MRX/OS Assembler

Reference Manual

2202.001-01

MEMOREX

Computer System Products

December 1972 Edition

This edition (2202.001-01) is a major revision of, and obsoletes, the previous preliminary edition (2202.001).

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

A reader's comment form is provided at the back of this publication. If the form has been removed, comments may be addressed to the Memorex Corporation, Publications Dept., 8941 – 10th Ave. No. (Golden Valley) Minneapolis, Minnesota 55427.

© 1972, MEMOREX CORPORATION

PREFACE

This reference publication is intended for programmers using the MRX/OS Assembler Language. This publication describes how to write assembler source statements, including assembler instructions, macro instructions, and conditional assembly statements. These instructions are summarized in Appendix E. Machine instructions and extended mnemonic codes are summarized in Section 3, and additional reference tables appear in Appendixes B, C, and D. The machine instructions are described in detail in the publication **7200 or 7300 Processing Unit Reference.**

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
	Function of the Assembler	1-1
	Relationship to the Operating System	1-2
	System Requirements	1-2
2	WRITING SOURCE STATEMENTS	2-1
	Character Set	2-1
	Basic Format of Source Statements	2-3
	Terms and Expressions	2-3
	Terms	2-4
	Constants	2-5
	String Constants	2-5
	Character String Constant	2-5
	Hexadecimal String Constant	2-6
	Packed Decimal String Constant	2-6
	Zoned Decimal String Constant	2-6
	Integer String Constant	2-6
	Floating Point String Constant	2-6a
	Arithmetic Constants	2-7
	Decimal Arithmetic Constant	2-7
	Hexadecimal Arithmetic Constant	2-8
	Symbols	2-8
	Ordinary Symbols	2-9
	Variable Symbols	2-10
	Concatenation of Variable Symbols	2-10
	Sequence Symbols	2-11
	Location Counter Reference	2-11
	Symbol Length Attribute	2-12
	Literals	2-13
	Expressions	2-14
	Evaluation of Expressions	2-15
	Absolute and Relocatable Expressions	2-17
	Absolute Expressions	2-17
	Relocatable Expressions	2-18
	Examples of Absolute and Relocatable Expressions	2-18
	Coding Form	2-18
	Name Field	2-20
	Operation Field	2-20
	Operand Field	2-20
	Comment Field	2-20
	Identification-Sequence Field	2-20
	Statement Continuation	2-21

TABLE OF CONTENTS (Continued)

Section		Page
3	MACHINE INSTRUCTIONS	3-1
	Source Statements	3-1
	Instruction Alignment and Checking	3-1
	Operands and Suboperands	3-1
	Name and Length Attributes	3-2
	Notation Used to Describe Machine Instructions	3-2
	Summary of Machine Instructions	3-3
	General-Purpose Instructions	3-5
	System Instructions	3-9
	Summary of Extended Mnemonics	3-10
	Extended Mnemonic Codes	3-11
4	ASSEMBLER INSTRUCTION SOURCE STATEMENTS – OVERVIEW	4-1
5	PROGRAM SECTIONING AND LINKING STATEMENTS	5-1
	CSECT – Identify Control Section	5-1
	Symbolic Linkage Statements – ENTRY and EXTRN	5-2
	ENTRY – Identify ENTRY Point SYMBOL	5-2
	EXTRN – Identify External Symbol	5-3
	COM – Define Common Control Section	5-4
	Reserved Symbolic Segment Name – $SYSEG	5-6
6	PROGRAM CONTROL STATEMENTS	6-1
	ORG – Set Location Counter	6-1
	END – End Assembly	6-3
	PUNCH – Write to File	6-3
	LTORG – Begin Literal Pool	6-4
	ICTL – Input Format Control	6-4
	ISEQ – Input Sequence Checking	6-5
	ALIGN – Align Location Counter	6-6
7	LINKAGE-EDITOR MAP DIRECTIVE – SEG	7-1
8	SYMBOL AND DATA DEFINITION STATEMENTS	8-1
	EQU – Equate	8-1
	WDD and BDD – Word and Byte Defined Data	8-2
	WRS and BRS – Word and Byte Reserve Storage	8-5
	FORM – Define Data Format	8-7
	FORM – Instruction Statement	8-7
	Padding and Truncation Rules for Form Statements	8-8

TABLE OF CONTENTS (Continued)

Section		Page
9	LISTING CONTROL STATEMENTS	9-1
	TITLE – Identify Listing	9-1
	EJECT – Start New Page	9-2
	SPACE – Insert Blank Lines	9-2
	PRINT – Print Optional Data	9-3
10	MACRO LANGUAGE AND CONDITIONAL ASSEMBLY STATEMENTS	10-1
	Macro Language	10-1
	Macro Definition	10-1
	Header Statement	10-2
	Prototype Statement	10-2
	Model Statements	10-3
	Termination Statement	10-5
	Macro Instruction	10-5
	Positional Operands	10-6
	Keyword Operands	10-6
	Special Characters in a Macro Instruction	10-7
	Escape Character	10-7
	Ampersand	10-7
	Apostrophe	10-8
	Parentheses	10-8
	Comma	10-8
	Semicolon	10-8
	Blank	10-9
	Sublists in Macro Instructions	10-9
	Sublists in Model Statements	10-9
	Substring Notation	10-10
	Concatenation of Variable Symbols	10-11
	Nesting of Macros	10-12
	MNOTE – Generate Error Message	10-12
	MEXIT – Alternate Termination for Macro Definition	10-13
	System Variable Symbols – &SYSNDX and &SYSECT	10-13
	&SYSNDX	10-13
	&SYSECT	10-15
	Conditional Assembly Statements	10-15
	Set Statements	10-16
	SETA – Assign Arithmetic Value to Set Symbol	10-16
	SETC – Assign Character Value to Set Symbol	10-18
	GBLA and GBLC – Global Arithmetic and Character Set Symbols	10-20
	ADO – Iterative Return	10-20

