NASA STANDARD SPACECRAFT COMPUTER NSSC-II

ASSEMBLER LANGUAGE

IBM No. 75W-00142

July 15, 1975

Federal Systems Division, Civil and Space Systems, Huntsville, Alabama

CHANGE NOTICE

This revision cancels and supersedes the change issued February 1, 1974.

Copies of this document may be obtained from the IBM Corporation (Department U34), 150 Sparkman Drive, Huntsville, Alabama 35803.

HTC/OS ASSEMBLER LANGUAGE

This publication contains specifications for the IBM System/360 Operating System Assembler Language (Level F) modified to be compatible with the extended Hybrid Technology Computer (HTC), i.e., the HTC including the short precision option and the double precision fixed-point arithmetic option.

The assembler language is a symbolic programming language used to write programs for the HTC. The language provides a convenient means for representing the machine instructions and related data necessary to program the HTC. The IBM HTC 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.

Appendices A through J follow Part II. Appendices 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 HTC 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. HTC machine operations are discussed in the publication "HTC Principles of Operation." Information on program assembling, linkage editing, executing, interpreting listings, and assembler programming considerations is provided in "OS Assembler (F) Programmer's Guide," Order No. GC26-3756.

The following publications are referred to in this publication:

OS Introduction, Order No. GC28-6534

OS Utilities, Order No. GC28-6586

OS Loader and Linkage Editor, Order No. GC28-6538

OS Supervisor Services and Macro Instructions, Order No. GC28-6646

OS Data Management Macro Instructions, Order No. GC26-3794

OS Data Management Services Guide, Order No. GC26-3746

CONTENTS

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

SECTION 1: INTRODUCTION

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

The HTC assembler uses the standard S/360 instruction set with the following exceptions:

1. The HTC I/O is different from S/360 and only uses the SIO instruction. The SIO instruction format has been changed from SI to an RS format. The S/360 TIO, HIO, and TCH instructions are not supported by the HTC assembler.

2. A new instruction, Timer Read and Set (TMRS) has been added for the HTC. The TMRS instruction has an RS format and the storage operand must be aligned on a halfword boundary.

3. The HTC assembler does not support the S/360 Floating Point Feature instructions, the Decimal Feature instructions, the Direct Control Feature instructions, the Channel Command Word (CCW) assembler instruction, or the Insert Storage Key (ISK) instruction.

4. No S/370 instructions are supported by the HTC assembler.

5. All extended HTC instructions, i.e., instructions in the short precision option or the double precision fixed-point arithmetic option, are not in the S/360 instruction set.

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. The output object program produced by the assembler will be linkage edited by a S/360 Linkage Editor. The HTC Formatter Program translates linkage editor output into magnetic tape or paper tape forms for loading into the HTC.

SECTION 2: GENERAL INFORMATION

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. The standard card form, IBM 6509 (shown in Figure 2-2), can be used for punching source statements. These statements may be written on the standard coding form, GX28-6509 (shown in Figure 2-1), provided by IBM. One line of coding on the form is punched into one card. 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.

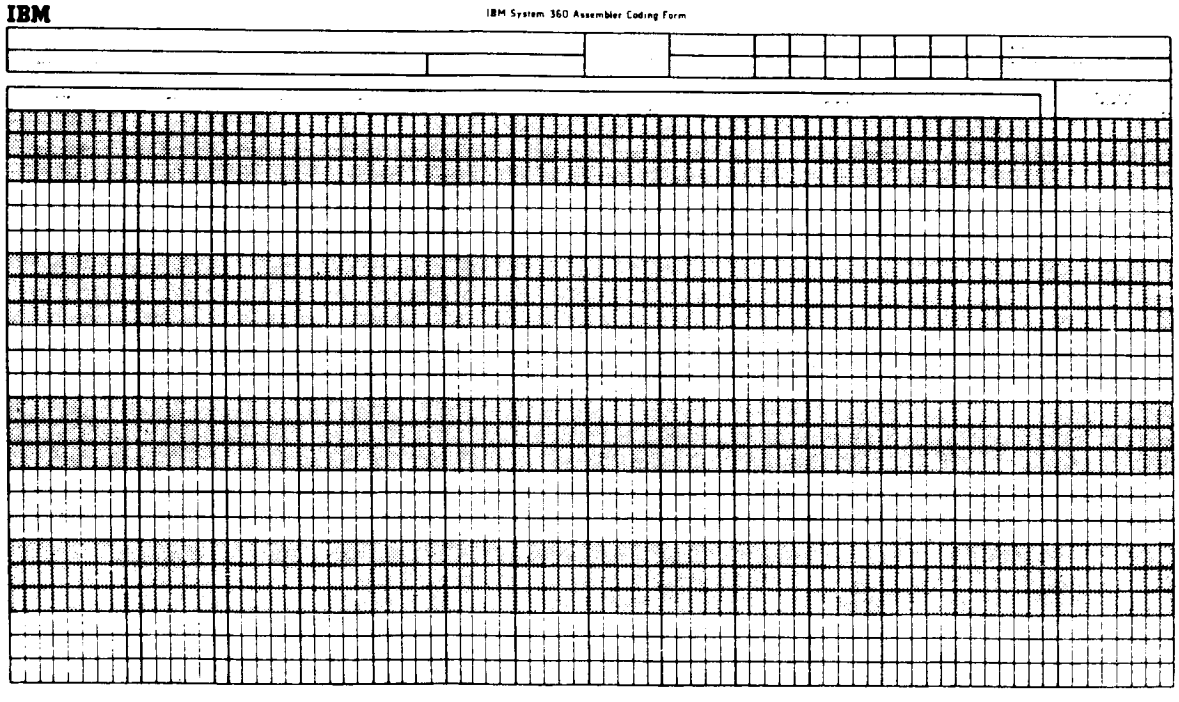


Figure 2-1. Coding Form

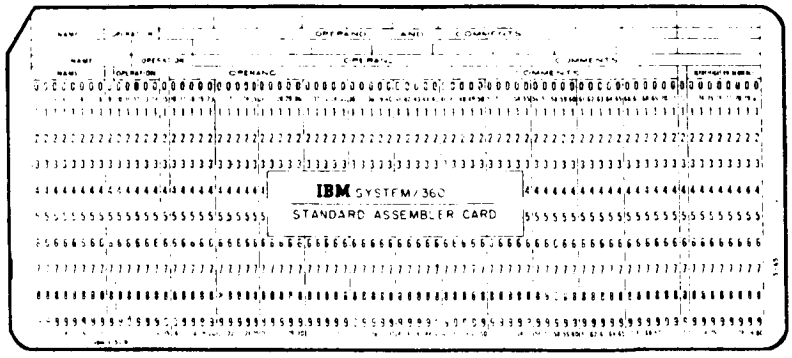


Figure 2-2. Punched Card Form

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).

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 Appendices 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. 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 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, 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 arithemtic combination of terms is reduced to a single value by the assembler.

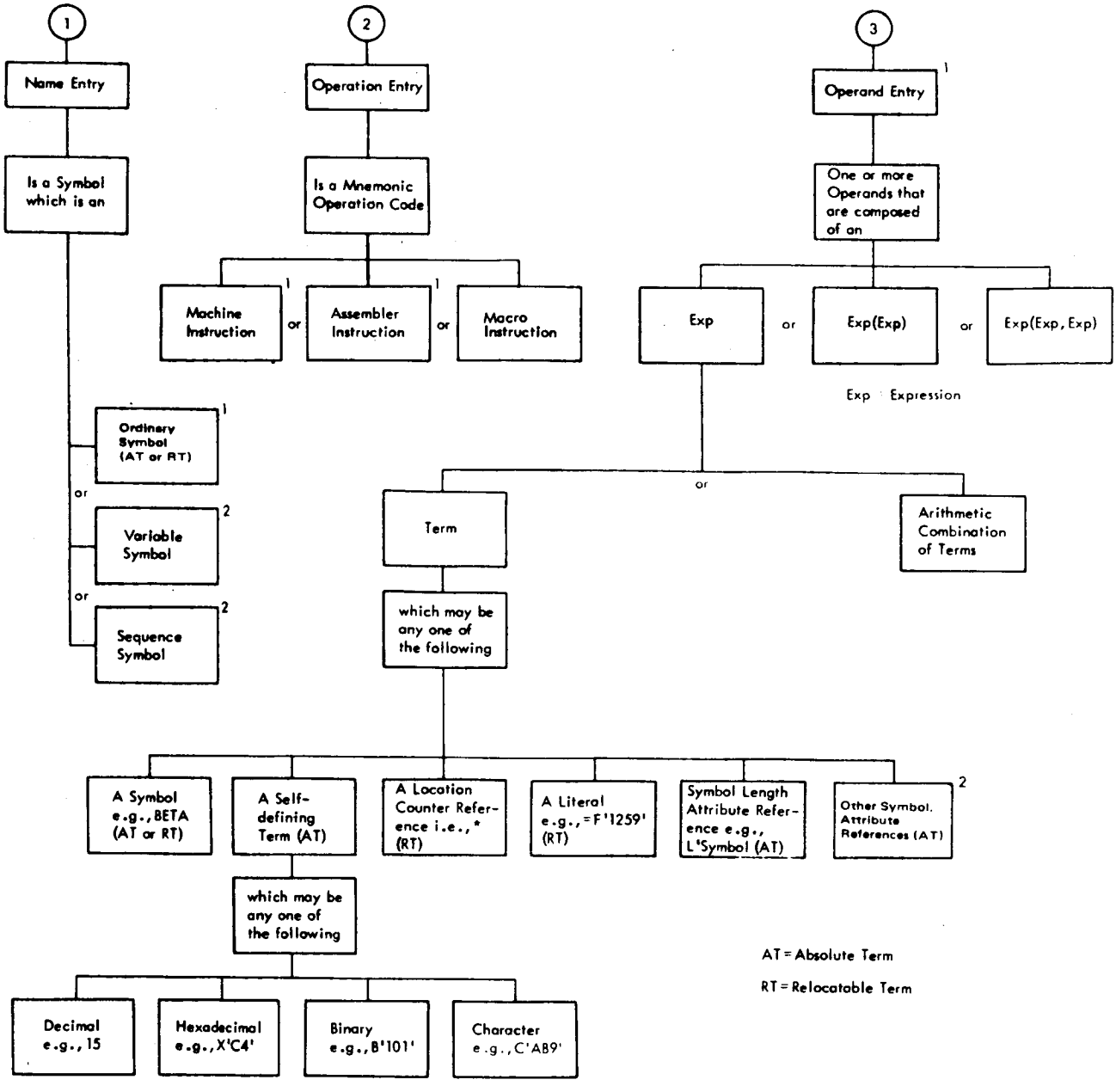
The following material discusses each type of term and the rules for its use.

