	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Add Halfword	AH	4A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Logical	AL	5E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Logical	ALR	1E	R1, R2	
Add Normalized, Extended	AXR	36	R1,R2	
Add Normalized, Long	AD	6A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Normalized, Long	ADR	2A	R1, R2	
Add Normalized, Short	AE	7A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Normalized, Short	AER	3A	R1, R2	
Add Unnormalized, Long	AW	6E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Unnormalized, Long	AWR	2E	R1, R2	
Add Unnormalized, Short	AU	7E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Unnormalized, Short	AUR	3E	R1, R2	
And Logical	N	54	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
And Logical	NC	D4	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
And Logical	NR	14	R1, R2	
And Logical Immediate	NI	94	D1(B1), I2	S1, I2
Branch and Link	BAL	45	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Branch and Link	BALR	05	R1, R2	
Branch on Condition	BC	47	M1, D2(X2, B2) or M1, D2(, B2)	M1, S2(X2) or M1, S2
Branch on Condition	BCR	07	M1, R2	
Branch on Count	BCT	46	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Branch on Count	BCTR	06	R1, R2	
Branch on Equal	BE	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on High	BH	47(BC 2)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Index High	BXH	86	R1, R3, D2(B2)	R1, R3, S2
Branch on Index Low or Equal	BXLE	87	R1, R3, D2(B2)	R1, R3, S2
Branch on Low	BL	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Mixed	BM	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Minus	BM	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Equal	BNE	47(BC 7)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not High	BNH	47(BC 13)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Low	BNL	47(BC 11)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Minus	BNM	47(BC 11)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Ones	BNO	47(BC 14)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Plus	BNP	47(BC 13)	D2(X2, B2) or D2(, B2)	S2(X2 or S2
Branch on Not Zeros	BNZ	47(BC 7)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Ones	BO	47(BC 1)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Overflow	BO	47(BC 1)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Plus	BP	47(BC 2)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Zeros	BZ	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Zero	BZ	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch Unconditional	B	47(BC 15)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch Unconditional	BR	07(BCR 15)	R2	
Compare Algebraic	C	59	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2 or R1, S2
Compare Algebraic	CR	19	R1, R2	
Compare Decimal	CP	F9	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Compare Halfword	CH	49	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare Logical	CL	55	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare Logical	CLC	D5	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Compare Logical	CLR	15	R1, R2	
Compare Logical Characters under Mask	CLM	BD	R1, M3, D2(B2)	R1, M3, S2
Compare Logical Immediate	CLI	95	D1(B1), I2	S1, I2
Compare Logical Long	CLCL	0F	R1, R2	
Compare, Long	CD	69	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare, Long	CDR	29	R1, R2	
Compare, Short	CE	79	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare, Short	CER	39	R1, R2	
Convert to Binary	CVB	4F	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Convert to Decimal	CVD	4E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2

• Operand Format (Add)

Instruction	Type of Instruction	Program Interruption Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Add	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add	RR			F				Sum=0	Sum<0	Sum>0	Overflow
Add Decimal	SS, Decimal	x		D	x	x	Data	Sum=0	Sum<0	Sum>0	Overflow
Add Halfword	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add Logical	RX	x	x					Sum=0 H	Sum 0 H	Sum= 0 I	Sum 0 I
Add Logical	RR							Sum=0 H	Sum= 0 H	Sum= 0 I	Sum 0 I
Add Normalized, Extended	RR Floating Pt.		x	E		x	B,C	R	L	M	
Add Normalized, Long	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	
Add Normalized, Long	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Add Normalized, Short	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	
Add Normalized, Short	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Add Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	
Add Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	
Add Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	
Add Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	
Add Logical	RX	x	x					J	K		
And Logical	SS	x			x			J	K		
And Logical	RR							J	K		
And Logical Immediate	SI	x			x			J	K		
Branch and Link	RX							N	N	N	N
Branch and Link	RR							N	N	N	N
Branch on Condition	RX							N	N	N	N
Branch on Condition	RR							N	N	N	N
Branch on Count	RX							N	N	N	N
Branch on Count	RR							N	N	N	N
Branch on Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on High	RX, Ext. Mnemonic							N	N	N	N
Branch on Index High	RS							N	N	N	N
Branch on Index Low or Equal	RS							N	N	N	N
Branch on Low	RX, Ext. Mnemonic							N	N	N	N
Branch if Mixed	RX, Ext. Mnemonic							N	N	N	N
Branch on Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Not High	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Low	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Ones	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Plus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch if Ones	RX, Ext. Mnemonic							N	N	N	N
Branch on Overflow	RX, Ext. Mnemonic							N	N	N	N
Branch on Plus	RX, Ext. Mnemonic							N	N	N	N
Branch if Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch on Zero	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RR, Ext. Mnemonic							N	N	N	N
Compare Algebraic	RX	x	x					Z	AA	BB	
Compare Algebraic	RR							Z	AA	BB	
Compare Decimal	SS, Decimal	x				x	Data	Z	AA	BB	
Compare Halfword	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	SS	x	x					Z	AA	BB	
Compare Logical	RR	x						Z	AA	BB	
Compare Logical Characters under Mask	RS	x			x	x		XX	YY	ZZ	
Compare Logical Immediate	SI	x						Z	AA	BB	
Compare Logical Long	RR	x	x		x	x		Z	AA	BB	
Compare, Long	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Long	RR, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RR, Floating Pt.		x			x		Z	AA	BB	
Convert to Binary	RX	x	x				Data, F	N	N	N	N
Convert to Decimal	RX	x	x		x			N	N	N	N

• Condition Code Set (Add)

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
Divide	D	5D	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide	DR	1D	R1, R2	
Divide Decimal	DP	FD	D1, (L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Divide, Long	DD	6D	R1, D2(X2, B2), or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide, Long	DDR	2D	R1, R2	
Divide, Short	DE	7D	R1, D2(X2, B2)or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide, Short	DER	3D	R1, R2	
Edit	ED	DE	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Edit and Mark	EDMK	DF	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Exclusive Or	X	57	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Exclusive Or	XC	D7	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Exclusive Or	XR	17	R1, R2	
Exclusive Or Immediate	XI	97	D1(B1), I2	S1, I2
Execute	EX	44	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) R1, S2
Halve, Long	HDR	24	R1, R2	
Halve, Short	HER	34	R1, R2	
Halt Device	HDV	9E[1]	D1, B1	S1
Halt I/O	HIO	9E[1]	D1(B1)	
Insert Character	IC	43	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Insert Characters under Mask	ICM	BF	R1, M3, D2(B2)	R1, M3, S2
Insert Storage Key	ISK	09	R1, R2	
Load	L	58	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load	LR	18	R1, R2	
Load Address	LA	41	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load and Test	LTR	12	R1, R2	
Load and Test, Long	LTDR	22	R1, R2	
Load and Test, Short	LTER	32	R1, R2	
Load Complement	LCR	13	R1, R2	
Load Complement, Long	LCDR	23	R1, R2	
Load Complement, Short	LCER	33	R1, R2	
Load Control	LCTL	B7	R1, R3, D2(B2)	R1, R3, S2
Load Halfword	LH	48	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Long	LD	68	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Long	LDR	28	R1, R2	
Load Multiple	LM	98	R1, R3, D2(B2)	R1, R3, S2
Load Negative	LNR	11	R1, R2	
Load Negative, Long	LNDR	21	R1, R2	
Load Negative, Short	LNER	31	R1, R2	
Load Positive	LPR	10	R1, R2	
Load Positive, Long	LPDR	20	R1, R2	
Load Positive, Short	LPER	30	R1, R2	
Load PSW	LPSW	82	D1(B1)	S1
Load Rounded, Extended to Long	LRDR	25	R1, R2	
Load Rounded, Long to Short	LRER	35	R1, R2	
Load, Short	LE	78	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Short	LER	38	R1, R2	
Move Characters	MVC	D2	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Move Immediate	MVI	92	D1(B1), I2	S1, I2
Move Long	MVCL	0E	R1, R2	
Move Numerics	MVN	D1	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Move with Offset	MVO	F1	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Move Zones	MVZ	D3	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Multiply	M	5C	R1, D2(X2, B2)or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply	MR	1C	R1, R2	
Multiply Decimal	MP	FC	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Multiply Extended	MXR	26	R1, R2	
Mulitply Halfword	MH	4C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Long	MD	6C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Long	MDR	2C	R1, R2	
Multiply, Long to Extended	MXD	67	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1(S2)
Multiply, Long to Extended	MXDR	27	R1, R2	
Multiply, Short	ME	7C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Short	MER	3C	R1, R2	
No Operation	NOP	47(BC 0)	D2(X2, B2) or D2(, B2)	S2(X2) or S2