TABLE OF CONTENTS (Continued)

Section		Page
10 (cont)	Nesting of ADO Statements	10-22
	AGO – Unconditional Branch	10-23
	ANOP – Label Definition	10-23
	Count and Number Attributes	10-24
	Count Attribute	10-24
	Number Attribute	10-25
11	CONTROL LANGUAGE STATEMENTS	11-1
APPENDIX A	– EBCDIC REPRESENTATION	A-1
APPENDIX B	– OBJECT FORMATS OF MACHINE INSTRUCTIONS	B-1
APPENDIX C	– ALPHABETICAL LIST OF MNEMONICS	C-1
APPENDIX D	– HEX CODE TO MNEMONIC CODE	D-1
APPENDIX E	– SUMMARY OF ASSEMBLER STATEMENTS	E-1
APPENDIX F	– MACRO EXAMPLE	F-1
APPENDIX G	– ASSEMBLER ERROR MESSAGES	G-1

LIST OF FIGURES

Figure		Page
2-1	Character Usage	2-2
2-2	Source Statement Format	2-3
2-3	Types of Terms	2-4
2-4	Character Constants	2-5
2-5	Truncation and Padding of String Constant Values	2-7
2-6	Examples of Assembled Constants	2-8
2-7	Concatenation of Variable Symbols	2-10
2-8	Examples of Length Attributes	2-12
2-9	Examples of Literals	2-13
2-10	Examples of Duplicate Literals	2-14
2-11	Types of Operators	2-16
2-12	Source Code Form	2-19
5-1	Example of EXTRN and ENTRY	5-3
5-2	Example of the COM Statement	5-5
8-1	Examples of EQU Statements	8-2
8-2	Examples of WDD and BDD Statements	8-3
8-3	Example of an ORG Statement with WDD and BRS	8-6
8-4	Examples of Padding and Truncation for Form Statements	8-9
10-1	Macro Definition	10-4
10-2	Macro Instruction – Positional Operands	10-6
10-3	Macro Instruction – Keyword Operands	10-7
10-4	Examples of Substring Notation	10-11
10-5	Concatenation of Variable Symbols	10-11
10-6	Nesting of Macros	10-12
10-7	Using &SYSNDX with Inner and Outer Macros	10-14
10-8	Examples of &SYSNDX	10-14
10-9	Example of &SYSECT	10-15
10-10	Example of the AGO Statement	10-24
10-11	Examples of the Count Attribute	10-25
11-1	Example of Control Language Statements	11-4
11-2	Example of Control Language Statements	11-5
11-3	Example of Control Language Statements	11-6
11-4	Placing Files on Disk – Example	11-6

1. INTRODUCTION

FUNCTION OF THE ASSEMBLER

The MRX/OS Assembler consists of a language and an assembler program. The language is a set of codes and coding rules for writing a source program. The assembler program translates the source program into an object program that can be executed by the system. The object program produced by the assembler is in the form of relocatable object modules. This translation process is called an assembly.

Two types of source statements can be expressed in the assembler language, *machine instructions* and *assembler instructions.*

The machine instruction source statements provide mnemonic codes for all machine instructions in the MRX 40/50 instruction set. Extended mnemonic codes are also provided for the skip and branch instructions. Section 3 of this manual describes the general format of the machine instructions. A complete description of the machine instructions, addressing techniques, and data representation is in the manual **7200 or 7300 Processing Unit Reference.**

The assembler instruction source statements specify auxiliary functions to be performed by the assembler program. These functions include:

- Checking and documenting programs
- Controlling address assignment
- Segmenting programs
- Defining data and symbols
- Generating macro and form instructions
- Controlling the assembly process through conditional assembly statements

The macro facility enables the programmer to define and use macro instructions. A macro instruction is represented by an operation code which, in turn, stands for a sequence of statements that accomplish the desired function.

Conditional assembly statements affect the order of source statement assembly and macro generation, or the content of generated statements.

A listing of the source program statements and the resulting object program statements may be produced with programmer control of form and content. A cross-reference list of symbol definitions and references is also produced unless suppressed by the programmer. Errors detected during assembly are indicated in the program listing. Warning errors may be suppressed.

RELATIONSHIP TO THE OPERATING SYSTEM

The assembler program is a component of the MRX 40/50 operating system and operates under its control. The operating system provides the assembler program with input/output, segment loading, library, and other services needed for its proper functioning. The assembler program is called through Control Language statements and resides in a user partition during execution.

SYSTEM REQUIREMENTS

The MRX 40/50 System equipment configuration required to execute the assembler program is as follows.