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] 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

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.

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 symbols 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 designated 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 use of the length attribute of a symbol defined with a DC or DS with explicit length given by an expression is invalid. 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, depend-

ing 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 twos 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      Y
  location counter)
```

SECTION 3: ADDRESSING -- PROGRAM SECTIONING AND LINKING

ADDRESSING

The IBM HTC 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.

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.

```
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

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

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 properties:

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 doubleword 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 unnamed control 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: Symbols that name statements in a dummy section may be used in A-type address constants only when they are paired with another symbol from the same dummy section in an absolute expression. (See "Absolute and Relocatable Expressions", Section 2.) For example, if X and B name statements in the same dummy section, C DC A(B-X) would be valid, but C DC A(X) would be invalid--yielding a relocatability error.

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 can 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 DC instruction. 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 EXTRN	RATEBL,PAYCALC 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 following 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 add to register 3 the contents of a data 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 OS Loader and Linkage Editor 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.

SECTION 4: MACHINE-INSTRUCTIONS

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
S	4
SI	4
SS	6
RI	4
R	2

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.

RI Format

Name	Operation	Operand
ALPHA1	AHI	REG1,X'1234'
ALPHA2	AHI	2,C'AB'
BETA	AHI	REG2,TEN

The operands of ALPHA1 and ALPHA2 are self-defining values which are categorized as absolute expressions. The operand of BETA is a symbol which is equated elsewhere to an absolute value.

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'

The ALPHA instructions specify explicit addresses, whereas the BETA instructions specify implied addresses.

S Format

Name	Operation	Operand
GAMMA1	SIO	40(9)
GAMMA2	SIO	0(9)
GAMMA3	SIO	40(0)
GAMMA4	SIO	ZETA

The GAMMA1, GAMMA2, and GAMMA3 instructions specify explicit addresses. The GAMMA4 instruction specifies an implied address. The GAMMA2 instruction specifies a displacement of zero. The GAMMA3 instruction 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.

SECTION 5: ASSEMBLER INSTRUCTION STATEMENTS

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

* 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
PUNCH - Punch a Card
REPRO - Reproduce Following Card
ORG - Set Location Counter
LTORG - Begin Literal Pool
CNOP - Conditional No Operation
COPY - Copy Predefined Source Coding
END - End Assembly

SYMBOL DEFINITION INSTRUCTION

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 relocatability 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 "OS Assembler (F) Programmer's Guide."

DATA DEFINITION INSTRUCTIONS

There are two data definition instruction statements: Define Constant (DC) and Define Storage (DS).

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 difference 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.

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.

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
I	ASCII	8-bit ASCII code for each character
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
W	Address	Value of address; a full 16-bit halfword
Q	Address	Space reserved for dummy section offset (Assembler F only)

Figure 5-1. Type Codes for Constants

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:

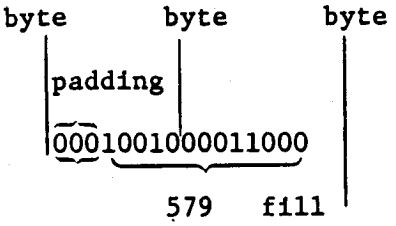


Figure 5-2. Bit-Length Specification (Single Constant)

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:

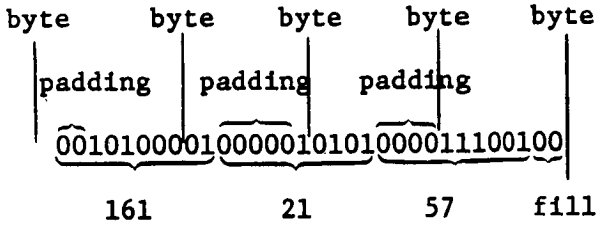


Figure 5-3. Bit-Length Specification (Multiple Constants)

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:

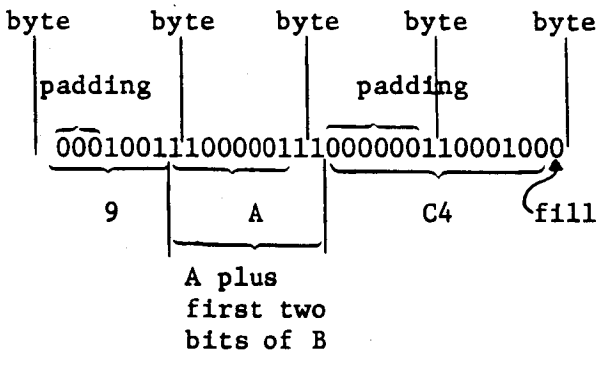


Figure 5-4. Bit-Length Specification (Multiple Operands)

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.

EXPONENT MODIFIER: This modifier is written as En, where n is either a decimal self-defining term or an absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. The decimal value 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 IBM "System/360 Principles of Operation."

Operand Subfield 4: Constant

This subfield supplied 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 Constant	Multiple Constants*
'constant'	'constant,...,constant'
(constant)	(constant,...,constant)

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

All constant types except character (C), hexadecimal (X), binary (B), packed decimal (P), and zoned decimal (Z), are aligned on the proper boundary, as shown in Appendix F, unless a length modifier is specified. In the presence of a length modifier, 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'

ASCII Character Constant -- I: This constant is identically the same as the above C type constant except that an ASCII character set is used instead of the EBCDIC. Appendix A specifies these codes.

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 or 512 hexadecimal digits when specified using an explicit length attribute (for example, HEX DC XL256'FF'). However, due to

the assembler's syntax restriction allowing only two continuation lines per input statement, the maximum length of an implicitly specified hexadecimal operand (X'FFFFFF', etc.) is 176 digits when normal statement boundaries are used.

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. No boundary alignment is performed.

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}
.4	$2^{3}-1$	-2^{3}
.2	$2^{1}-1$	-2^{1}
.1	0	-1

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:

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.

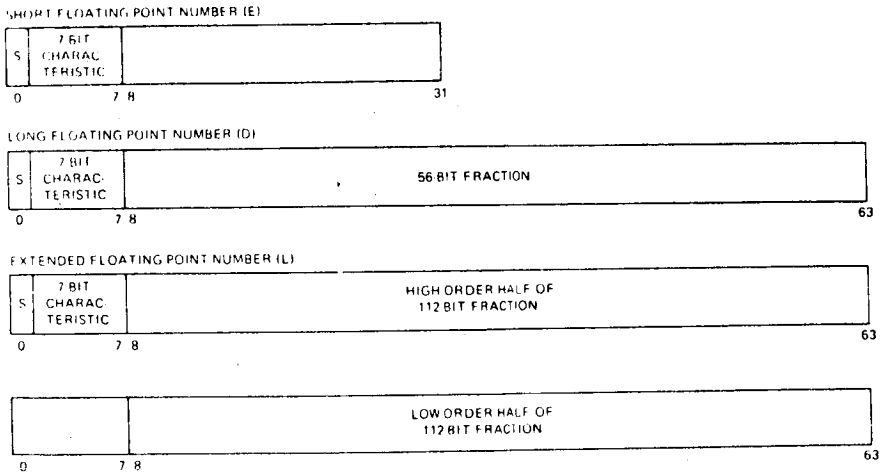


Figure 5-5. Floating-Point External Formats

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 "OS 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 two's 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 using "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 to 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	Operation	Operand
ACON	DC LM	A(108,LOP,END-STRT,*+4096) 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 rightmost 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.

W-Type Address Constant: A W-type address constant is identical with the Y-type except that maximum value can be 2^{15} as opposed to the $2^{15}-1$ allowed by the Y-type.

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 an 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

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.

Note: 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.

4. 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 zre 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 1,r;or blank

The operands 1 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 1. Columns to be checked must not be between the begin and end columns.

Sequence checking begins with the first card following the ISEQ statement. Comparison of adjacent cards makes use of the 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.

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 double-word boundary following the LTORG statement. If no literals follow the LTORG statement, alignment of the next instruction (which is not an 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
 SH  3,=H'2'             ②
 LM  0,3,=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 an 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

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.

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

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

Name	Operation	Operand
	CNOP BALR	0,8 2,14

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

Name	Operation	Operand
	CNOP BALR	6,8 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 BCR BCR BALR	0,0 0,0 0,0 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 IEBUPDAT or IEBUPDTE routines, details of which are covered in the "OS Utilities."

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

SECTION 6: INTRODUCTION TO 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 facilty 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 "OS Supervisor Services and Macro Instructions," and "OS Data Management Macro Instructions."

VARYING THE GENERATED STATEMENTS

Each time a macro-instruction appears in the source program it is replaced by the same sequence of assembler language statements. 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 of 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.

SECTION 7: HOW TO PREPARE MACRO-DEFINITIONS

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" in this section.

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.

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

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

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 field.