• Operand Format (Divide)

Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Divide	RX	x	x				F	N	N	N	N
Divide	RR		x				F	N	N	N	N
Divide Decimal	SS, Decimal	x	x		x	x	D, Data	N	N	N	N
Divide, Long	RX,Floating Pt.	x	x	E		x	B, E	N	N	N	N
Divide, Long	RR, Floating Pt.		x	E		x	B, E	N	N	N	N
Divide, Short	RX, Floating Pt.	x	x	E		x	B, E	N	N	N	N
Divide, Short	RR, Floating Pt.		x	E		x	B, E	N	N	N	N
Edit	SS, Decimal	x			x	x	Data	S	T	U	
Edit and Mark	SS, Decimal	x			x	x	Data	S	T	U	
Exclusive Or	RX	x	x					J	K		
Exclusive Or	SS	x			x			J	K		
Exclusive Or	RR							J	K		
Exclusive Or Immediate	SI	x			x			J	K		
Execute	RX	x	x				G	(May be set by this instruction)			
Halve, Long	RR, Floating Pt.		x			x		N	N	N	N
Halve, Short	RR, Floating Pt.		x			x		N	N	N	N
Halt Device	SI						A	AAM	CC	AAL	
Halt I/O	SI						A	DD	CC	GG	KK
Insert Character	RX	x						N	N	N	N
Insert Characters under Mask	RS	x			x	x		UU	TT	SS	
Insert Storage Key	RR	x	x			x	A	N	N	N	N
Load	RX	x	x					N	N	N	N
Load	RR							N	N	N	N
Load Address	RX							N	N	N	N
Load and Test	RR							J	L	M	
Load and Test, Long	RR, Floating Pt.		x			x		R	L	M	
Load and Test, Short	RR, Floating Pt.		x			x		R	L	M	
Load Complement	RR			F				P	L	M	O
Load Complement, Long	RR, Floating Pt.		x			x		R	L	M	
Load Complement, Short	RR, Floating Pt.		x			x		R	L	M	
Load Control	RS	x	x		x	x	A	N	N	N	N
Load Halfword	RX	x	x					N	N	N	N
Load, Long	RX, Floating Pt.	x	x			x		N	N	N	N
Load, Long	RR, Floating Pt.		x			x		N	N	N	N
Load Multiple	RS	x	x					N	N	N	N
Load Negative	RR							J	L		
Load Negative, Long	RR, Floating Pt.		x			x		R	L		
Load Negative, Short	RR, Floating Pt.		x			x		R	L		
Load Positive	RR			F				J		M	O
Load Positive, Long	RR, Floating Pt.		x			x		R	L	M	
Load Positive, Short	RR, Floating Pt.		x			x		R	L	M	
Load PSW	SI	x	x				A	QQ	QQ	QQ	QQ
Load Rounded, Extended to Long	RR, Floating Pt.		x	E		x		N	N	N	N
Load Rounded, Long to Short	RR, Floating Pt.		x	E		x		N	N	N	N
Load, Short	RX, Floating Pt.	x	x			x		N	N	N	N
Load, Short	RR, Floating Pt.		x			x		N	N	N	N
Move Characters	SS	x			x			N	N	N	N
Move Immediate	SI	x			x			N	N	N	N
Move Long	RR	x	x		x	x		AAA	AAB	AAC	AAD
Move Numerics	SS	x			x			N	N	N	N
Move with Offset	SS	x			x			N	N	N	N
Move Zones	SS	x			x			N	N	N	N
Multiply	RX	x	x					N	N	N	N
Multiply	RR		x					N	N	N	N
Multiply Decimal	SS, Decimal	x	x		x	x	Data	N	N	N	N
Multiply Halfword	RX	x	x					N	N	N	N
Multiply, Extended	RR, Floating Pt.		x	E		x	B	N	N	N	N
Multiply, Long/Extended	RX, Floating Pt.	x	x	E	x	x	B	N	N	N	N
Multiply, Long/Extended	RR, Floating Pt.		x	E		x	B	N	N	N	N
Multiply, Long	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
Multiply, Long	RR, Floating Pt.		x	E		x	B	N	N	N	N
Multiply, Short	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
Multiply, Short	RR, Floating Pt.		x	E		x	B	N	N	N	N
No Operation	RX, Ext.Mnemonic							N	N	N	N

Condition Code Set (Divide)

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
No Operation	NOPR	07(BCR 0)	R2	
Or Logical	O	56	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Or Logical	OC	D6	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Or Logical	OR	16	R1, R2	
Or Logical Immediate	OI	96	D1(B1), I2	S1, I2
Pack	PACK	F2	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Read Direct	RDD	85	D1(B1), I2	S1, I2
Set Clock	SCK	B204	D1(B1)	S1
Set Program Mask	SPM	04	R1	
Set Storage Key	SSK	08	R1, R2	
Set System Mask	SSM	80	D1(B1)	S1
Shift and Round Decimal	SRP	F0	D1(L1, B1), D2(B2), I3	S1(L1), S2, I3 or S1, S2, I3
Shift Left Double Algebraic	SLDA	8F	R1, D2(B2)	R1, S2
Shift Left Double Logical	SLDL	8D	R1, D2(B2)	R1, S2
Shift Left Single Algebraic	SLA	8B	R1, D2(B2)	R1, S2
Shift Left Single Logical	SLL	89	R1, D2(B2)	R1, S2
Shift Right Double Algebraic	SRDA	8E	R1, D2(B2)	R1, S2
Shift Right Double Logical	SRDL	8C	R1, D2(B2)	R1, S2
Shift Right Single Algebraic	SRA	8A	R1, D2(B2)	R1, S2
Shift Right Single Logical	SRL	88	R1, D2(B2)	R1, S2
Start I/O	SIO	9C	D1(B1)	S1
Start I/O Fast Release	SIOF	9C	D1(B1)	S1
Store	ST	50	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Store Character	STC	42	R1, D2(X2, B2) or R1, D2(, B2	R1, D2(X2) or R1, S2
Store Channel ID	STIDC	B203	D1(B1)	S1
Store Halfword	STH	40	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Store Long	STD	60	R1, D2(X2, B2)	R1, S2(X2) or R1, S2
Store Multiple	STM	90	R1, R2, D2(B2)	R1, R2, S2
Store Short	STE	70	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Store Characters under Mask	STCM	BE	R1, M3, D2(B2)	R1, M3, S2
Store Clock	STCK	B205	D1(B1)	S1
Store Control	STCTL	B6	R1, R3, D2(B2)	R1, R3, S2
Store CPU ID	STIDP	B202	D1(B1)	S1
Subtract Normalized, Extended	SXR	37	R1, R2	
Subtract	S	5B	R1, D2(X2	R1, S2(X2) or R1, S2
Subtract	SR	1B	R1, R2	
Subtract Decimal	SP	FB	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Subtract Halfword	SH	4B	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Logical	SL	5F	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Logical	SLR	1F	R1, R2	
Subtract Normalized, Long	SD	6B	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Normalized, Long	SDR	2B	R1, R2	
Subtract Normalized, Short	SE	7B	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Normalized, Short	SER	3B	R1, R2	
Subtract Unnormalized, Long	SW	6F	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Unnormalized, Long	SWR	2F	R1, R2	
Subtract Unnormalized, Short	SU	7F	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Subtract Unnormalized, Short	SUR	3F	R1, R2	
Supervisor Call	SVC	0A	I	
Test and Set	TS	93	D1(B1)	S1
Test Channel	TCH	9F	D1(B1)	S1
Test I/O	TIO	9D	D1(B1)	S1
Test Under Mask	TM	91	D1(B1), I2	S1, I2
Translate	TR	DC	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Translate and Test	TRT	DD	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Unpack	UNPK	F3	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Write Direct	WRD	84	D1(B1), I2	S1, I2
Zero and Add Decimal	ZAP	F8	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2