- 16K bytes of main storage, of which at least 8K bytes must be available to the assembler (additional storage, up to 24K, will increase the performance of the assembler)
- One source input device or data set
- One list device or data set
- One operator console
- One 660 disc storage drive
- The standard instruction set

2. WRITING SOURCE STATEMENTS

To write source statements, the programmer should be familiar with the following topics:

- Character set
- Basic format of source statements
- Types of terms and expressions
- Coding form

CHARACTER SET

Source statements may contain the following characters:

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

The EBCDIC formats and card punch codes for these characters are listed in Appendix A. Any of the 256 punch combinations may appear inside a character constant, in comments, or in macro instruction operands. The meanings of these characters, and combinations of these characters, are explained in Figure 2-1.

Character	Explanation	Example
A through Z, and $	Used in symbols and character string constants	C'ACCOUNT NO.'
0 through 9	Used in numeric constants and symbols	TAG3,5825
,	Operand or suboperand separator	HERE,THERE
=	Indicates a literal term or a keyword parameter value	=A+2
C'	Defines a character constant (all characters to the next apostrophe)	C'ABC'
X'	Defines a hexadecimal string constant (all hexadecimal characters to the next apostrophe)	X'1AFEE'
P'	Defines a packed decimal string constant (all characters to the next apostrophe)	P'425'
Z'	Defines a zoned decimal string constant (all characters to the next apostrophe)	Z'-44'
I'	Defines an integer string constant (all characters to the next apostrophe)	I'4286'
D'	Defines a floating point string constant (all characters to the next apostrophe)	D'147.25E-1'
"	Defines a hexadecimal arithmetic constant	"FF1A
< >	Define relational (EQ, GT, LT, NE, LE, GE) and logical (NOT, AND, OR, EOR) operations	A<EQ>B A<AND>B
L'	Defines a reference to a symbol length attribute	L'SYMX
*	Location counter reference or multiplication indicator	*+4 12*20
/	Division indicator (Note that 1/2=0 because division always results in an integer, not a fraction.)	10/0 TAG/B
+	Addition indicator	TAG+12
-	Subtraction indicator	TAB-4
&	Defines a variable symbol	&TAC
( )	Separates an address-modifying index from the rest of the address, delimits sublisted operands, or encloses operands or suboperands	PAG(R2)
.	Used for sequence symbols and concatenation	.LAST
.*	Used for macro definition comments	.*COMMENT
#	The character following this symbol is to be evaluated for its literal value, not for its special function. In the example, the symbol following the # sign is a semicolon, not a continuation indicator.	C'24#;4'
;	Continuation Indicator	THE STATEMENT IS;
@	Indirect addressing	@REG1,@TAG1
blank	Field separator	ADDR 3,4

Figure 2-1. Character Usage

BASIC FORMAT OF SOURCE STATEMENTS

Source program statements have the fields outlined in Figure 2-2.

Name	Operation	Operand	Comment
Any symbol or blank	Machine instruction, assembler instruction, macro instruction, or FORM instruction	Single expression, several expressions, or blank	Informational material or blank

Figure 2-2. Source Statement Format

The name field entry is a symbol used to identify a statement. The name field is necessary for certain statements, or when the statement is referred to in another statement, such as in a Branch instruction.

The operation field entry is a predefined mnemonic code (or mnemonic) which identifies the function of a machine, macro, assembler, or FORM instruction. Mnemonics are designed to be easily learned and remembered; for example, ADDR for Add Register-Register, or EQU for an Equate assembler instruction.

The operand field entry defines or identifies the data involved in the operation. Most statements have one or more operands, although some statements have no operands at all. Each operand has one or more terms, which may be used in a combination to form one or more expressions. (Refer to immediately following text for a discussion of terms and expressions.) An operand field may not have more than 35 terms. Operands of machine statements generally represent storage locations, general registers, immediate data, or constant values. Operands of assembler statements provide the information necessary for the assembler to perform the designated operation.

The optional comment field contains any informational material the programmer wishes to add.

TERMS AND EXPRESSIONS

A term is a symbol, character, or number that represents a value; an expression is a single term or a combination of terms. An expression is used in the operand field of a source statement. The following text fully defines terms and expressions.

TERMS

Every term represents a value; the value may be assigned by the assembler program (symbol, symbol length attribute, location counter reference) or may be inherent in the term itself (constant, literal).

An arithmetic combination of terms is reduced to a single arithmetic value by the assembler. An arithmetic value is represented as a 16-bit binary value in two's complement form. A logical value has a range of 0 through 65,535; and an arithmetic value has a range of -32,768 through 32,767. Limitations on the value of an expression depend on its use. For example, a term designating a general register must have a value between 0-7 inclusively; a term representing an address must not exceed the size of storage.

A term is absolute if its value does not change upon program relocation. It is relocatable if its value changes upon program relocation.

The terms used in assembler statements are outlined in Figure 2-3. An explanation of each type of term and the rules for its use are provided in the following text.