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
- Macro prototypes
- The following instructions:

ACTR	GBLA	MEXIT
AGO	GBLB	MNOTE
AIF	GBLC	OPSYN
ANOP	ICTL	PRINT
COPY	ISEQ	REPRO
CSECT	LCLA	SETA
DSECT	LCLB	SETB
END	LCLC	SETC
	MACRO	START
	MEND	

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, 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 were used in a source program, the following assembler language statements would be generated:

Name	Operation	Operand
HERE	ST L ST L	2,SAVE 2,FIELDB 2,FIELDA 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 "OS Utilities" manual.

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.

SECTION 8: HOW TO WRITE MACRO-INSTRUCTIONS

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." Unlike prototype statements, macro-instructions can have omitted operands, and they can have consecutive commas or a comma at the end of the operand list.

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 sublist, 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 Model		ADD L	&NUM,&REG,&AREA &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.

SECTION 9: HOW TO WRITE CONDITIONAL ASSEMBLY INSTRUCTIONS

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. The use of the length attribute of a symbol defined with a DC or DS with explicit length given by an expression is invalid. 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, U, 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 arithemetic 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 chracter 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 prototytpe 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. Each arithmetic expression 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 for 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.

Note: An operand of a SETC symbol cannot be passed as a sublist to a macro-instruction.

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')
(0)
```

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 sequence
                               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 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 in 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 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

SECTION 10: EXTENDED FEATURES OF THE MACRO LANGUAGE

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 a 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 a 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 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 a 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-definition, 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

        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, OUTER1, 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(0) may be used to refer to a symbolic parameter in the macro-instruction prototype. If the symbolic parameter is omitted in the macro-instruction prototype, then &SYSLIST(0) would refer to a null character value.

&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.

If the value of "subscript n is zero", then &SYSLIST(n) is assigned the value specified in the name field of the macro-instruction, except when it is a sequence symbol.

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 operand.

2. The count includes operands specifically omitted (by means of commas).

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. 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 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.

Note 1: 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.

Note 2: Positional parameters cannot be changed to keywords by substitution. That is, in the following example, the expression A=FB, statement 2, will be treated as a positional operand consisting of a character string in the generation of the MAC macro; it will not be treated as a keyword A with the value FB.

```
        Name        Operation       Operand
1                   GBLC            &VALUE
2       &VALUE      SETC            'A=FB'
3                   MAC             &VALUE
```

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