- Operation Format (No Operation)

Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
No Operation	RR, Ext. Mnemonic							N	N	N	N
Or Logical	RX	x	x					J	K		
Or Logical	SS	x			x			J	K		
Or Logical	RR							J	K		
Or Logical Immediate	SI	x			x			J	K		
Pack	SS	x			x			N	N	N	N
Read Direct	SI	x			x	x	A	N	N	N	N
Set Clock	SI	x	x		x	x	A	AAE	AAF		AAG
Set Program Mask	RR							RR	RR	RR	RR
Set Storage Key	RR	x	x			x	A	N	N	N	N
Set System Mask	SI	x					A	N	N	N	N
Shift Left Double Algebraic	RS		x	F				J	L	M	O
Shift and Round Decimal	SS	x		D	x	x	Data	J	L	M	O
Shift Left Double Logical	RS		x					N	N	N	N
Shift Left Single Algebraic	RS			F				J	L	M	O
Shift Left Single Logical	RS							N	N	N	N
Shift Right Double Algebraic	RS		x					J	L	M	
Shift Right Double Logical	RS		x					N	N	N	N
Shift Right Single Algebraic	RS							J	L	M	
Shift Right Single Logical	RS							N	N	N	N
Start I/O	SI						A	MM	CC	EE	AA
Start I/O Fast Release	SI						A	MM	CC	EE	KK
Store	RX	x	x		x			N	N	N	N
Store Character	RX	x			x			N	N	N	N
Store Halfword	RX	x	x		x			N	N	N	N
Store Long	RX, Floating Pt.	x	x		x	x		N	N	N	N
Store Multiple	RS	x	x		x			N	N	N	N
Store Short	RX, Floating Pt.	x	x		x	x		N	N	N	N
Store Channel ID	SI					x	A	AAH	CC	AAI	KK
Store Characters under Mask	RS	x			x	x		N	N	N	N
Store Clock	SI	x			x	x		AAJ	AAK	AAN	AAG
Store Control	RS	x	x		x	x	A	N	N	N	N
Store CPU ID	SI	x	x		x	x	A	N	N	N	N
Subtract	RX	x	x	F				V	X	Y	O
Subtract	RR			F				V	X	Y	O
Subtract Decimal	SS, Decimal	x		D	x	x	Data	V	X	Y	O
Subtract Halfword	RX	x	x	F				V	X	Y	O
Subtract Logical	RX	x	x						W,H	V,I	W,I
Subtract Logical	RR								W,H	V,I	W,I
Subtract Normalized, Long	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Long	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Normalized, Extended	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Subtract Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Supervisor Call	RR							N	N	N	N
Test and Set	SI	x			x			SS	TT		
Test Channel	SI						A	JJ	II	FF	HH
Test I/O	SI						A	LL	CC	EE	KK
Test Under Mask	SI	x						UU	VV		WW
Translate	SS	x			x			N	N	N	N
Translate and Test	SS	x						PP	NN	OO	
Unpack	SS	x			x			N	N	N	N
Write Direct	SI	x				x	A	N	N	N	N
Zero and Add Decimal	SS, Decimal	x		D	x	x	Data	J	L	M	O

Condition Code Set (No Operation)

Program Interruptions Possible

Under Ov: D = Decimal
E = Exponent
F = Fixed Point

Under Other:

A	Privileged Operation
B	Exponent Underflow
C	Significance
D	Decimal Divide
E	Floating Point Divide
F	Fixed Point Divide
G	Execute
GA	Monitoring

Condition Code Set

H	No Carry
I	Carry
J	Result = 0
K	Result is Not Equal to Zero
L	Result is Less Than Zero
M	Result is Greater Than Zero
N	Not Changed
O	Overflow
P	Result Exponent Underflows
Q	Result Exponent Overflows
R	Result Fraction = 0
S	Result Field Equals Zero
T	Result Field is Less Than Zero
U	Result Field is Greater Than Zero
V	Difference = 0
W	Difference is Not Equal to Zero
X	Difference is Less Than Zero
Y	Difference is Greater Than Zero
Z	First Operand Equals Second Operand
AA	First Operand is Less Than Second Operand
BB	First Operand is Greater Than Second Operand
CC	CSW Stored
DD	Channel and Subchannel not Working
EE	Channel or Subchannel Busy
FF	Channel Operating in Burst Mode
GG	Burst Operation Terminated
HH	Channel Not Operational
II	Interruption Pending in Channel
JJ	Channel Available
KK	Not Operational
LL	Available
MM	I/O Operation Initiated and Channel Proceeding With its Execution
NN	Nonzero Function Byte Found Before the First Operand Field is Exhausted
OO	Last Function Byte is Nonzero
PP	All Function Bytes Are Zero
QQ	Set According to Bits 34 and 35 of the New PSW Loaded
RR	Set According to Bits 2 and 3 of the Register Specified by R1
SS	Leftmost Bit of Byte Specified = 0
TT	Leftmost Bit of Byte Specified = 1
UU	Selected Bits Are All Zeros; Mask is All Zeros
VV	Selected Bits Are Mixed (zeros and ones)
WW	Selected Bits Are All Ones
XX	Selected bytes are equal, or mask is zero
YY	Selected field of first operand is low
ZZ	Selected field of first operand is high
AAA	First-operand and second-operand counts are equal
AAB	First operand count is lower
AAC	First operand count is higher
AAD	No movement because of destructive overlap
AAE	Clock value set
AAF	Clock value secure
AAG	Clock not operational
AAH	Channel ID correctly stored
AAI	Channel activity prohibited during ID
AAJ	Clock value is valid
AAK	Clock value not necessarily valid
AAL	Channel working with another device
AAM	Subchannel busy or interruption pending
AAN	Clock in error state