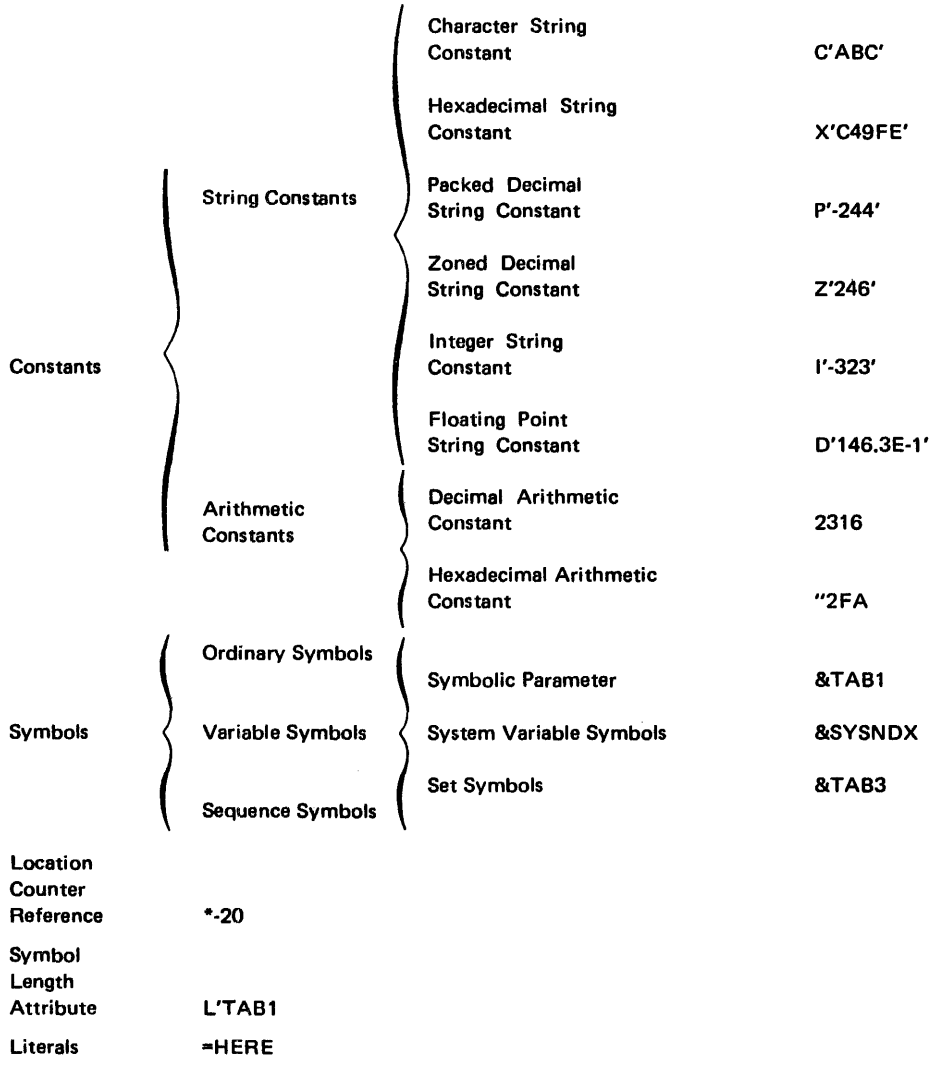


Figure 2-3. Types of Terms

CONSTANTS

Constants are terms whose values are inherent in the terms themselves. They specify machine values or bit configurations directly, rather than by equating the values to symbols and then using symbolic references. Constants represent such program elements as immediate data, masks, registers, addresses, and address increments.

Constants are string or arithmetic. String constants are of variable size; arithmetic constants are 16 bits long. Examples of all types of constants are presented in Figure 2-6.

String Constants

A string constant can only be used as a single term expression, or in a relational expression. In a relational expression, both terms must be of the same type (character, hexadecimal, etc.), for example: C'ABCE'< LT>C'&P1'.

Character String Constant

A character string constant is written as the letter C followed by a character string enclosed in apostrophes, for example: C'STRING'. To represent the literal value of an apostrophe, an ampersand, a semicolon, or a pound sign as part of the character constant, the character must be immediately preceded by an escape character, which is the pound sign. The length of a character constant is equal to the number of characters in the constant, excluding the escape characters, which do not appear in storage.

Examples of character constants are shown in Figure 2-4. In the last example of Figure 2-4, the generated code is: THIS STRING HAS MANY SPECIAL CHARS IN IT: # ; ' &.