```
        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
```

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.

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.

Operand Sublists: 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."

Keyword Inner Macro-Instructions: Keyword and positional inner macro-instructions may be used as model statements in either keyword or positional macro-definitions.

MIXED-MODE MACRO-DEFINITIONS AND INSTRUCTIONS

Mixed-mode macro-definitions allow the programmer to use the features of keyword and positional macro-definitions in the same macro-definition.

Mixed-mode macro-definitions are prepared 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 macro-instruction. 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	One 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
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

8-Bit Code	Character Set Punch Combination	Decimal	Hexa-Decimal	ASCII Code	EBCDIC Printer Graphics	ASCII Printer Graphics
00000000	12,0,9,8,1	0	00	00		NUL
00000001	12,9,1	1	01	01		SOH
00000010	12,9,2	2	02	02		STX
00000011	12,9,3	3	03	03		ETX
00000100	12,9,4	4	04	04		EOT
00000101	12,9,5	5	05	09		HT
00000110	12,9,6	6	06	06		ACK
00000111	12,9,7	7	07	7F		DEL
00001000	12,9,8	8	08	08		BS
00001001	12,9,8,1	9	09	09		HT
00001010	12,9,8,2	10	0A	0A		LF
00001011	12,9,8,3	11	0B	0B		VT
00001100	12,9,8,4	12	0C	0C		FF
00001101	12,9,8,5	13	0D	0D		CR
00001110	12,9,8,6	14	0E	0E		SO
00001111	12,9,8,7	15	0F	0F		SI
00010000	12,11,9,8,1	16	10	10		DLE
00010001	11,9,1	17	11	11		DC1
00010010	11,9,2	18	12	12		DC2
00010011	11,9,3	19	13	13		DC3
00010100	11,9,4	20	14	14		DC4
00010101	11,9,5	21	15	15		NAK
00010110	11,9,6	22	16	08		BS
00010111	11,9,7	23	17	17		ETB
00011000	11,9,8	24	18	18		CAN
00011001	11,9,8,1	25	19	19		EM
00011010	11,9,8,2	26	1A	1A		SUB
00011011	11,9,8,3	27	1B	1B		ESC
00011100	11,9,8,4	28	1C	1C		FS
00011101	11,9,8,5	29	1D	1D		GS
00011110	11,9,8,6	30	1E	1E		RS
00011111	11,9,8,7	31	1F	1F		US
00100000	11,0,9,8,1	32	20	20		SP
00100001	0,9,1	33	21	21		1①
00100010	0,9,2	34	22	22		"
00100011	0,9,3	35	23	23		#
00100100	0,9,4	36	24	24		$
00100101	0,9,5	37	25	0A		LF
00100110	0,9,6	38	26	17		ETB
00100111	0,9,7	39	27	1B		ESC
00101000	0,9,8	40	28	28		(
00101001	0,9,8,1	41	29	29		)

8-Bit Code	Character Set Punch Combinaions	Decimal	Hexa-Decimal	ASCII Code	EBDIC Printer Graphics	ASCII Printer Graphics
00101010	0,9,8,2	42	2A	2A		*
00101011	0,9,8,3	43	2B	2B		+
00101100	0,9,8,4	44	2C	2C		,
00101101	0,9,8,5	45	2D	05		ENQ
00101110	0,9,8,6	46	2E	06		ACK
00101111	0,9,8,7	47	2F	07		BEL
00110000	12,11,0,9,8,1	48	30	30		0
00110001	9,1	49	31	31		1
00110010	9,2	50	32	16		SYN
00110011	9,3	51	33	33		3
00110100	9,4	52	34	34		4
00110101	9,5	53	35	35		5
00110110	9,6	54	36	36		6
00110111	9,7	55	37	04		EOT
00111000	9,8	56	38	38		8
00111001	9,8,1	57	39	39		9
00111010	9,8,2	58	3A	3A		:
00111011	9,8,3	59	3B	3B		;
00111100	9,8,4	60	3C	14		DC4
00111101	9,8,5	61	3D	15		NAK
00111110	9,8,6	62	3E	3E		>
00111111	9,8,7	63	3F	1A		SUB
01000000		64	40	20	(blank)	,
01000001	12,0,9,1	65	41	41		A
01000010	12,0,9,2	66	42	42		B
01000011	12,0,9,3	67	43	43		C
01000100	12,0,9,4	68	44	44		D
01000101	12,0,9,5	69	45	45		E
01000110	12,0,9,6	70	46	46		F
01000111	12,0,9,7	71	47	47		G
01001000	12,0,9,8	72	48	48		H
01001001	12,8,1	73	49	49		I
01001010	12,8,2	74	4A	5B	¢ (cent sign)	E
01001011	12,8,3	75	4B	2E	. (period)	.
01001100	12,8,4	76	4C	3C	<	<
01001101	12,8,5	77	4D	28	(	(
01001110	12,8,6	78	4E	2B	+	+
01001111	12,8,7	79	4F	21	| (Logical OR)	|
01010000	12	80	50	26	&	&
01010001	12,11,9,1	81	51	51		Q
01010010	12,11,9,2	82	52	52		R
01010011	12,11,9,3	83	53	53		S
01010100	12,11,9,4	84	54	54		T
01010101	12,11,9,5	85	55	55		U

8-Bit Code	Character Set Punch Combination	Decimal	Hexa-Decimal	ASCII Codes	EBCDIC Printer Graphics	ASCII Printer Graphics
01010110	12,11,9,6	86	56	56		V
01010111	12,11,9,7	87	57	57		W
01011000	12,11,9,8	88	58	58		X
01011001	11,8,1	89	59	59		Y
01011010	11,8,2	90	5A	5D	!	]
01011011	11,8,3	91	5B	24	$	$
01011100	11,8,4	92	5C	2A	*	*
01011101	11,8,5	93	5D	29	)	)
01011110	11,8,6	94	5E	3B	;	;
01011111	11,8,7	95	5F	3E	¬(logical NOT)	¬
01100000	11	96	60	2D	- (hyphen)	-
01100001	0,1	97	61	2F	/	/
01100010	11,0,9,2	98	62	62		b
01100011	11,0,9,3	99	63	63		c
01100100	11,0,9,4	100	64	64		d
01100101	11,0,9,5	101	65	65		e
01100110	11,0,9,6	102	66	66		f
01100111	11,0,9,7	103	67	67		g
01101000	11,0,9,8	104	68	68		h
01101001	0,8,1	105	69	69		/
01101010	12,11	106	6A	7C		/
01101011	0,8,3	107	6B	2C	, (comma)	,
01101100	0,8,4	108	6C	25	%	%
01101101	0,8,5	109	6D	5F	_ (underscore)	_ (under-score)
01101110	0,8,6	110	6E	3E	>	>
01101111	0,8,7	111	6F	3F	?	?
01110000	12,11,0	112	70	70		p
01110001	12,11,0,9,1	113	71	71		q
01110010	12,11,0,9,2	114	72	72		r
01110011	12,11,0,9,3	115	73	73		s
01110100	12,11,0,9,4	116	74	74		t
01110101	12,11,0,9,5	117	75	75		u
01110110	12,11,0,9,6	118	76	76		v
01110111	12,11,0,9,7	119	77	77		w
01111000	12,11,0,9,8	120	78	78		x
01111001	8,1	121	79	60		`
01111010	8,2	122	7A	3A	:	:
01111011	8,3	123	7B	23	#	#
01111100	8,4	124	7C	40	@	@
01111101	8,5	125	7D	27	' (apostrophe)	' (apos-trophe)
01111110	8,6	126	7E	3D	=	=
01111111	8,7	127	7F	22	"	"

8-Bit Code	Character Set Punch Combination	Decimal	Hexa-Decimal	ASCII Code	EBCDIC Printer Graphics	ASCII Printer Graphics
10000000	12,0,8,1	128	80	80		
10000001	12,0,1	129	81	61	a	a
10000010	12,0,2	130	82	62	b	b
10000011	12,0,3	131	83	63	c	c
10000100	12,0,4	132	84	64	d	d
10000101	12,0,5	133	85	65	e	e
10000110	12,0,6	134	86	66	f	f
10000111	12,0,7	135	87	67	g	g
10001000	12,0,8	136	88	68	h	h
10001001	12,0,9	137	89	69	i	i
10001010	12,0,8,2	138	8A	8A		
10001011	12,0,8,3	139	8B	8B		
10001100	12,0,8,4	140	8C	8C		
10001101	12,0,8,5	141	8D	8D		
10001110	12,0,8,6	142	8E	8E		
10001111	12,0,8,7	143	8F	8F		
10010000	12,11,8,1	144	90	90		
10010001	12,11,1	145	91	6A	j	j
10010010	12,11,2	146	92	6B	k	k
10010011	12,11,3	147	93	6C	l	l
10010100	12,11,4	148	94	6D	m	m
10010101	12,11,5	149	95	6E	n	n
10010110	12,11,6	150	96	6F	o	o
10010111	12,11,7	151	97	70	p	p
10011000	12,11,8	152	98	71	q	q
10011001	12,11,9	153	99	72	r	r
10011010	12,11,8,2	154	9A	9A		
10011011	12,11,8,3	155	9B	9B		
10011100	12,11,8,4	156	9C	9C		
10011101	12,11,8,5	157	9D	9D		
10011110	12,11,8,6	158	9E	9E		
10011111	12,11,8,7	159	9F	9F		
10100000	11,0,8,1	160	A0	A0		
10100001	11,0,1	161	A1	7E		~
10100010	11,0,2	162	A2	73	s	s
10100011	11,0,3	163	A3	74	t	t
10100100	11,0,4	164	A4	75	u	u
10100101	11,0,5	165	A5	76	v	v
10100110	11,0,6	166	A6	77	w	w
10100111	11,0,7	167	A7	78	x	x
10101000	11,0,8	168	A8	79	y	y
10101001	11,0,9	169	A9	7A	z	z
10101010	11,0,8,2	170	AA	AA		
10101011	11,0,8,3	171	AB	AB		

8-Bit Code	Character Set Punch Combination	Decimal	Hexa-Decimal	ASCII Code	EBCDIC Printer Graphics	ASCII Printer Graphics
10101100	11,0,8,4	172	AC	AC		
10101101	11,0,8,5	173	AD	AD		
10101110	11,0,8,6	174	AE	AE		
10101111	11,0,8,7	175	AF	AF		
10110000	12,11,0,8,1	176	B0	B0		
10110001	12,11,0,1	177	B1	B1		
10110010	12,11,0,2	178	B2	B2		
10110011	12,11,0,3	179	B3	B3		
10110100	12,11,0,4	180	B4	B4		
10110101	12,11,0,5	181	B5	B5		
10110110	12,11,0,6	182	B6	B6		
10110111	12,11,0,7	183	B7	B7		
10111000	12,11,0,8	184	B8	B8		
10111001	12,11,0,9	185	B9	B9		
10111010	12,11,0,8,2	186	BA	BA		
10111011	12,11,0,8,3	187	BB	BB		
10111100	12,11,0,8,4	188	BC	BC		
10111101	12,11,0,8,5	189	BD	BD		
10111110	12,11,0,8,6	190	BE	BE		
10111111	12,11,0,8,7	191	BF	BF		
11000000	12,0	192	C0	7B		{
11000001	12,1	193	C1	41	A	A
11000010	12,2	194	C2	42	B	B
11000011	12,3	195	C3	43	C	C
11000100	12,4	196	C4	44	D	D
11000101	12,5	197	C5	45	E	E
11000110	12,6	198	C6	46	F	F
11000111	12,7	199	C7	47	G	G
11001000	12,8	200	C8	48	H	H
11001001	12,9	201	C9	49	I	I
11001010	12,0,9,8,2	202	CA	CA		
11001011	12,0,9,8,3	203	CB	CB		
11001100	12,0,9,8,4	204	CC	CC		
11001101	12,0,9,8,5	205	CD	CD		
11001110	12,0,9,8,6	206	CE	CE		
11001111	12,0,9,8,7	207	CF	CF		
11010000	11,0	208	D0	7D		}
11010001	11,1	209	D1	4A	J	J
11010010	11,2	210	D2	4B	K	K
11010011	11,3	211	D3	4C	L	L
11010100	11,4	212	D4	4D	M	M
11010101	11,5	213	D5	4E	N	N
11010110	11,6	214	D6	4F	O	O
11010111	11,7	215	D7	50	P	P

8-Bit Code	Character Set Punch Combination	Decimal	Hexa-Decimal	ASCII Code	EBCDIC Printer Graphics	ASCII Printer Graphics
11011000	11,8	216	D8	51	Q	Q
11011001	11,9	217	D9	52	R	R
11011010	12,11,9,8,2	218	DA	DA		
11011011	12,11,9,8,3	219	DB	DB		
11011100	12,11,9,8,4	220	DC	DC		
11011101	12,11,9,8,5	221	DD	DD		
11011110	12,11,9,8,6	222	DE	DE		
11011111	12,11,9,8,7	223	DF	DF		
11100000	0,8,2	224	E0	E0		
11100001	11,0,9,1	225	E1	5C		\
11100010	0,2	226	E2	53	S	S
11100011	0,3	227	E3	54	T	T
11100100	0,4	228	E4	55	U	U
11100101	0,5	229	E5	56	V	V
11100110	0,6	230	E6	57	W	W
11100111	0,7	231	E7	58	X	X
11101000	0,8	232	E8	59	Y	Y
11101001	0,9	233	E9	5A	Z	Z
11101010	11,0,9,8,2	234	EA	EA		
11101011	11,0,9,8,3	235	EB	EB		
11101100	11,0,9,8,4	236	EC	EC		
11101101	11,0,9,8,5	237	ED	ED		
11101110	11,0,9,8,6	238	EE	EE		
11101111	11,0,9,8,7	239	EF	EF		
11110000	0	240	F0	30	0	0
11110001	1	241	F1	31	1	1
11110010	2	242	F2	32	2	2
11110011	3	243	F3	33	3	3
11110100	4	244	F4	34	4	4
11110101	5	245	F5	35	5	5
11110110	6	246	F6	36	6	6
11110111	7	247	F7	37	7	7
11111000	8	248	F8	38	8	8
11111001	9	249	F9	39	9	9
11111010	12,11,0,9,8,2	250	FA	FA		
11111011	12,11,0,9,8,3	251	FB	FB		
11111100	12,11,0,9,8,4	252	FC	FC		
11111101	12,11,0,9,8,5	253	FD	FD		
11111110	12,11,0,9,8,6	254	FE	FE		
11111111	12,11,0,9,8,7	255	FF	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	-	Number Sign, Hyphen	@	At Sign
|	Vertical Bar, Logical OR	/	Slash	'	Prime, Apostrophe
&	Ampersand	,	Comma	=	Equal
!	Exclamation Point	%	Percent	"	Quotation Mark
$	Dollar Sign	_	Underscore		

Examples	Type	Bit Pattern Bit Positions 01 23 4567	Hole Pattern	
			Zone Punches	Digit Punches
PF	Control Character	00 00 0100	12-9-4	
%	Special Graphic	01 10 1100	0-8-4	
R	Upper Case	11 01 1101	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	

APPENDIX B: HEXADECIMAL-DECIMAL NUMBER CONVERSION TABLE

From hex: Locate each hex digit in its corresponding column position and note the decimal equivalents. Add these to obtain the decimal value.

From decimal: (1) Locate the largest decimal value in the table that will fit into the decimal number to be converted, and (2) note its hex equivalent and hex column position. (3) Find the decimal remainder. Repeat the process on this and subsequent remainders.