• Program Interruptions Possible

RR Format

Operation Code	Name	Mnemonic	Remarks
00			
01			
02			
03			
04	Set Program Mask	SPM	
05	Branch and Link	BALR	
06	Branch on Count	BCTR	
07	Branch on Condition	BCR	
08	Set Storage Key	SSK	
09	Insert Storage Key	ISK	
0A	Supervisor Call	SVC	
0B			
0C			
0D			
0E	Move Long	MVCL	S/370 only
0F	Compare Logical Long	CLCL	S/370 only
10	Load Positive	LPR	
11	Load Negative	LNR	
12	Load and Test	LTR	
13	Load Complement	LCR	
14	AND	NR	
15	Compare Logical	CLR	
16	OR	OR	
17	Exclusive OR	XR	
18	Load	LR	
19	Compare	CR	
1A	Add	AR	
1B	Subtract	SR	
1C	Multiply	MR	
1D	Divide	DR	
1E	Add Logical	ALR	
1F	Subtract Logical	SLR	
20	Load Positive (Long)	LPDR	
21	Load Negative (Long)	LNDR	
22	Load and Test (Long)	LTDR	
23	Load Complement (Long)	LCDR	
24	Halve (Long)	HDR	
25	Load Rounded (Extended to Long)	LRDR	85,195,S/370 only
26	Multiply (Extended)	MXR	85,195,S/370 only
27	Multiply (Long to Extended)	MXDR	85,195,S/370 only
28	Load (Long)	LDR	
29	Compare (Long)	CDR	
2A	Add Normalized (Long)	ADR	
2B	Subtract Normalized	SDR	
2C	Multiply (Long)	MDR	
2D	Divide (Long)	DDR	
2E	Add Unnormalized (Long)	AWR	
2F	Subtract Unnormalized (Long)	SWR	
30	Load Positive (Short)	LPER	
31	Load Negative (Short)	LNER	
32	Load and Test (Short)	LTER	
33	Load Complement (Short)	LCER	
34	Halve (Short)	HER	
35	Load Rounded (Long or Short)	LRER	85,195,S/370 only
36	Add Normalized (Extended)	AXR	85,195,S/370 only
37	Subtract Normalized (Extended)	SXR	85,195,S/370 only
38	Load (Short)	LER	

RR Format			
Operation Code	Name	Mnemonic	Remarks
39	Compare (Short)	CER	
3A	Add Normalized (Short)	AER	
3B	Subtract Normalized (Short)	SER	
3C	Multiply (Short)	MER	
3D	Divide (Short)	DER	
3E	Add Unnormalized (Short)	AUR	
3F	Subtract Unnormalized (Short)	SUR	
RX Format			
40	Store Halfword	STH	
41	Load Address	LA	
42	Store Character	STC	
43	Insert Character	IC	
44	Execute	EX	
45	Branch and Link	BAL	
46	Branch on Count	BCT	
47	Branch on Condition	BC	
48	Load Halfword	LH	
49	Compare Halfword	CH	
4A	Add Halfword	AH	
4B	Subtract Halfword	SH	
4C	Multiply Halfword	MH	
4D			
4E	Convert to Decimal	CVD	
4F	Convert to Binary	CVB	
50	Store	ST	
51			
52			
53			
54	AND	N	
55	Compare Logical	CL	
56	OR	O	
57	Exclusive OR	X	
58	Load	L	
59	Compare	C	
5A	Add	A	
5B	Subtract	S	
5C	Multiply	M	
5D	Divide	D	
5E	Add Logical	AL	
5F	Subtract Logical	SL	
60	Store (Long)	STD	
61			
62			
63			
64			
65			
66			
67	Multiply (Long to Extended)	MXD	85,195,S/370 only
68	Load (Long)	LD	
69	Compare (Long)	CD	
6A	Add Normalized (Long)	AD	
6B	Subtract Normalized (Long)	SD	
6C	Multiply (Long)	MD	
6D	Divide (Long)	DD	
6E	Add Unnormalized (Long)	AW	
6F	Subtract Unnormalized (Long)	SW	

Operation Code	Name	Mnemonic	Remarks
RX Format			
70	Store (Short)	STE	
71			
72			
73			
74			
75			
76			
77			
78	Load (Short)	LE	
79	Compare (Short)	CE	
7A	Add Normalized (Short)	AE	
7B	Subtract Normalized (Short)	SE	
7C	Multiply (Short)	ME	
7D	Divide (Short)	DE	
7E	Add Unnormalized (Short)	AU	
7F	Subtract Unnormalized (Short)	SU	
RS,SI Format			
80	Set System Mask	SSM	
81			
82	Load PSW	LPSW	
83	Diagnose		
84	Write Direct	WRD	
85	Read Direct	RDD	
86	Branch on Index High	BXH	
87	Branch on Index Low or Equal	BXLE	
88	Shift Right Single Logical	SRL	
89	Shift Left Single Logical	SLL	
8A	Shift Right Single	SRA	
8B	Shift Left Single	SLA	
8C	Shift Right Double Logical	SRDL	
8D	Shift Left Double Logical	SLDL	
8E	Shift Right Double	SRDA	
8F	Shift Left Double	SLDA	
90	Store Multiple	STM	
91	Test under Mask	TM	
92	Move (Immediate)	MVI	
93	Test and Set	TS	
94	AND (Immediate)	NI	
95	Compare Logical (Immediate)	CLI	
96	OR (Immediate)	OI	
97	Exclusive OR (Immediate)	XI	
98	Load Multiple	LM	
99			
9A			
9B			
9C	Start I/O, Start I/O Fast Release	SIO,SIOF	See Note 2
9D	Test I/O	TIO	
9E	Halt I/O, Halt Device	HIO,HDV	See Note 1
9F	Test Channel	TCH	
A0			
A1			
A2			
A3			
A4			
A5			
A6			

RS,SI Format

Operation Code	Name	Mnemonic	Remarks
A7			
A8			
A9			
AA			
AB			
AC			
AD			
AE			
AF			
B0			
B1			
B2	(First byte of two-byte operation codes)		See Note 3
B3			
B4			
B5			
B6	Store Control	STCTL	S/370 only
B7	Load Control	LCTL	S/370 only
B8			
B9			
BA			
BB			
BC			
BD	Compare Logical Characters under Mask	CLM	S/370 only
BE	Store Characters under Mask	STCM	S/370 only
BF	Insert Characters under Mask	ICM	S/370 only

SS Format

Operation Code	Name	Mnemonic	Remarks
C0			
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
CA			
CB			
CC			
CD			
CE			
CF			
D0			
D1	Move Numerics	MVN	
D2	Move (Characters)	MVC	
D3	Move Zones	MVZ	
D4	AND (Characters)	NC	
D5	Compare Logical (Characters)	CLC	
D6	OR (Characters)	OC	
D7	Exclusive OR (Characters)	XC	
D8			
D9			
DA			
DB			
DC	Translate	TR	

SS Format			
Operation Code	Name	Mnemonic	Remarks
DD	Translate and Test	TRT	
DE	Edit	ED	
DF	Edit and Mark	EDMK	
E0			
E1			
E2			
E3			
E4			
E5			
E6			
E7			
W8			
E9			
EA			
EB			
EC			
ED			
EE			
EF			
F0	Shift and Round Decimal	SRP	S/370 only
F1	Move with Offset	MVO	
F2	Pack	PACK	
F3	Unpack	UNPK	
F4			
F5			
F6			
F7			
F8	Zero and Add Decimal	ZAP	
F9	Compare Decimal	CP	
FA	Add Decimal	AP	
FB	Subtract Decimal	SP	
FC	Multiply Decimal	MP	
FD	Divide Decimal	DP	
FE			
FF			

Note 1: Under the System/370 architecture the machine operations for Halt Device and Halt I/O are as follows:

1001 1110 XXXX XXX0	Halt I/O	HIO
1001 1110 XXXX XXX1	Halt Device	HDV

(X denotes an ignored bit position)

Under System/360 the Halt I/O operation code is:

1001 1110 XXXX XXXX

The Halt Device instruction does not exist under this system; the second byte is completely ignored.

Note 2: Under the System/370 architecture the machine operations for Start I/O and Start I/O Fast Release are as follows:

1001 1100 XXXX XXX0	Start I/O	SIO	SIO
1001 1100 XXXX XXX1	Start I/O Fast Release		SIOF

(X denotes an ignored bit position)

Under System/360 the Start I/O code is:

1001 1100 XXXX XXXX

The Start I/O Fast Release instruction does not exist under this system. The second byte is completely ignored.