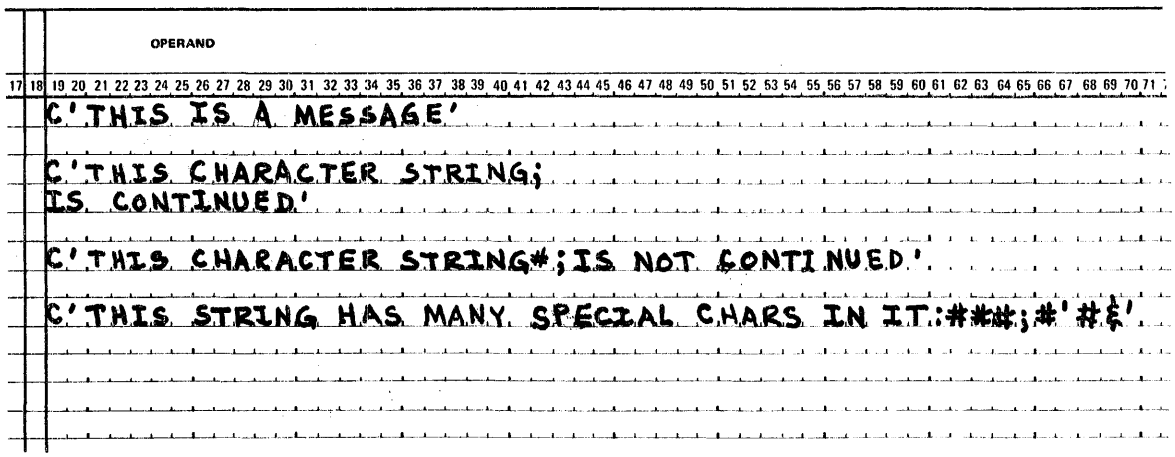


Figure 2-4. Character Constants

If the following characters are not preceded by an escape character in a character constant, they have the meaning given below:

Character	Meaning
;	A character constant is continued on the next line
'...'	Encloses the characters of a character constant
&	Variable symbol
#	Next character retains its literal value

Hexadecimal String Constant

A hexadecimal string constant is written as the letter X followed by a string of hexadecimal digits enclosed in apostrophes, such as: X'C49FE'. Each hexadecimal digit is translated into its four-bit equivalent. The maximum size of a hexadecimal string constant is limited to the maximum number of digits that can be contained on two coding lines. If an odd number of digits is specified, the leftmost four bits in the leftmost byte are set to zero. The implied length of the constant is half the number of hexadecimal digits in the constant, rounded to the next higher integer.

Packed Decimal String Constant

A packed decimal string constant is written as the letter P followed by a signed integer number enclosed in apostrophes, such as: P'-244'. If the sign is omitted, the number is assumed to be positive. Each pair of decimal digits is translated into one byte. The rightmost byte of a packed field contains the rightmost digit and the sign. Signs generated are "C_{16}" for plus, and "D_{16}" for minus. The maximum length of a packed decimal string constant is limited to the number of digits that can be contained on two coding lines.

Zoned Decimal String Constant

A zoned decimal string constant is written as the letter Z followed by a signed integer number enclosed in apostrophes, for example: Z'246'. If the sign is omitted, the number is assumed to be positive. Each decimal digit is translated into one byte. The rightmost byte contains the sign and the rightmost digit. Signs generated are "C_{16}" for plus, and "D_{16}" for minus. The maximum length of a zoned decimal string constant is limited to the number of digits that can be contained on two coding lines.

Integer String Constant

An integer string constant is written as the letter I followed by a signed integer number enclosed in apostrophes, such as: I'-246'. If the sign is omitted, the number is assumed to be positive. An integer string constant is translated into its four-byte binary equivalent. Integer constants consist of 1-10 digits with a value ranging from -2^{31} to $2^{31}-1$. The constant is word aligned when used in a WDD statement or a literal.

Floating Point String Constant

A floating point string constant is written as the letter D followed by a floating point number enclosed in apostrophes, such as: D'+2.461E-1'. The number is written as a signed or unsigned decimal value; if the sign is omitted, the number is assumed to be positive. The decimal point may appear before, within, or after the number, or it may be omitted. The exponent is optional. It is written immediately after the number in the form Enn, where nn is a signed or unsigned decimal integer specifying the exponent of the factor 10. If the exponent is unsigned, a plus sign is assumed.

Alignment can be specified by all data definition statements (WDD, BDD, and FORM). Byte alignment is allowed to facilitate building tables containing mixed values with odd bounding. Note, however, that all floating point *instructions* require word bounding of the operand in storage. The optional size and repetition factors may be coded.

The implicit length of a floating point constant is eight bytes, and it is carried as such through the assembly. Truncation or padding occurs at object output time with no rounding. The floating point decimal value may contain a maximum of 152 digits (including the exponent) in addition to the decimal point and sign. The approximate decimal range of values allowed is 7.2×10^{75} to 5.4×10^{-79}.

The internal format of floating point numbers is described in the **7300 Processing Unit Reference** manual.

When string constants define data in storage, truncation and padding of their values is performed according to the rules presented in Figure 2-5.

Constant	Explicit Length= Implicit Length	Explicit Length> Implicit Length	Explicit Length< Implicit Length
Character C'ABC'	C'ABC'(3)=ABC	Left justify. Blank fill on right. C'ABC'(4)=ABC	Left justify. Truncate on right. Warning message is given. C'ABC'(2)=AB
Hexadecimal X'10A'	Right justify. Zero fill on left if the constant contains an odd number of digits. X'10A'(2)=010A	Right justify. Zero fill on left. X'10A'(3)=00010A	Right justify. Truncate on left. Warning message is given. X'10A'(1)=0A
Packed Decimal P'-24'	Right justify. Zero fill on left if the constant contains even number of digits. P'-24'(2)=024D	Right justify. Zero fill on left. P'-24'(3)=00024D	Right justify. Truncate on left. Warning message is given. P'-24'(1)=4D
Zoned Decimal Z'123'	Z'123'(3)=F1F2C3	Right justify. Zero fill on left. Z'123'(4)=F0F1F2C3	Right justify. Truncate on left. Warning message is given. F'123'(2)=F2C3
Integer I'-758'	Right justify. Propagate sign on left. I'-758'(4)=FFFFFD0A	Right justify. Propagate sign on left I'-758'(6)= FFFFFFFFFD0A	Right justify. Truncate on left. Sign is lost. Warning message is given. I'-758'(1)=0A
Floating Point D'+1.3'	Normalize Size = 8 bytes D'+1.3'= 4114CCCCCCCCCCD	Expand fraction. Zero fill on right. D'+1.3(10)= 4114CCCCCCCCCCD0000	Truncate fractional position on right, no rounding. Warning message is given. D'+1.3(4)=4114CCCC