NOTE: Decimal, hexadecimal, (and binary) equivalents of all numbers from 0 to 255 are listed on panels 11-14.

HEXADECIMAL COLUMNS												
6		5		4		3		2		1		
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	
0	0	0	0	0	0	0	0	0	0	0	0	
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1	
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2	
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3	
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4	
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5	
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6	
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7	
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8	
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9	
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10	
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11	
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12	
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13	
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14	
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15	
0 1 2 3		4 5 6 7		0 1 2 3		4 5 6 7		0 1 2 3		4 5 6 7		
BYTE		BYTE						BYTE				

POWERS OF 2

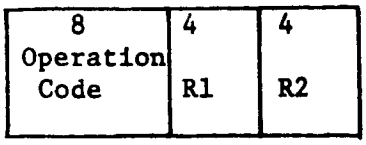
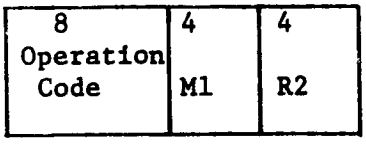
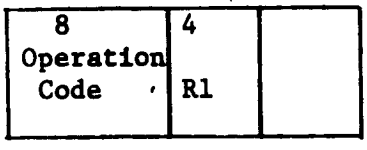
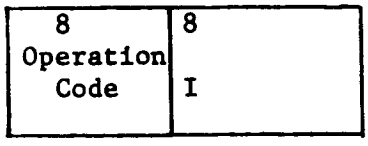
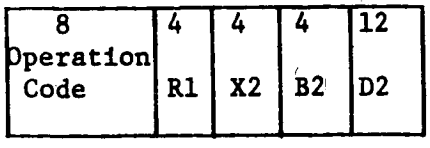
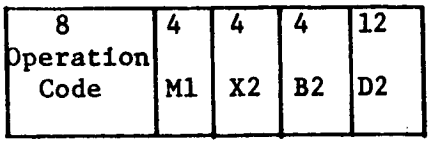
2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

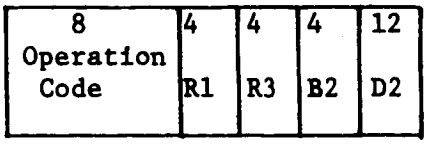
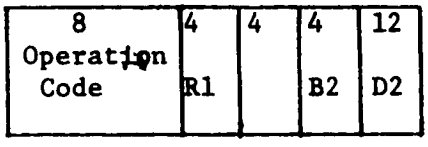
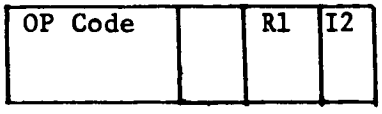
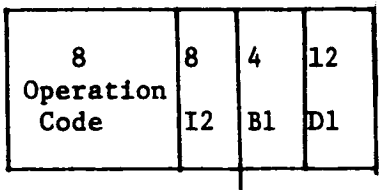
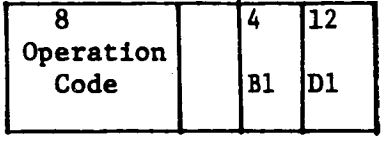
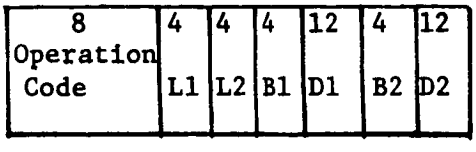
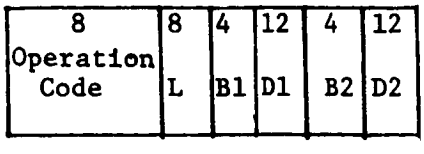
$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

POWER OF 16

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

APPENDIX C: MACHINE-INSTRUCTION FORMAT

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
RR		R1,R2	All RR instructions except BCR, SPM, BR, and SVC
		M1,R2	BCR
		R1	BR
		R1	SPM
		I (See Notes 1,6,8, and 9)	SVC
RX		R1,D2(X2,B2) R1,D2(,B2) R1,S2(X2) R1,S2	All RX instructions except BC and BUR
		D2(X1,B1) D1(,B1) S1(X1)	BUR
		M1,D2(X2,B2) M1,D2(,B2) M1,S2(X2) M1,S2 (See Notes 1,6,8, and 9)	BC

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
RS		R1,R3,D2(B2) R1,R3,S2	BXH,BXLE,LM,STM,SIO,TMRS
		R1,D2(B2) R1,S2	All shift instructions
RI		R1,I	All 16 bit immediate instructions
SI		D1(B1),I2 S1,I2	All SI instructions except those listed for other SI formats
		D1(B1) S1 (See Notes 2,3, 6,7, and 8)	LPSW,SSM,TIO,TCH,TS
SS		D1(L1,B1),D2(L2,B2) S1(L1),S2(L2)	PACK,UNPK,MVO,AP, CP,DP,MP,SP,ZAP
		D1(L,B1),D2(B2) S1(L),S2	NC,OC,XC,CLC,MVC,MVN, MVZ,TR,TRT,ED,EDMK

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 specifies a 4-bit mask.

APPENDIX D. Machine Instruction Mnemonic Operation Codes

This appendix contains two tables of the mnemonic operation codes for all machine instructions that can be represented in assembler language, including extended mnemonic operation codes.

The first table is in alphabetic order by instruction. The second table is in numeric order by operation code.

In the first table is indicated: both the mnemonic and machine operation codes, explicit and implicit operand formats, program interruptions possible, and condition code set.

The column headings in the first table and the information each column provides follow:

Instruction: This column contains the name of the instruction associated with the mnemonic operation code.

Mnemonic Operation Code: This column contains the mnemonic operation code for the 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, or operand two respectively. M1 and M3 indicate a 4-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, SSI or RI). 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	Implicit
Add	A	5A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add	AR	1A	R1,R2	
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	
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),12	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 in 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

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 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
And Logical	RX	X	X					J	K		
And Logical	SS	X			X			J	K		
And Logical	RR							J	K		
And Logical Immediate	SI	X			X			J	K		
Branch and Link	RX							N	N	N	N
Branch and Link	RR							N	N	N	N
Branch 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

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
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 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 Immediate	CLI	95	D1(B1),I2	S1,I2
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
Divide	D	5D	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide	DR	1D	R1,R2	
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),D1(B2)	S1(L),S2 or S1,S2
Exclusive Or	XR	17	R1,R2	
Exclusive Or Immediate	XI	97	D1(B1),12	S1,12
Execute	EX	44	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTION POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OF	OTHER	00	01	10	11
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.Menmonic							N	N	N	N
Branch if Zeros	RX,Ext.Mnemonic							N	N	N	N
Branch on Zero	RX,Ext.Menmonic							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 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 Immediate	SI	X						Z	AA	BB	
Convert to Binary	RX	X	X		X		DataF	N	N	N	N
Convert to Decimal	RX	X	X		X			N	N	N	N
Divide	RX	X	X				F	N	N	N	N
Divide	RR		X				F	N	N	N	N
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)			

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
Insert Character	IC	43	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
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 Complement	LCR	13	R1,R2	
Load Halfword	LH	48	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Multiple	LM	98	R1,R3,D2(B2)	R1,R3,S2
Load Negative	LNR	11	R1,R2	
Load Positive	LPR	10	R1,R2	
Load PSW	LPSW	82	D1(B1)	S1
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 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 Halfword	MH	4C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
No Operation	NOP	47(BC 0)	D2(X2,B2) or D2(,B2)	S2(X2) or S2
No Operation	NOPR	07(BC 0)	R2	
Or Logical	O	56	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Or Logical	OC	D6	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Or Logical	OR	16	R1,R2	
Or Logical Immediate	OI	96	D1(B1),I2	S1,I2
Pack	PACK	F2	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Set Program Mask	SPM	04	R1	
Set Storage Key	SSK	08	R1,R2	
Set System Mask	SSM	80	D1(B1)	S1
Shift Left Double Algebraic	SLDA	8F	R1,D2(B2)	R1,S2

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTION POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OP	OTHER	00	01	10	11
Insert Character	RX	X						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 Complement	RR			F				P	L	M	O
Load Halfword	RX	X	X					N	N	N	N
Load Multiple	RS	X	X					N	N	N	N
Load Negative	RR							J	L		
Load Positive	RR			F				J		M	O
Load PSW	S1	X	X				A	QQ	QQ	QQ	QQ
Move Characters	SS	X			X			N	N	N	N
Move Immediate	S1	X			X			N	N	N	N
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 Halfword	RX	X	X					N	N	N	N
No Operation	RX,Ext.Mnemonic							N	N	N	N
No Operation	RR,Ext.Mnemonic							N	N	N	N
Or Logical	RX	X	X					J	K		
Or Logical	SS	X			X			J	K		
Or Logical	RR							J	K		
Or Logical Immediate	SI	X			X			J	K		
Pack	SS	X		X				N	N	N	N
Set Program Mask	RR		X					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

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
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	R1,R3,D2(B2)	R1,R3,S2
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 Halfword	STH	40	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store Multiple	STM	90	R1,R2,D2(B2)	R1,R2,S2
Subtract	S	5B	R1,D2(X2) or R1,D2(X2,B2)	R1,S2(X2) or R1,S2
Subtract	SR	1B	R1,R2	
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	
Add Double	AD	6A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add Double Register	ADR	2A	R1,R2	
Add Halfword Immediate	AHI	BA	R1,I2	
Add Short	AS	53	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add Short Immediate	ASI	AA	R1,I2	
Add Short Register	ASR	CA	R1,R3	
Branch Unconditional	BU	73	D2(X2,B2) or D2(,B2)	S2(X2) or S2
Branch Unconditional Register	BUR	CE	R2	

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTIONS POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OP	OTHER	00	01	10	11
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	RS	X	X					MM	EE	N	N
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 Multiple	RS	X	X		X			N	N	N	N
Subtract	RX	X	X	F				V	X	Y	O
Subtract	RR			F				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
Add Double	RX	X	X	F				Sum=0	Sum<0	Sum>0	Overflow
Add Double Register	RR		X	F				Sum=0	Sum<0	Sum>0	Overflow
Add Halfword Immediate	RI			F				Sum=0	Sum<0	Sum>0	Overflow
Add Short	RX	X	X	F				Sum=0	Sum<0	Sum>0	Overflow
Add Short Immediate	RI			F				Sum=0	Sum<0	Sum>0	Overflow
Add Short Register	RR			F				Sum=0	Sum<0	Sum>0	Overflow
Branch Unconditional	RX							N	N	N	N
Branch Unconditional Register	RR							N	N	N	N

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
Compare Double	CD	69	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare Double Register	CDR	29	R1,R2	
Compare Halfword Immediate	CHI	B9	R1,I2	
Compare Logical Short	CLS	65	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare Logical Short Immediate	CLSI	B5	R1,I2	
Compare Logical Short Register	CLSR	C5	R1,R2	
Compare Short	CS	61	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare Short Immediate	CSI	A9	R1,I2	
Compare Short Register	CSR	C9	R1,R2	
Divide Short	DS(X)	4D	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide Short Immediate	DSI	B0	R1,I2	
Divide Short Register	DSR	CD	R1,R2	
Load Address Short	LAS	51	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Complement Double Register	LCDR	23	R1,R2	
Load Complement Short Register	LCSR	C3	R1,R2	
Load Double	LD	68	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Double Register	LDR	28	R1,R2	
Load Full to Short Register	LFSR	0B	R1,R2	
Load Halfword Immediate	LHI	B8	R1,I2	
Load Halfword Register	LHR	D0	R1,R2	
Load Negative Short Register	LNSR	C1	R1,R2	