Note 3: The following operation codes occupy two bytes of SI-type instructions. They can be used on System/370 machines only.

Operation Code	Name	Mnemonic
B202	Store CPU ID	STIDP
B203	Store Channel ID	STIDC
B204	Set Clock	SCK
B205	Store Clock	STCK

The special Model 85, Model 195, and System/370 instructions are supported only by the F Assembler.

Operation	Name Entry	Operand Entry
ACTR	Must not be present	An arithmetic SETA expression
AGO	A sequence symbol or not present	A sequence symbol
AIF	A sequence symbol or not present	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP	A sequence symbol	Must not be present
CCW	Any symbol or not present	Four operands, separated by commas
CNOP	A sequence symbol or not present	Two absolute expressions, separated by a comma
COM	A sequence symbol or not present	Must not be present
COPY	Must not be present	A symbol
CSECT	Any symbol or not present	Must not be present
CXD *	Any symbol or not present	Must not be present
DC	Any symbol or not present	One or more operands, separated by commas
DROP	A sequence symbol or not present	One to sixteen absolute expressions, separated by commas
DS	Any symbol or not present	One or more operands, separated by commas
DSECT	A variable symbol or an ordinary symbol	Must not be present
DXD *	A symbol	One or more operands, separated by commas
EJECT	A sequence symbol or not present	Must not be present
END	A sequence symbol or not present	A relocatable expression or not present
ENTRY	A sequence symbol or not present	One or more relocatable symbols, separated by commas
EQU	A variable symbol or an ordinary symbol	An absolute or relocatable expression
EXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas
GBLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
ICTL	Must not be present	One to three decimal values, separated by commas

* Assembler F only

Operation Entry	Name Entry	Operand Entry
ISEQ	Must not be present	Two decimal values, separated by a comma
LCLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLC	Must not be present	One or more variable symbols separated by commas[2]
LTORG	Any symbol or not present	Must not be present
MACRO[1]	Must not be present	Must not be present
MEND[1]	A sequence symbol or not present	Must not be present
MEXIT[1]	A sequence symbol or not present	Must not be present
MNOTE[1]	A sequence symbol, a variable symbol or not present	A severity code, followed by a comma, followed by any combination of characters enclosed in apostrophes
OPSYN*	An ordinary symbol	A machine instruction mnemonic code, an extended mnemonic code, or an operation code defined by a previous OPSYN instruction
	A machine or extended mnemonic operation code	Blank
ORG	A sequence symbol or not present	A relocatable expression or not present
PRINT	A sequence symbol or not present	One to three operands
PUNCH	A sequence symbol or not present	One to eighty characters enclosed in apostrophes
REPRO	A sequence symbol or not present	Must not be present
SETA	A SETA symbol	An arithmetic expression
SETB	A SETB symbol	A 0 or a 1, or logical expression enclosed in parentheses
SETC	A SETC symbol	A type attribute, a character expression, a substring notation, or a concatenation of character expressions and substring notations
SPACE	A sequence symbol or not present	A decimal self-defining term or not present
START	Any symbol or not present	A self-defining term or not present
TITLE[3]	A special symbol (0 to 4 characters), a sequence symbol, a variable symbol, or not present	One to 100 characters, enclosed in apostrophes
USING	A sequence symbol or not present	An absolute or relocatable expression followed by 1 to 16 absolute expressions, separated by commas
WXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas

[1]May only be used as part of a macro-definition.

[2]SET symbols may be defined as subscripted SET symbols.

[3]See Section 5 for the description of the name entry.

*Assembler F only.

INSTRUCTION	NAME ENTRY	OPERAND ENTRY
Model Statements[3] [4]	An ordinary symbol, variable symbol, sequence variable symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present	Any combination of characters (including variable symbols)
Prototype Statement[1]	A symbolic parameter or not present	Zero or more operands that are symbolic parameters, separated by commas, followed by zero or more operands (separated by commas) of the form symbolic parameter, equal sign, optional standard value
Macro-Instruction Statement[1]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol,[2] or not present	Zero or more positional operands separated by commas, followed by zero or more keyword operands (separated by commas) of the form keyword, equal sign, value[2]
Assembler Language Statement[3] [4]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present	Any combination of characters (including variable symbols)

[1] May only be used as part of a macro definition.
[2] Variable symbols appearing in a macro instruction are replaced by their values before the macro instruction is processed.
[3] Variable symbols may be used to generate assembler language mnemonic operation codes as listed in Section 5, except ACTR, COPY, END, ICTL, CSECT, DSECT, ISEQ, PRINT, REPRO, and START. Variable symbols may not be used in the name and operand entries of the following instructions: COPY, END, ICTL, and ISEQ. Variable symbols may not be used in the name entry of the ACTR instruction.
[4] No substitution for variables in the line following a REPRO statement is performed.

TYPE	IMPLIED LENGTH (BYTES)	ALIGN-MENT	LENGTH MODI-FIER RANGE	SPECIFIED BY	NUMBER OF CON-STANTS PER OPERAND	RANGE FOR EX-PONENTS	RANGE FOR SCALE	TRUN-CATION/ PADDING SIDE
C	as needed	byte	.1 to 256 (1)	characters	one			right
X	as needed	byte	.1 to 256 (1)	hexadecimal digits	one			left
B	as needed	byte	.1 to 256	binary digits	one			left
F	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 to +346	left (4)
H	2	half word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 +346	left (4)
E	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right (4)
D	8	double word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right (4)
L (3)	16	double word	.1 to 16	decimal digits	multi-ple	-85 to +75	0-28	right (4)
P	as needed	byte	.1 to 16	decimal digits	multi-ple			left
Z	as needed	byte	.1 to 16	decimal digits	multi-ple			left
A	4	word	.1 to 4 (2)	any expression	multi-ple			left
Q (3)	4	word	1-4	symbol nam-ing a DXD or DSECT	multi-ple			left
V	4	word	3 or 4	relocatable symbol	multi-ple			left
S	2	half word	2 only	one absolute or relocatab-le expression or two absol-ute express-ions: exp (exp)	multi-ple			
Y	2	half word	.1 to 2 (2)	any expression	multi-ple			left

(1) In a DS assembler instruction C and X type constants may have length specification to 65535.
(2) Bit length specification permitted with absolute expressions only. Relocatable A-type constants, 3 or 4 bytes only; relocatable Y-type constants, 2 bytes only.
(3) Assembler F only.
(4) Errors will be flagged if significant bits are truncated or if the value specified cannot be contained in the implied length of the constant.

The four charts in this appendix summarize the macro language described in Part II of this publication.

Chart 1 indicates which macro language elements may be used in the name and operand entries of each statement.

Chart 2 is a summary of the expressions that may be used in macro-instruction statements.

Chart 3 is a summary of the attributes that may be used in each expression.

Chart 4 is a summary of the variable symbols that may be used in each expression.

	Variable Symbols										Attributes						
		Global SET Symbols			Local SET Symbols			System Variable Symbols									
Statement	Symbolic Parameter	SETA	SETB	SETC	SETA	SETB	SETC	&SYSNDX	&SYSECT	&SYSLIST	Type	Length	Scaling	Integer	Count	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
SETA	Operand[2]	Name Operand	Operand[3]	Operand[9]	Name Operand	Operand[3]	Operand[9]	Operand		Operand[2]		Operand	Operand	Operand	Operand	Operand	
SETB	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	
SETC	Operand	Operand[7]	Operand[8]	Name Operand	Operand[7]	Operand[8]	Name Operand	Operand	Operand	Operand	Operand						
AIF	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Name Operand
AGO																	Name Operand
ACTR	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[3]	Operand[2]	Operand		Operand[2]		Operand	Operand	Operand	Operand	Operand	
ANOP																	Name
MEXIT																	Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand							Name
MEND																	Name
Outer Macro		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand										Name
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	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.
9. Only if one to eight decimal digits.