Figure 2-5. Truncation and Padding of String Constant Values

Arithmetic Constants

Arithmetic constants can be used in multi-term expressions. An arithmetic constant is assembled as its two-byte binary equivalent. The maximum size of an arithmetic constant is $2^{16}-1$. If arithmetic constants are used in statements where an explicit size is specified, truncation and padding follow the same rules as those for an integer string constant.

Decimal Arithmetic Constant

A decimal arithmetic constant is written as an unsigned integer number of 1-5 digits, for example: 20.

Hexadecimal Arithmetic Constant

A hexadecimal arithmetic constant is written as quotation marks followed by a string of 1-4 hexadecimal digits, for example: "2FA. Each hexadecimal digit is assembled as its four-bit binary equivalent.

Type	Example	Generated Hexadecimal Code
Character String	C'F12AY9*' C'$Z#'##5' C'B'	C6F1F2C1E8F95C 5BE97D7BF5 C2
Hexadecimal String	X'C49FE' X'F2' X'C'	0C49FE F2 0C
Packed Decimal String	P'14' P'925860' P'-2' P'-2596'	014C 0925860C 2D 02596D
Zoned Decimal String	Z'14' Z'925860' Z'-2' Z'-2596'	F1C4 F9F2F5F8F6C0 D2 F2F5F9D6
Integer String	I'14' I'925860' I'-2' I'-2596'	00 00 00 0E 00 0E 20 A4 FF FF FF FF FF FF F5 DC
Floating Point String	D'10' D'+99999999' D'-50.25E-1'	41 A0 00 00 00 00 00 00 49 25 40 BE 3F F0 00 00 C1 50 66 66 66 66 66 66
Decimal Arithmetic	14 302 57399	000E 012E E037
Hexadecimal Arithmetic	"14 "F2A "E09F	0014 0F2A E09F

Figure 2-6. Examples of Assembled Constants

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 operands, provide the programmer with an efficient way to name and reference a program element. A symbol is defined when it appears in the name field of a source statement.

In general, symbols must conform to these rules:

1. The symbol must not have more than eight characters.

2. The first character must be a letter, a period, a dollar sign, or an ampersand (&).

3. The remaining characters may be digits, letters, or dollar signs. If the first character is a period or an ampersand, the second character must be a letter or a dollar sign.

4. The first blank after the start of a symbol terminates that symbol.

5. Symbol definitions cannot be continued.

The assembler has three types of symbols: ordinary symbols, variable symbols, and sequence symbols. Sequence symbols and variable symbols are used only for the macro language and for conditional assembly.

Ordinary Symbols

An ordinary symbol consists of 1-8 alphanumeric characters, the first of which must be a letter or a dollar sign. Ordinary symbols identify program locations or arbitrary values. The value of an ordinary symbol may be absolute or relocatable. Examples of ordinary symbols are:

```
BETA

X242

$ENTRYP1
```

An ordinary symbol that names an instruction, a storage area, a data definition, or a control section is the address of the leftmost byte of the identified field. Address values are relocatable terms. The value of an address symbol must not exceed $2^{16}-1$.

An ordinary 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 more than once in the name field of a COM or CSECT assembler statement, because the coding of a control section may be suspended and then resumed at a subsequent point. Some statements require that a symbol in the operand field be previously defined.

During assembly, the assembler assigns a length attribute to all ordinary symbols. 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 four.

Variable Symbols

A variable symbol is a symbol that is assigned different values by the programmer or the assembler. The three types of variable symbols are:

1. Symbolic parameters — used only in macro definitions; values are assigned by macro instructions.

2. System variable symbols — used only in macro definitions; values are assigned by the assembler.

3. Set Symbols — used anywhere in the source program; values are assigned by SET or GBL statements.

Variable symbols consist of an ampersand (&) followed by one to seven alphanumeric characters, the first of which must be a letter or a dollar sign. Examples of variable symbols are:

&BETA

&X24

&P1

Concatenation of Variable Symbols

When a variable symbol is assembled, the current value assigned to the variable symbol is substituted for the variable symbol. If a variable symbol is immediately preceded or followed by other characters or by another variable symbol, concatenation of the variable symbol with another variable symbol or character occurs. To concatenate a variable symbol with a letter, digit, period, or left parenthesis that follows the symbol, a period must immediately follow the variable symbol, for example: &VAL.8. The period merely indicates the end of the variable symbol and does not appear in the generated code. The size of a concatenated symbol is limited only by the maximum statement size. However, the generated symbol is limited by the rules which pertain to the generated name, operation, or operand field. See Figure 2-7 for examples of the concatenation of variable symbols.