Load Positive Short Register	LPSR	C0	R1,R2	
Load Short	LS	74	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTIONS POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OP	OTHER	00	01	10	11
Compare Double	RX	X	X				Z	AA	BB		
Compare Double Register	RR		X					Z	AA	BB	
Compare Halfword Immediate	RI							Z	AA	BB	
Compare Logical Short	RX	X	X					Z	AA	BB	
Compare Logical Short Immediate	RI							Z	AA	BB	
Compare Logical Short Register	RR							Z	AA	BB	
Compare Short	RX	X	X					Z	AA	BB	
Compare Short Immediate	RI							Z	AA	BB	
Compare Short Register	RX							Z	AA	BB	
Divide Short	RX	X	X				F	N	N	N	N
Divide Short Immediate	RI							N	N	N	N
Divide Short Register	RR							N	N	N	N
Load Address Short	RX							N	N	N	N
Load Complement Double Register	RR		X	F				J	L	M	O
Load Complement Short Register	RR			F				J	L	M	O
Load Double	RX	X	X					N	N	N	N
Load Double Register	RR		X					N	N	N	N
Load Full to Short Register	RR			F				J	L	M	O
Load Halfword Immediate	RI										
Load Halfword Register	RR							N	N	N	N
Load Negative Short Register	RR							J	L		
Load Positive Short Register	RR			F				J		M	O
Load Short	RX	X	X					N	N	N	N

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
Load Short Immediate	LSI	A8	R1,I2	
Load Short Register	LSR	C8	R1,R2	
Load and Test	LT	62	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load and Test Short	LTS	52	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load and Test Short Register	LTSR	C2	R1,R2	
Multiple Halfword Immediate	MHI	BC	R1,R2	
Multiply Short	MS	71	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply Short Immediate	MSI	B3	R1,I2	
Multiply Short Register	MSR	CC	R1,R2	
Normalize	NRM	CF	R1,R2	
And Short	NS	64	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
And Short Immediate	NSI	B4	R1,I2	
And Short Register	NSR	C4	R1,R2	
Or Short	OS	66	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Or Short Immediate	OSI	A6	R1,I2	
Or Short Register	OSR	C6	R1,R2	
Subtract Double	SD	6B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Double Register	SDR	2B	R1,R2	
Subtract Halfword Immediate	SHI	BB	R1,I2	
Shift Left Arithmetic Short	SLAS	A3	R1,D2(B2)	R1,S2
Shift Left Logical Short	SLLS	A1	R1,D2(B2)	R1,S2
Shift Right Arithmetic Short	SRAS	A2	R1,D2(B2)	R1,S2
Shift Right Logical Short	SRLS	A0	R1,D2(B2)	R1,S2
Subtract Sort	SS	72	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Short Immediate	SSI	AB	R1,I2	
Subtract Short Register	SSR	CB	R1,R2	

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTIONS POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OP	OTHER	00	01	10	11
Load Short Immediate	RI							N	N	N	N
Load Short Register	RR							N	N	N	N
Load and Test	RX	X	X					J	L	M	
Load and Test Short	RX							J	L	M	
Load and Test Short Register	RR							J	L	M	
Multiply Halfword Immediate	RI							N	N	N	N
Multiply Short	RX	X	X					N	N	N	N
Multiply Short Immediate	RI							N	N	N	N
Multiple Short Register	RR							N	N	N	N
Normalize	RR							J	L	M	
And Short	RX	X	X					J	K		
And Short Immediate	RI							J	K		
And Short Register	RR							J	K		
Or Short	RX	X	X					J	K		
Or Short Immediate	RI							J	K		
Or Short Register	RR							J	K		
Subtract Double	RX	X	X	F				V	X	Y	O
Subtract Double Register	RR		X	F				V	X	Y	O
Subtract Halfword Immediate	RI			F				V	X	Y	O
Shift Left Arithmetic Short	RS			F				J	L	M	O
Shift Left Logical Short	RS							N	N	N	N
Shift Right Arithmetic Short	RS							J	L	M	
Shift Right Logical Short	RS							N	N	N	N
Subtract Short	RX	X	X	F				V	X	Y	O
Subtract Short Immediate	RI							V	X	Y	O
Subtract Short Register	RX							V	X	Y	O

Instruction	Mnemonic Code Code	Machine Code Code	Operand Format	
			Explicit	Implicit
Store Double	STD	60	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Exclusive or Short	XS	63	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Exclusive or Short Immediate	XSI	A7	R1,I2	
Exclusive or Short Register	XSR	C7	R1,R2	
Test Bits	TB	75	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Test Bits Immediate	TBI	AE	R1,I2	
Supervisor Call	SVC	0A	I	
Test and Set	TS	93	D1(B1)	S1
Test Under Mask	TM	91	D1(B1),I2	S1,I2
Timer Read and Set	TMRS	A4	R1,R3,D2(B2)	R1,R3,S2
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

INSTRUCTION	TYPE OF INSTRUCTION	PROGRAM INTERRUPTIONS POSSIBLE						CONDITION CODE SET			
		A	S	OV	P	OP	OTHER	00	01	10	11
Store Double	RX	X	X		X			N	N	N	N
Exclusive Or Short	RX	X	X					J	K		
Exclusive Or Short Immediate	RI							J	K		
Exclusive Or Short Register	RX							J	K		
Test Bits	RX	X	X					UU	VV		W
Test Bits Immediate	RI							UU	VV		W
Supervisor Call	RR							N	N	N	N
Test and Set	SI	X			X			SS	TT		
Test Under Mask	SI	X						UU	VV		WW
Timer Read and Set	RS	X	X					N	N	N	N
Translate	SS	X			X			N	N	N	N
Translate and Test	SS	X						PP	NN	OO	
Unpack	SS	X			X			N	N	N	N

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

Program Interruptions Possible (Continued)

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
AAO	Segment-or Page-Table Length Violation
AAP	Page-Table Entry Invalid (1-Bit One)
AAQ	Reference Bit Zero, Change Bit Zero
AAR	Reference Bit Zero, Change Bit One
AAS	Reference Bit One, Change Bit Zero
AAT	Reference Bit One, Change Bit One
AAU	Segment Table Entry Invalid (1-Bit One)
AAV	Translation Available

RR Format

OPERATION CODE	NAME	MNEMONIC	REMARKS
00			
01			
02			
03			
04	Set Program Mask	SPM	
05	Branch and Link	BALM	
06	Branch on Count	BCTR	
07	Branch on Condition	BCR	
08	Set Storage Key	SSK	
09			
0A	Supervisor Call	SVC	
0B	Load Full to Short Register	LFSR	
0C			
0E			
0F			
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			
21			
22			
23	Load Complement Double Register	LCDR	
24			
25			

RR Format			
OPERATION CODE	NAME	MNEMONIC	REMARKS
26			
27			
28	Load Double Register	LDR	
29	Compare Double Register	CDR	
2A	Add Double Register	ADR	
2B	Subtract Double Register	SDR	
2C			
2D			
2E			
2F			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
3A			
3B			
3C			
3D			
3E			
3F			
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	

RX Format

OPERATION CODE	NAME	MNEMONIC	REMARKS
49	Compare Halfword	CH	
4A	Add Halfword	AH	
4B	Subtract Halfword	SH	
4C	Multiply Halfword	MH	
4D	Divide Short	DSX	
4E	Convert to Decimal	CVD	
4F	Convert to Binary	CVB	
50	Store	ST	
51	Load Address Short	LAS	
52	Load and Test Short	LTS	
53	Add Short	AS	
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 Double	SD	
61	Compare Short	CS	
62	Load and Test	LT	
63	Exclusive Or Short	XS	
64	And Short	NS	
65	Compare Logical Short	CLS	
66	Or Short	OS	
67			
68	Load Double	LD	
69	Compare Double	CD	
6A	Add Double	AD	
6B	Subtract Double	SD	
6C			
6D			
6E			
6F			

RX Format			
OPERATION CODE	NAME	MNEMONIC	REMARKS
70			
71	Multiply Short	MS	
72	Subtract Short	SS	
73(X)	Branch Unconditional	BU	
74	Load Short	LS	
75	Test Bits	TB	
76			
77			
78			
79			
7A			
7B			
7C			
7D			
7E			
7F			
RS, SI, and S Format			
80	Set System Mask	SSM	
81			
82	Load PSW	LPSW	
83	Diagnose		
84			
85			
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	

RS, SI, and S Format

OPERATION CODE	NAME	MNEMONIC	REMARKS
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	SIO	
9D			
9E			
9F			
A0	Shift Right Logical Short	SRLS	
A1	Shift Left Logical Short	SLLS	
A2	Shift Right Arithmetic Short	SRAS	
A3	Shift Left Arithmetic Short	SLAS	
A4	Timer Read and Set	TMRS	HTC Only
A5			
A6	OR Short Immediate	OSI	
A7	Exclusive or Short Immediate	XSI	
A8	Load Short Immediate	LSI	
A9	Compare Short Immediate	CSI	
AA	Add Short Immediate	ASI	
AB	Subtract Short Immediate	SSI	
AC			
AD			
AE	Test Bits Immediate	TBI	
AF			
B0	Divide Short Immediate	DSI	
B1			
B2			
B3	Multiply Short Immediate	MSI	
B4	And Short Immediate	ASI	
B5	Compare Logical Short Immediate	CLSI	
B6			
B7			
B8	Load Halfword Immediate	LHI	
B9	Compare Halfword Immediate	CHI	
BA	Add Halfword Immediate	AHI	

RS, SI, and S Format			
OPERATION CODE	NAME	MNEMONIC	REMARKS
BB	Subtract Halfword Immediate	SHI	
BC	Multiply Halfword Immediate	MHI	
BD			
BE			
BF			
SS Format			
C0	Load Positive Short Register	LPSR	
C1	Load Negative Short Register	LNSR	
C2	Load and Test Short Register	LTSR	
C3	Load Complement Short Register	LCSR	
C4	And Short Register	ASR	
C5(RR)	Compare Logical Short Register	CLSR	
C6	Or Short Register	OSR	
C7	Exclusive or Short Register	XSR	
C8	Load Short Register	LSR	
C9	Compare Short Register	CSR	
CA(RR)	Add Short Register	ASR	
CB	Subtract Short Register	SSR	
CC	Multiply Short Register	MSR	
CD	Divide Short Register	DSR	
CE(R)	Branch Unconditional Register	BUR	
CF	Normalize	NRM	
D0	Load Halfword Register	LHR	
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	
DD	Translate and Test	TRT	
DE			

SS Format

OPERATION CODE	NAME	MNEMONIC	REMARKS
DF			
E0			
E1			
E2			
E3			
E4			
E5			
E6			
E7			
E8			
E9			
EA			
EB			
EC			
ED			
EE			
EF			
F0			
F1	Move with Offset	MVO	
F2	Pack	PACK	
F3	Unpack	UNPK	
F4			
F5			
F6			
F7			
F8			
F9			
FA			
FB			
FC			
FD			
FE			
FF			

APPENDIX E: ASSEMBLER INSTRUCTIONS

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	Will be taken as a remark
CNOP	A sequence symbol or not present	Two absolute expressions, separated by a comma
COM	A sequence symbol or not present	Will be taken as a remark
COPY	Must not be present	A symbol
CSECT	Any symbol or not present	Will be taken as a remark
CXD*	Any symbol or not present	Will be taken as a remark
DC	Any symbol or not present	One or more operands, separated by commas
DROP	A sequence symbol or not present	One to 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	Will be taken as a remark
DXD*	A symbol	One or more operands, separated by commas
EJECT	A sequence symbol or not present	Will be taken as a remark

Operation	Name Entry	Operand Entry
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

[2] SET symbols may be defined as subscripted SET symbols.

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	Will be taken as a remark
MACRO[1]	Must not be present	Will be taken as a remark
MEND[1]	A sequence symbol or nor present	Will be taken as a remark
MEXIT[1]	A sequence symbol or not present	Will be taken as a remark
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

[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.

Operation Entry	Name Entry	Operand Entry
PUNCH	A sequence symbol or not present	One to eighty characters enclosed in apostrophes
REPRO	A sequence symbol or not present	Will be taken as a remark