Chart 1. Macro Language Elements

Chart 2. Conditional Assembly 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 the corresponding operand is a self-defining term 6. &SYSLIST(n) if the corresponding operand is a self-defining term 7. &SYSLIST(n,m) if the corresponding operand is a self-defining term 8. &SYSNDX	1. Any combination of characters enclosed in apostrophes 2. Any variable symbol enclosed in apostrophes 3. A concatenation of variable symbols and other characters enclosed in apostrophes 4. A request for a type attribute	1. SETB symbols 2. Arithmetic relations[1] 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 through 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 the operators GT, LT, EQ, NE, GE, or LE.

[2] A character relation consists of two character expressions related by the operator GT, LT, EQ, NE, GE, or LE. The type attribute notation and the substring notation may also be used in character relations. The maximum size of the character expressions that can be compared is 255 characters. If the two character expressions are of unequal size, then the smaller one will always compare less than the larger.

[3] Maximum of eight characters will be assigned.

Chart 3. Attributes

Attribute	Notation	May be used with:	May be used only if type attribute is:	May be used in *
Type	T'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m) inside macro definitions	(May always be used)	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 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,L,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,L,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	Arithmetic expressions
Number	N'	Symbolic parameters, &SYSLIST, and &SYSLIST(n) inside macro definitions	Any letter	Arithmetic expressions

* NOTE: There are definite restrictions in the use of these attributes. Refer to text, Section 9.

Chart 4. 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	LCLA or GBLA instruction	0	SETA instruction	1. Arithmetic expressions 2. Character expressions
SETB	LCLB or GBLB instruction	0	SETB instruction	1. Arithmetic expressions 2. Character expressions 3. Logical expressions
SETC	LCLC or GBLC instruction	Null character value	SETC instruction	1. Arithmetic expressions if value is self-defining term 2. Character expressions
&SYSNDX[1]	The assembler	Macro instruction index	(Constant throughout definition; unique for each macro-instruction)	1. Arithmetic expressions 2. Character expressions
&SYSECT[1]	The assembler	Control section in which macro instruction appears	(Constant throughout definition; set by CSECT, DSECT, and START)	Character expressions
&SYSLIST[1]	The assembler	Not applicable	Not applicable	N'&SYSLIST in arithmetic expressions
&SYSLIST(n)[1] &SYSLIST(n,m)[1]	The assembler	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions

[1] May only be used in macro definitions.

Given:

1. A TABLE with 15 entries, each 16 bytes long, having the following 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 the following 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.

```
         TITLE
         PRINT DATA
*
*        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' SAMPL044
         MEXIT                                                           SAMPL045
.ERROR2  MNOTE 1,'OPERAND TYPES DIFFERENT, NO STATEMENTS GENERATED'      SAMPL046
         MEXIT                                                           SAMPL047
.ERROR3  MNOTE 1,'IMPROPER OPERAND TYPES, NO STATEMENTS GENERATED'       SAMPL048
         MEXIT                                                           SAMPL049
.ERROR4  MNOTE 1,'IMPROPER OPERAND LENGTHS, NO STATEMENTS GENERATED'     SAMPL050
         MEND                                                            SAMPL051
*                                                                        SAMPL052
*        MAIN ROUTINE                                                    SAMPL053
*                                                                        SAMPL054
SAMPLR   CSECT                                                           SAMPL055
BEGIN    SAVE  (14,12),,*                                                SAMPL056
         BALR  R12,0             ESTABLISH ADDRESSABILITY OF PROGRAM     SAMPL057
         USING *,R12             AND TELL THE ASSEMBLER WHAT BASE TO USE SAMPL058
         ST    13,SAVE13                                                 SAMPL059
         LM    R5,R7,=A(LISTAREA,16,LISTEND)  LOAD LIST AREA PARAMETERS SAMPL060
         USING LIST,R5         REGISTER 5 POINTS TO THE LIST             SAMPL061
MORE     BAL   R14,SEARCH      FIND LIST ENTRY IN TABLE                  SAMPL062
         TM    SWITCH,NONE     CHECK TO SEE IF NAME WAS FOUND            SAMPL063
         BO    NOTTHERE        BRANCH IF NOT                             SAMPL064
         USING TABLE,R1        REGISTER 1 NOW POINTS TO TABLE ENTRY      SAMPL065
         MOVE  TSWITCH,LSWITCH             MOVE FUNCTIONS                SAMPL066
         MOVE  TNUMBER,LNUMBER                     FROM LIST ENTRY       SAMPL067
         MOVE  TADDRESS,LADDRESS                       TO TABLE ENTRY    SAMPL068
LISTLOOP BXLE  R5,R6,MORE      LOOP THROUGH THE LIST                     SAMPL069
         CLC   TESTTABL(240),TABLAREA                                    SAMPL070
         BNE   NOTRIGHT                                                  SAMPL071
         CLC   TESTLIST(96),LISTAREA                                     SAMPL072
         BNE   NOTRIGHT                                                  SAMPL073
         WTO   'ASSEMBLER SAMPLE PROGRAM SUCCESSFUL'                     SAMPL074
EXIT     L     R13,SAVE13                                                SAMPL075
         RETURN (14,12),RC=0                                             SAMPL076
*                                                                        SAMPL077
NOTRIGHT WTO   'ASSEMBLER SAMPLE PROGRAM UNSUCCESSFUL'                   SAMPL078
         B     EXIT                                                      SAMPL079
NOTTHERE OI    LSWITCH,NONE TURN ON SWITCH IN LIST ENTRY                 SAMPL080
         B     LISTLOOP      GO BACK AND LOOP                            SAMPL081
SAVE13   DC    F'0'                                                      SAMPL082
SWITCH   DC    X'00'                                                     SAMPL083
NONE     EQU   X'80'                                                     SAMPL084
*                                                                        SAMPL085
*        BINARY SEARCH ROUTINE                                           SAMPL086
*                                                                        SAMPL087
SEARCH   NI    SWITCH,255-NONE TURN OFF NOT FOUND SWITCH                 SAMPL088
         LM    R1,R3,=F'128,4,128' LOAD TABLE PARAMETERS                 SAMPL089
         LA    R1,TABLAREA-16(R1)  GET ADDRESS OF MIDDLE ENTRY           SAMPL090
LOOP     SRL   R3,1                DIVIDE INCREMENT BY 2                 SAMPL091
         CLC   LNAME,TNAME         COMPARE LIST ENTRY WITH TABLE ENTRY   SAMPL092
         BH    HIGHER              BRANCH IF SHOULD BE HIGHER IN TABLE   SAMPL093
         BCR   8,R14               EXIT IF FOUND                         SAMPL094
         SR    R1,R3               OTHERWISE IT IS LOWER IN THE TABLE   XSAMPL095
                                     SO SUBTRACT INCREMENT               SAMPL096
         BCT   R2,LOOP             LOOP 4 TIMES                          SAMPL097
         B     NOTFOUND            ARGUMENT IS NOT IN THE TABLE          SAMPL098
HIGHER   AR    R1,R3               ADD INCREMENT                         SAMPL099
         BCT   R2,LOOP             LOOP 4 TIMES                          SAMPL100
NOTFOUND OI    SWITCH,NONE         TURN ON NOT FOUND SWITCH              SAMPL101
         BR    R14                 EXIT                                  SAMPL102
*                                                                        SAMPL103
*        THIS IS THE TABLE                                               SAMPL104
*                                                                        SAMPL105
         DS    0D                                                        SAMPL106
TABLAREA DC    XL8'0',CL8'ALPHA'                                         SAMPL107
         DC    XL8'0',CL8'BETA'                                          SAMPL108
         DC    XL8'0',CL8'DELTA'                                         SAMPL109
         DC    XL8'0',CL8'EPSILON'                                       SAMPL110
         DC    XL8'0',CL8'ETA'                                           SAMPL111
         DC    XL8'0',CL8'GAMMA'                                         SAMPL112
         DC    XL8'0',CL8'IOTA'                                          SAMPL113
         DC    XL8'0',CL8'KAPPA'                                         SAMPL114
         DC    XL8'0',CL8'LAMBDA'                                        SAMPL115
         DC    XL8'0',CL8'MU'                                            SAMPL116
         DC    XL8'0',CL8'NU'                                            SAMPL117
         DC    XL8'0',CL8'OMICRON'                                       SAMPL118
         DC    XL8'0',CL8'PHI'                                           SAMPL119
         DC    XL8'0',CL8'SIGMA'                                         SAMPL120
         DC    XL8'0',CL8'ZETA'                                          SAMPL121
*                                                                        SAMPL122
```

```
*        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',FL3'465',A(0)
LISTEND  DC    CL8'ALPHA',X'00',FL3'1',A(123)
*
*        THIS IS THE CONTROL TABLE
*
         DS    0D
TESTTABL DC    FL3'1',X'00',A(123),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    FL3'29',X'0A',A(BEGIN),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    FL3'5',X'05',A(LOOP),CL8'ZETA'
*
*        THIS IS THE CONTROL LIST
*
TESTLIST DC    CL8'LAMBDA',X'0A',FL3'29',A(BEGIN)
         DC    CL8'ZETA',X'05',FL3'5',A(LOOP)
         DC    CL8'THETA',X'82',FL3'45',A(BEGIN)
         DC    CL8'TAU',X'80',FL3'0',A(1)
         DC    CL8'LIST',X'9F',FL3'465',A(0)
         DC    CL8'ALPHA',X'00',FL3'1',A(123)
*
*        THESE ARE THE SYMBOLIC REGISTERS
*
R0       EQU   0
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
*
*        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   BEGIN
```