Assume that the following values have been assigned to these variable symbols:

```
&P1 = ROP
&P2 = 5
&P3 = @
```

Initial Code	Generated Code
&P1&P2	ROP5
&P1.8	ROP8
&P3.R7	@R7
B.&P2	B.5
&P1	ROP
703&P2	7035

Figure 2-7. Concatenation of Variable Symbols

Sequence Symbols

Sequence symbols consist of a period followed by one to seven alphanumeric characters, the first of which must be a letter or a dollar sign. Sequence symbols can be used in the name field of any statement except MACRO, GBLA, and GBLC, and in the operand field of only ADO or AGO statements. The programmer can use sequence symbols to vary the sequence in which statements are processed by the assembler. Examples of sequence symbols are:

.LAST

.HERE

LOCATION COUNTER REFERENCE

A location counter assigns 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 of the symbol is the value of the location counter before addition of the length.

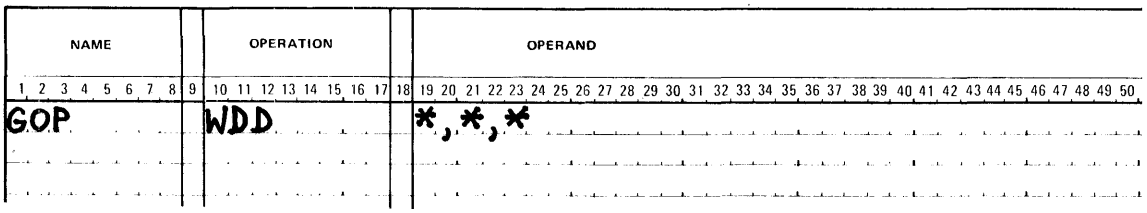
The assembler maintains a location counter for each control section of the program and manipulates each location counter as previously described. Source statements for each section are assigned addresses from the location counter for that section. The location counter for a given control section assigns locations in storage without regard to assignments made within other control sections.

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 is controlled by using the ORG and ALIGN assembler statements. The counter affected by an ORG statement is the counter for the control section in which it appears. The maximum value for the location counter is $2^{16}-1$.

The programmer can refer to the current location counter by using an asterisk in the operand field. The asterisk represents the value of the current location counter at the start of the current statement. This value is relocatable.

An example of the use of the location counter is:



If the location counter is at 0100 when this statement is encountered, the following data is generated:

Location	Value
0100	0100
0102	0100
0104	0100

SYMBOL LENGTH ATTRIBUTE

The length attribute of a symbol may be used as a term by writing L' followed by the symbol, for example: L'SYMX. The length attribute of SYMX is then substituted for the term. The length attribute of an ordinary symbol is the length, in bytes, of the storage field whose address is represented by the symbol.

The length attribute of * is invalid. If the operand of an EQU statement is an asterisk or an arithmetic constant, the symbol defined by the EQU statement has a length attribute of one. In any other context, the length attribute of an arithmetic constant is two. Examples of symbol length attributes are shown in Figure 2-8.

```
NAME      OPERATION   OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
NUM       WDD         64
* L'NUM=2  (ARITH CONSTANT)

NUM1      EQU         256*10
* L'NUM1=1 (ARITH CONSTANT IN EQU STATEMENT)

INST      LODD        25,R1
* L'INST=4 (LODD INSTRUCTION IS 4 BYTES LONG)

LOC       EQU         *
* L'LOC=1  (LOCATION CTR REF IN EQU STATEMENT)

CONS      EQU         C'THIS STRING'
* L'CONS=11 (ELEVEN CHARACTERS IN STRING)

INT       WDD         I'-26'
* L'INT=4  (INTEGER STRING IS 4 BYTES LONG)
```

Figure 2-8. Examples of Length Attributes

LITERALS

A literal term is used to introduce data into a program. The formats of a literal term are as follows.

Where:

=a	a	is the data value to be generated (required); any legal expression except another literal term.
=a(b,c) =a(b)	b	is the length specification (in bytes): a positive absolute expression. If omitted, the length specification is the implied size of the expression.
=a(,c)	c	is the repetition factor; a positive absolute expression. If omitted, a repetition of 1 is assumed. If the size or length is specified symbolically, the symbol must have been previously defined.

Examples of literals are shown in Figure 2-9.

==C'ABD'	Invalid: literal cannot define another literal.
=C'ABD'(4,3)	Valid: same as =C'ABD ABD ABD'
=A+B/2+4	Valid: implied length is the length of symbol A; implied repetition factor is 1.
=P'-446'(6)	Valid: specified length is 6; implied repetition factor is 1.
=X'FF00'(,3)	Valid: implied length is 2; specified repetition factor is 3.

Figure 2-9. Examples of Literals

The assembler generates the literal data, stores this data in a literal pool, and places the address of the stored data in the operand field of the statement using the literal. The position of the literal pool may be controlled by the programmer with the LTORG assembler statement. If LTORG is not specified at the end of a control section, the literal pool for that segment is placed at the end of the first control section.

A literal can be defined at any point in a program by specifying the literal in the operand of the statement in which it is used. In contrast, data definition statements define and label data, and then the label is used to specify the data.

A literal may not be combined with any other term, nor may a literal be used as a receiving field of a statement that modifies storage.

Literals are relocatable, because the address of the literal, not the literal itself, is assembled into the statement using the literal.