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

[3]See Section 5 for the description of the name entry.
*Assembler F only.

ASSEMBLER STATEMENTS

INSTRUCTIONS	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.

[5] When the name field of a macro instruction contains a sequence symbol, the sequence symbol is not passed as a name field parameter. It only has meaning as a possible branch target for conditional assembly.

APPENDIX F: SUMMARY OF CONSTANTS

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

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
S	2	half word	2 only	One absolute or relocatable expression or two absolute expressions: exp (exp)	multi-ple			
Y	2	half word	.1 to 2 (2)	any expression	multi-ple			left
I	as needed	byte	.1 to 256 (1)		one			right
W	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.

APPENDIX G: MACRO LANGUAGE SUMMARY

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[1]
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 samller 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.

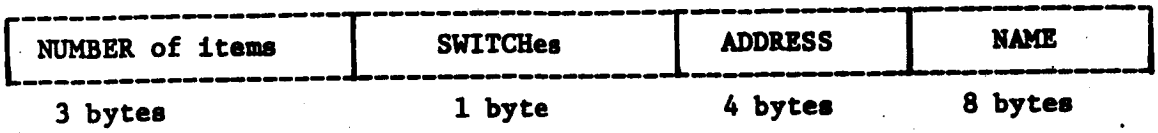
Chart 4. Variable Symbols

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.				

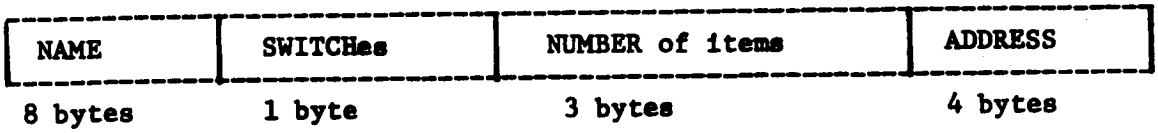
APPENDIX H: SAMPLE PROGRAM

Given:

1. A TABLE with 15 entries, each 16 bytes long, having the following format:



2. A LIST of items, each 16 bytes long, having the following format:



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                                                                  SAMPL001
         PRINT DATA                                                             SAMPL002
*                                                                               SAMPL003
*        THIS IS THE MACRO DEFINITION                                           SAMPL004
*                                                                               SAMPL005
         MACRO                                                                  SAMPL006
         MOVE &TO,&FROM                                                         SAMPL007
.*                                                                              SAMPL008
.*       DEFINE SETC SYMBOL                                                     SAMPL009
.*                                                                              SAMPL010
         LCLC &TYPE                                                             SAMPL011
.*                                                                              SAMPL012
.*       CHECK NUMBER OF OPERANDS                                               SAMPL013
.*                                                                              SAMPL014
         AIF      (N'&SYSLIST NE 2).ERROR1                                      SAMPL015
.*                                                                              SAMPL016
.*       CHECK TYPE ATTRIBUTES OF OPERANDS                                      SAMPL017
.*                                                                              SAMPL018
         AIF      (T'&TO NE T'&FROM).ERROR2                                     SAMPL019
         AIF      (T'&TO EQ 'C' OR T'&TO EQ 'G' OR T'&TO EQ 'K'),TYPECGK        SAMPL020
         AIF      (T'&TO EQ 'D' OR T'&TO EQ 'E' OR T'&TO EQ 'H'),TYPEDEH        SAMPL021
         AIF      (T'&TO EQ 'F').MOVE                                           SAMPL022
         AGO      .ERROR3                                                       SAMPL023
*TYPEDEH  ANOP                                                                  SAMPL024
```

```
.*                                                                      SAMPL025
.*          ASSIGN TYPE ATTRIBUTE TO SETC SYMBOL                        SAMPL026
.*                                                                      SAMPL027
&TYPE       SETC    T'&TO                                               SAMPL028
.MOVE       ANOP                                                        SAMPL029
*           NEXT TWO STATEMENTS GENERATED FOR MOVE MACRO                SAMPL030
            L&TYPE  2,&FROM                                             SAMPL031
            ST&TYPE 2,&TO                                               SAMPL032
            MEXIT                                                       SAMPL033
.*                                                                      SAMPL034
.*          CHECK LENGTH ATTRIBUTES OF OPERANDS                         SAMPL035
.*                                                                      SAMPL036
.TYPECGK    AIF     (L'&TO NE L'&FROM or L'&TO GT 256).ERROR4           SAMPL037
*           NEXT STATEMENT GENERATED FOR MOVE MACRO                     SAMPL038
            MVC     &TO,&FROM                                           SAMPL039
            MEXIT                                                       SAMPL040
.*                                                                      SAMPL041
.*          ERROR MESSAGES FOR INVALID MOVE MACRO INSTRUCTIONS          SAMPL042
 *                                                                      SAMPL043
.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 L,'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                                                    SAMPL123
*                                                                               SAMPL124
```

```
LISTAREA  DC      CL8'LAMBDA',X'0A',FL3'29'A(BEGIN)                SAMPL125
          DC      CL8'ZETA',X'05',FL3'5',A(LOOP)                   SAMPL126
          DC      CL8'THETA',X'02',FL3'45',A(BEGIN)                SAMPL127
          DC      CL8'TAU',X'00',FL3'0',A(1)                       SAMPL128
          DC      CL8'LIST',X'1F',FL3'465',A(0)                    SAMPL129
LISTEND   DC      CL8'ALPHA',X'00',FL3'1',A(123)                   SAMPL130
*                                                                  SAMPL131
*         THIS IS THE CONTROL TABLE                                SAMPL132
*                                                                  SAMPL133
          DS      0D                                               SAMPL134
TESTTABL  DC      FL3'1',X'00',A(123),CL8'ALPHA'                   SAMPL135
          DC      XL8'0',CL8'BETA'                                 SAMPL136
          DC      XL8'0',CL8'DELTA'                                SAMPL137
          DC      XL8'0',CL8'EPSILON'                              SAMPL138
          DC      XL8'0',CL8'ETA'                                  SAMPL139
          DC      XL8'0',CL8'GAMMA'                                SAMPL140
          DC      XL8'0',CL8'IOTA'                                 SAMPL141
          DC      XL8'0',CL8'KAPPA'                                SAMPL142
          DC      FL3'29',X'0A',A(BEGIN),CL8'LAMBDA'               SAMPL143
          DC      XL8'0',CL8'MU'                                   SAMPL144
          DC      XL8'0',CL8'NU'                                   SAMPL145
          DC      XL8'0',CL8'OMICRON'                              SAMPL146
          DC      XL8'0',CL8'PHI'                                  SAMPL147
          DC      XL8'0',CL8'SIGMA'                                SAMPL148
          DC      FL3'5',X'05',A(LOOP),CL8'ZETA'                   SAMPL149
*                                                                  SAMPL150
*         THIS IS THE CONTROL LIST                                 SAMPL151
*                                                                  SAMPL152
TESTLIST  DC      CL8'LAMBDA',X'0A',FL3'29',A(BEGIN)               SAMPL153
          DC      CL8'ZETA',X'05',FL3'5',A(LOOP)                   SAMPL154
          DC      CL8'THETA',X'82',FL3'45',A(BEGIN)                SAMPL155
          DC      CL8'TAU',X'80',FL3'0',A(1)                       SAMPL156
          DC      CL8'LIST',X'9F,FL3'465',A(0)                     SAMPL157
          DC      CL8'ALPHA',X'00',FL3'1',A(123)                   SAMPL158
*                                                                  SAMPL159
*         THESE ARE THE SYMBOLIC REGISTERS                         SAMPL160
*                                                                  SAMPL161
R0        EQU     0                                                SAMPL162
R1        EQU     1                                                SAMPL163
R2        EQU     2                                                SAMPL164
R3        EQU     3                                                SAMPL165
R5        EQU     5                                                SAMPL166
R6        EQU     6                                                SAMPL167
R7        EQU     7                                                SAMPL168
R12       EQU     12                                               SAMPL169
R13       EQU     13                                               SAMPL170
R14       EQU     14                                               SAMPL171
R15       EQU     15                                               SAMPL172
*                                                                  SAMPL173
*         THIS IS THE FORMAT DEFINITION OF LIST ENTRYS             SAMPL174
```

```
*                                                                SAMPL175
LIST     DSECT                                                   SAMPL176
LNAME    DS      CL8                                             SAMPL177
LSWITCH  DS      C                                               SAMPL178
LNUMBER  DS      FL3                                             SAMPL179
LADDRESS DS      F                                               SAMPL180
*                                                                SAMPL181
*        THIS IS THE FORMAT DEFINITION OF TABLE ENTRYS           SAMPL182
*                                                                SAMPL183
TABLE    DSECT                                                   SAMPL184
TNUMBER  DS      FL3                                             SAMPL185
TSWITCH  DS      C                                               SAMPL186
TADDRESS DS      F                                               SAMPL187
TNAME    DS      CL8                                             SAMPL188
         END     BEGIN                                           SAMPL189
```