Features not shown below are common to all assemblers. In the chart:
Dash = Not allowed.
X = As defined in Operating System/360 Assembler Language Manual.

Feature	Basic Programming Support/360: Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	DOS/TOS Assembler	OS/360 Assembler
No. of Continuation Cards/Statement (exclusive of macro-instructions)	0	0	1	1	2
Input Character Code	EBCDIC	BCD & EBCDIC	EBCDIC	EBCDIC	EBCDIC
ELEMENTS:					
Maximum Characters per symbol	6	6	8	8	8
Character self-defining terms	1 Char. only	X	X	X	X
Binary self-defining terms	- -	- -	X	X	X
Length attribute reference	- -	- -	X	X[1]	X
Literals	- -	- -	X	X	X
Extended mnemonics	- -	X	X	X	X
Maximum Location Counter value	$2^{16}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$
Multiple Control Sections per assembly	- -	- -	X	X	X
EXPRESSIONS:					
Operators	+ -*	+ -*/	+ -*/	+ -*/	+ -*/
Number of terms	3	16	3	16	16
Levels of parentheses	- -	- -	1	5	5
Complex relocatability	- -	- -	X	X	X
ASSEMBLER INSTRUCTIONS:					
DC and DS					
Expressions allowed as modifiers	- -	- -	- -	X	X
Multiple operands	- -	- -	- -	X[2]	X
Multiple constants in an operand	- -	- -	Except Address Consts.	X	X
Bit length specifications	- -	- -	- -	X[2]	X
Scale modifier	- -	- -	X	X	X
Exponent Modifier	- -	- -	X	X	X
DC types	Except B, P, Z V, Y, S, L	Except B, V, L	Except L	Except L	X
DC duplication factor	Except A	X	Except S	X	X

[1] Assembler F only

Feature	Basic Programming Support/360: Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	DOS/TOS Assembler	OS/360 Assembler
DC duplication factor of zero	- -	- -	Except S	X	X
DC length modifier	Except H, E, D	X	X	X	X
DS types	Only C, H, F, D	Only C, H, F, D	Except L	X[2]	X
DS length modifer	Only C	Only C	X	X	X
DS maximum length modifier	256	256	256	65,535	65,535
DS constant subfield permitted	- -	- -	X	X	X
COPY	- -	- -	- -	X	X
CSECT	- -	- -	X	X	X
DSECT	--	- -	X	X	X
ISEQ	- -	- -	X	X	X
LTORG	- -	- -	X	X	X
PRINT	- -	- -	X	X	X
TITLE	- -	X	X	X	X
COM	- -	- -	- -	X	X
ICTL	1 operand (1 or 25 only)	1 operand	X	X	X
USING	2 operands (operand 1 relocatable only)	2-17 operands (operand 1 relocatable only)	6 operands	X	X
DROP	1 operand only	X	5 operands	X	X
CCW	operand 2 (relocatable only)	X	X	X	X
ORG	no blank operand	no blank operand	X	X	X
ENTRY	1 operand only	1 operand only	1 operand only	X	X
EXTRN	1 operand only (max 14)	1 operand only	1 operand only	X	X
WXTRN	- -	- -	- -	X[2]	X[1]
CNOP	2 decimal digits	2 decimal digits	2 decimal digits	X	X
PUNCH	- -	- -	- -	X	X
REPRO	- -	- -	X	X	X
Macro Instructions	- -	- -	X	X	X
OPSYN	- -	- -	- -	- -	X[1]
EQU	X	X	X	X	X

[1] Assembler F only

[2] DOS Assembler 14KD only

Features not shown below are common to all assemblers. In the chart:
Dash = Not allowed.
X = As defined in Operating System/360 Assembler Language Manual.

Feature	Basic Programming Support/360: Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	DOS/TOS Assembler	OS/360 Assembler
No. of Continuation Cards/Statement (exclusive of macro-instructions)	0	0	1	1	2
Input Character Code	EBCDIC	BCD & EBCDIC	EBCDIC	EBCDIC	EBCDIC
ELEMENTS:					
Maximum Characters per symbol	6	6	8	8	8
Character self-defining terms	1 Char. only	X	X	X	X
Binary self-defining terms	- -	- -	X	X	X
Length attribute reference	- -	- -	X	X	X
Literals	- -	- -	X	X	X
Extended mnemonics	- -	X	X	X	X
Maximum Location Counter value	$2^{16}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$
Multiple Control Sections per assembly	- -	- -	X	X	X
EXPRESSIONS:					
Operators	+ -*	+ -*/	+ -*/	+ -*/	+ -*/
Number of terms	3	16	3	16	16
Levels of parentheses	- -	- -	1	5	5
Complex relocatability	- -	- -	X	X	X
ASSEMBLER INSTRUCTIONS:					
DC and DS					
Expressions allowed as modifiers	- -	- -	- -	X	X
Multiple operands	- -	- -	- -	X[2]	X
Multiple constants in an operand	- -	- -	Except Address Consts.	X	X
Bit length specifications	- -	- -	- -	X[2]	X
Scale modifier	- -	- -	X	X	X
Exponent Modifier	- -	- -	X	X	X
DC types	Except B, P, Z V, Y, S, L	Except B, V, L	Except L	X[2]	X
DC duplication factor	Except A	X	Except S	X	X

[1]Assembler F only

[2]DOS 14K D Assembler only

Macro Facility Features	BPS 8K Tape, BOS 8K Disk Assemblers	BOS 16K Disk/Tape Assembler	OS/360 Assembler
Operand Sublists	- -	X	X
Attributes of macro-instruction operands inside macro definitions and symbols used in conditional assembly instructions outside macro definitions.	- -	X	X
Subscripted SET symbols	- -	X	X
Maximum number of operands	49	100[1]	200
Conditional assembly instructions outside macro definitions	- -	X	X
Maximum number of SET symbols			
global SETA	16	*	*
global SETB	128	*	*
global SETC	16	*	*
local SETA	16	*	*
local SETB	128	*	*
local SETC	0	*	*

* The number of SET symbols permitted is variable, dependent upon available main storage.