APPENDIX I: ASSEMBLER LANGUAGES--FEATURES COMPARISON CHART

Features not shown 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
Level of parentheses	- -	- -	1	5	5

[1] Assembler F only

[2] DOS 14K D Assembler 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
Complex Relocatability	- -	- -	X	X	X
ASSEMBLER INSTRUCTIONS:					
DC and DS					
Expression allowed on 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
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 modifier	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

[1]Assembler F only

[2]DOS 14K D Assembler 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
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

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
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 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

APPENDIX J: SAMPLE MACRO DEFINITIONS

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                                                        00020000
       MACRO                                                            00040000
&NAME  SAVE   &REG,&CODE,&ID                                            00060000
       LCLA   &A,&B,&C                                                  00080000
       LCLC   &E,&F,&G,&H                                               00100000
       AIF    ('&REG' EQ '').E1                                         00120000
       AIF    ('&ID' EQ '').NULLID                                      00140000
       AIF    ('&ID' EQ '*').SPECID                                     00160000
&A     SETA   ((K'&ID+2)/2)*2+4                                         00180000
&NAME  B      &A.(0,15)          BRANCH AROUND ID                      00200000
&A     SETA   K'&ID                                                    00220000
       DC     AL1(&A)            LENGTH OF IDENTIFIER                  00240000
.CONTB AIF    (&A GT 32).SPLITUP                                       00260000
.CONTAA AIF   (&A GT 8).BRAKDWN                                        00280000
&E     SETC   '&ID'(&B+1,&A)                                           00300000
       DC     CL&A'&E'           IDENTIFIER                            00320000
       AGO    .CONTA                                                   00340000
.BRAKDWN ANOP                                                          00360000
&E     SETC   '&ID'(&B+1,8)                                            00380000
       DC     CL8'&E'            IDENTIFIER                            00400000
&B     SETA   &B+8                                                     00420000
&A     SETA   &A-8                                                     00440000
       AGO    .CONTAA                                                  00460000
.SPLITUP ANOP                                                          00480000
&E     SETC   '&ID'(&B+1,8)                                            00500000
&F     SETC   '&ID'(&B+9,8)                                            00520000
&G     SETC   '&ID'(&B+17,8)                                           00540000
&H     SETC   '&ID'(&B+25,8)                                           00560000
       DC     CL32'&E.&F.&G.&H'   IDENTIFIER                          00580000
&B     SETA   &B+32                                                    00600000
&A     SETA   &A-32                                                    00620000
       AGO    .CONTB                                                   00640000
.NULLID ANOP                                                           00660000
&NAME  DS     OH                                                       00680000
       AGO    .CONTA                                                   00700000
.SPECID AIF   ('&NAME' EQ '').CSECTN                                   00720000
&E     SETC   '&NAME'                                                  00740000
&A     SETA   1
```

```
.CONTQ   AIF    ('&E'(1,&A) EQ '&E').LEAVE                                00760000
&A       SETA   &A+1                                                      00780000
         AGO    .CONTQ                                                    00800000
.LEAVE   ANOP                                                             00820000
&B       SETA   ((&A+2)/2)*2+4                                            00840000
&NAME    B      &B.(0,15)                  BRANCH AROUND ID               00860000
         DC     AL1(&A)                                                   00880000
         DC     CL&A'&E'                   IDENTIFIER                     00900000
         AGO    .CONTA                                                    00920000
.CSECTN  AIF    ('&SYSECT' EQ '').E4                                      00940000
&E       SETC   '&SYSECT'                                                 00960000
&A       SETA   1                                                         00980000
         AGO    .CONTQ                                                    01000000
.E4      IHBERMAC 78,360                   CSECT NAME NULL                01020000
.CONTA   AIF    (T'&REG(1) NE 'N').E3                                     01040000
         AIF    ('&CODE' EQ 'T').CONTC                                    01060000
         AIF    ('&CODE' NE '').E2                                        01080000
&A       SETA   &REG(1)*4+20                                              01100000
         AIF    (&A LE 75).CONTD                                          01120000
&A       SETA   &A-64                                                     01140000
.CONTD   AIF    (N'&REG NE 2).CONTE                                       01160000
         STM    &REG(1),&REG(2),&A,(13)    SAVE REGISTERS                 01180000
         MEXIT                                                            01200000
.CONTE   AIF    (N'&REG NE 1).E3                                          01220000
         ST     &REG(1),&A.(13,0)          SAVE REGISTER                  01240000
         MEXIT                                                            01260000
.CONTC   AIF    (&REG(1) GE 14 OR &REG(1) LE 2).CONTF                     01280000
         STM    14,15,12(13)               SAVE REGISTERS                 01300000
&A       SETA   &REG(1)*4+20                                              01320000
         AIF    (N'&REG NE 2).CONTG                                       01340000
         STM    &REG(1),&REG(2),&A.(13)    SAVE REGISTERS                 01360000
         MEXIT                                                            01380000
.CONTG   AIF    (N'&REG NE 1).E3                                          01400000
         ST     &REG(1),&A.(13,0)          SAVE REGISTERS                 01420000
         MEXIT                                                            01440000
.CONTF   AIF    (N'&REG NE 2),CONTH                                       01460000
         STM    14,&REG(2),12(13)          SAVE REGISTERS                 01480000
         MEXIT                                                            01500000
.CONTH   AIF    (N'&REG NE 1).E3                                          01520000
         STM    14,&REG(1),12(13)          SAVE REGISTERS                 01540000
         MEXIT                                                            01560000
.E1      IHBERMAC 18,360                   REG PARAM MISSING              01580000
         MEXIT                                                            01600000
.E2      IHBERMAC 37,360,&CODE             INVALID CODE SPECIFIED         01620000
         MEXIT                                                            01640000
.E3      IHBERMAC 36,360,&REG              INVALID REGS. SPECIFIED        01660000
         MEND                                                             01680000
END OF DATA FOR SDS OR MEMBER
```

```
*
*                    SAMPLE SAVE MACRO INSTRUCTIONS
*

FOGHORN     SAVE        (14,12)
FORHORN     DS          OH
            STM         14,12,12(13) SAVE REGISTERS
                                     ***********

            SAVE        (REG14,REG12),T
            DS          OH
                        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                                                          00020000
&NAME       NOTE        &DCB                                               00040000
            AIF         ('&DCB' EQ '').ERR                                 00060000
&NAME       IHBINNRA &DCB                                                  00080000
            L           15,84(0,1)                LOAD VOTE RIN ADDRESS    00100000
            BALR        14,15                     LINK TO NOTE ROUTINE     00120000
            MEXIT                                                          00140000
.ERR        IHBERMAC 6                                                     00160000
            MEND                                                           00180000

MEMBER NAME POINT
            MACRO                                                          00020000
&NAME       POINT &DCB,&LOC                                                00040000
            AIF         ('&DCB' EQ '').ERR1                                00060000
            AIF         ('&LOC' EQ '').ERR2                                00080000
&NAME       IHBINNRA & DCB,&LOC                                            00100000
            L           15,84(0,1)                LOAD POINT RTN ADDRESS   00120000
            BAL         14,4(15,0)                LINK TO POINT ROUTINE    00140000
            MEXIT                                                          00160000
.ERR1       IHBERMAC 6                                                     00180000
            MEXIT                                                          00200000
.ERR2       IHBERMAC 3                                                     00220000
            MEND                                                           00240000
```

```
MEMBER NAME CHECK
            MACRO                                                    00020000
&NAME       CHECK &DECB                                              00040000
            AIF         ('&DECB' EQ '').E1                           00060000
&NAME       IHBINNRA  &DECB                                          00080000
            L           14,8(0,1)          PICK UP DCB ADDRESS       00100000
            L           15,52(0,14)        LOAD CHECK ROUT. ADDR.    00120000
            BALR        14,15              LINK TO CHECK ROUTINE     00140000
            MEXIT                                                    00160000
.E1         IHBERMAC 07,018                                          00180000
            MEND                                                     00200000
```