Note: The maximum size of a character expression is 127 characters for the DOS/TOS[1] Assembler D and 255 characters for the OS Assembler F.

[1] 200 for Assembler F

The macro definitions in this appendix are typical applications of the macro language and conditional assembly. Another macro definition is included in the sample program as part of Appendix H.

Notice the use of the inner macro instruction (IHBERMAC) within SAVE for the purpose of generating MNOTE statements. Included with SAVE are some examples of the statements generated from it.

```
MEMBER NAME  SAVE
         MACRO
&NAME    SAVE  &REG,&CODE,&ID
         LCLA  &A,&B,&C
         LCLC  &E,&F,&G,&H
         AIF   ('&REG' EQ '').E1
         AIF   ('&ID' EQ '').NULLID
         AIF   ('&ID' EQ '*').SPECID
&A       SETA  ((K'&ID+2)/2)*2+4
&NAME    B     &A.(0,15)                         BRANCH AROUND ID
&A       SETA  K'&ID
         DC    AL1(&A)                           LENGTH OF IDENTIFIER
.CONTB   AIF   (&A GT 32).SPLITUP
.CONTAA  AIF   (&A GT 8).BRAKDWN
&E       SETC  '&ID'(&B+1,&A)
         DC    CL&A'&E'                          IDENTIFIER
         AGO   .CONTA
.BRAKDWN ANOP
&E       SETC  '&ID'(&B+1,8)
         DC    CL8'&E'                           IDENTIFIER
&B       SETA  &B+8
&A       SETA  &A-8
         AGO   .CONTAA
.SPLITUP ANOP
&E       SETC  '&ID'(&B+1,8)
&F       SETC  '&ID'(&B+9,8)
&G       SETC  '&ID'(&B+17,8)
&H       SETC  '&ID'(&B+25,8)
         DC    CL32'&E.&F.&G.&H'                     IDENTIFIER
&B       SETA  &B+32
&A       SETA  &A-32
         AGO   .CONTB
.NULLID  ANOP
&NAME    DS    0H
         AGO   .CONTA
.SPECID  AIF   ('&NAME' EQ '').CSECTN
&E       SETC  '&NAME'
&A       SETA  1
.CONTQ   AIF   ('&E'(1,&A) EQ '&E').LEAVE
&A       SETA  &A+1
         AGO   .CONTQ
.LEAVE   ANOP
&B       SETA  ((&A+2)/2)*2+4
&NAME    B     &B.(0,15)                         BRANCH AROUND ID
         DC    AL1(&A)
         DC    CL&A'&E'                          IDENTIFIER
         AGO   .CONTA
.CSECTN  AIF   ('&SYSECT' EQ '').E4
&E       SETC  '&SYSECT'
&A       SETA  1
         AGO   .CONTQ
.E4      IHBERMAC 78,360                     CSECT NAME NULL
.CONTA   AIF   (T'&REG(1) NE 'N').E3
         AIF   ('&CODE' EQ 'T').CONTC
         AIF   ('&CODE' NE '').E2
&A       SETA  &REG(1)*4+20
         AIF   (&A LE 75).CONTD
&A       SETA  &A-64
.CONTD   AIF   (N'&REG NE 2).CONTE
         STM   &REG(1),&REG(2),&A.(13)           SAVE REGISTERS
         MEXIT
.CONTE   AIF   (N'&REG NE 1).E3
         ST    &REG(1),&A.(13,0)                 SAVE REGISTER
         MEXIT
.CONTC   AIF   (&REG(1) GE 14 OR &REG(1) LE 2).CONTF
         STM   14,15,12(13)                      SAVE REGISTERS
&A       SETA  &REG(1)*4+20
         AIF   (N'&REG NE 2).CONTG
         STM   &REG(1),&REG(2),&A.(13)           SAVE REGISTERS
         MEXIT
.CONTG   AIF   (N'&REG NE 1).E3
         ST    &REG(1),&A.(13,0)                 SAVE REGISTER
         MEXIT
```

```
.CONTF   AIF   (N'&REG NE 2).CONTH
         STM   14,&REG(2),12(13)                SAVE REGISTERS
         MEXIT
.CONTH   AIF   (N'&REG NE 1).E3
         STM   14,&REG(1),12(13)                SAVE REGISTERS
         MEXIT
.E1      IHBERMAC 18,360                 REG PARAM MISSING
         MEXIT
.E2      IHBERMAC 37,360,&CODE           INVALID CODE SPECIFIED
         MEXIT
.E3      IHBERMAC 36,360,&REG            INVALID REGS. SPECIFIED
         MEND
 END OF DATA FOR SDS OR MEMBER
```

```
*
*                   SAMPLE SAVE MACRO INSTRUCTIONS
*

FOGHORN  SAVE  (14,12)
FOGHORN  DS    0H
         STM   14,12,12(13) SAVE REGISTERS

                         ***********

         SAVE  (REG14,REG12),T
         DS    0H
               12,***  IHB002  INVALID FIRST OPERAND SPECIFIED-(REG14,R

                         ***********

SAVMACRO SAVE  (14,12),T,*
SAVMACRO B     14(0,15) BRANCH AROUND ID
         DC    AL1(8)
         DC    CL8'SAVMACRO' IDENTIFIER

         STM   14,12,12(13) SAVE REGISTERS
```

```
MEMBER NAME  NOTE
         MACRO
&NAME    NOTE  &DCB
         AIF   ('&DCB' EQ '').ERR
&NAME    IHBINNRA &DCB
         L     15,84(0,1)                       LOAD NOTE  RTN ADDRESS
         BALR  14,15                            LINK TO NOTE  ROUTINE
         MEXIT
.ERR     IHBERMAC 6
         MEND
```

```
MEMBER NAME  POINT
         MACRO
&NAME    POINT &DCB,&LOC
         AIF   ('&DCB' EQ '').ERR1
         AIF   ('&LOC' EQ '').ERR2
&NAME    IHBINNRA &DCB,&LOC
         L     15,84(0,1)                       LOAD POINT RTN ADDRESS
         BAL   14,4(15,0)                       LINK TO POINT ROUTINE
         MEXIT
.ERR1    IHBERMAC 6
         MEXIT
.ERR2    IHBERMAC 3
         MEND
```

```
MEMBER NAME  CHECK
         MACRO
&NAME    CHECK &DECB
         AIF   ('&DECB' EQ '').E1
&NAME    IHBINNRA &DECB
         L     14,8(0,1)                        PICK UP DCB ADDRESS
         L     15,52(0,14)                      LOAD CHECK ROUT. ADDR.
         BALR  14,15                            LINK TO CHECK ROUTINE
         MEXIT
.E1      IHBERMAC 07,018
         MEND
```

Indexes to systems reference library manuals are consolidated in the publication IBM System/360 Operating System Systems Reference Library Master Index, Order No. GC28-6644. For additional information about any subject listed below, refer to other publications listed for the same subject in the Master Index.

GC28-6514-7

System/360 OS Assembler Language (S360-21) Printed in U. S. A. GC28-6514-7

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]