If duplicate literals are specified within one literal pool, only one literal is stored. Literals are duplicate if their final specifications, size, and repetition factors are identical on a character-by-character basis. A literal may be a duplicate even when it appears to be different (see examples in Figure 2-10). A literal is a duplicate if it contains no forward references and the expressions evaluate to the same value as the corresponding expressions of an existing literal.

A literal which contains a reference to the location counter is stored even if it duplicates another literal (see examples). If an expression used in a literal term contains a forward reference to a symbol, the symbol is assumed to represent a two-byte value.

Examples of duplicate literals are shown in Figure 2-10.

=C'ABC'(4,3) =C'ABC'(4,3)	Only one literal is stored.
=C'ABC' =X'C1C2C3'	Both literals are stored.
=A+B =B+A	Only one literal is stored if A and B are predefined symbols.
=C<EQ>D =1	Only one literal is stored if C is defined to be equal to D, so that the expression is equal to 1.
=*+10 =*+10	Both literals are stored

Figure 2-10. Examples of Duplicate Literals

EXPRESSIONS

An expression is defined as one or more terms linked by arithmetic, relational, or logical operators. Expressions may be single term or multi-term (see examples below).

Single Term Expressions	Multi-Term Expressions
29	SYMX+40
"F0	A+B/2+10
SYMX	(X<OR>"F0F0)<EQ>(SP2<OR>"F0F0)
*	(((A+4)/2+1)*2<AND>"00FF)<EQ>24)
L'SYMX	*+L'BETA
P'-240'	A+B<LE>SUM

During assembly, all expressions are resolved to a single value. Figure 2-11 provides an explanation of all types of operators.

The rules for coding expressions within an operand field are as follows.

1. An expression may not start with an arithmetic, relational, or logical operator. However, an expression may begin with a unary operator: positive sign (+), negative sign (-), or logical complement (< NOT >). A unary operator indicates the state of the numbers it precedes (such as negative, positive, or complement), rather than indicating an arithmetic operation (such as addition or subtraction).

2. An expression may not contain two terms in succession.

3. An expression may not contain two operators in succession, except for the logical operator < NOT >, which may follow the logical operators <AND>, <OR>, and <EOR>.

4. A multi-term expression may not contain a literal.

5. In a multi-term expression, string constants are restricted to relational operations.

EVALUATION OF EXPRESSIONS

A single term expression has the value of the term involved.

A multi-term expression is reduced to a single arithmetic value as follows.

1. Each term is given its value.

2. Operations are performed from left to right using the following rules of precedence:

   a. Unary arithmetic operations: positive (+) and negative (-).

   b. Arithmetic multiplication (*) and division (/).

   c. Arithmetic addition (+) and subtraction (-).

   d. Relational operations (<EQ>, <NE>, <LT>, <GT>, <LE>, and <GE>).

   e. Unary logical complement (<NOT>).

   f. Logical product (<AND>).

   g. Logical addition (<OR>) and subtraction (<EOR>).

3. The expression is computed to 32 bits, and then truncated to 16 bits or less, depending on its contextual use.

4. Division always yields an integer result. For example, 1/2*10 yields a zero result, whereas 10*1/2 yields 5. Division by zero is permitted and yields a zero result.

5. A relational operation yields a binary result of 0 or 1. If string constants are used in relational operations, both relational terms must be of the same type; thus, P'246'<EQ>Z'246' is illegal.

6. Logical operations are performed on a bit-by-bit basis equivalent to a masking operation. A non-zero value is considered true and a zero value is considered false.

Arithmetic Operators		
Operator	Meaning	Example
+	Addition	A+B
-	Subtraction	10-C
*	Multiplication	D*16
/	Division	25/X

Relational Operators		
Operator	Meaning	Example
<EQ>	Equal to	A<EQ>B
<NE>	Not equal to	A<NE>B
<LT>	Less than	17<LT>&P1
<GT>	Greater than	69<GT>TAB
<LE>	Less than or equal to	73<LE>M
<GE>	Greater than or equal to	"3F<GE>&TAB1

Logical Operators		
Operator	Meaning	Example
<NOT>	Logical complement (one's complement)	<NOT>A
<AND>	Logical product	A<AND>B
<OR>	Logical addition (inclusive or)	A<OR>B
<EOR>	Logical subtraction (exclusive or)	A<EOR>W

Figure 2-11. Types of Operators

Parentheses are used in the normal role of arithmetic grouping to change the order of evaluation. Parenthesized parts of an expression are evaluated before the rest of the terms in the expression. In the case of nested parentheses, the innermost parentheses are evaluated first. For example, the expression ((A+4)/2+1)*B is evaluated as follows, if A=10 and B=3.

1. A+4 = 14 giving (14/2+1)*B
2. 14/2 = 7 giving , (7+1)*B
3. 7+1 = 8 giving 8*B
4. 8*B = 24 giving 24

ABSOLUTE AND RELOCATABLE EXPRESSIONS

An expression is absolute if its value is unaffected by program relocation. It is relocatable if its value is changed by program relocation.

Absolute Expressions

An absolute expression may contain relocatable terms (RT) alone, or in combination with absolute terms (AT), provided the following conditions are met.

1. The relocatable terms must be paired or used in a relational operation. The terms in a pair must have opposite signs, but do not have to be contiguous, as in the example: RT+AT-RT. Each pair must be relocated to the same location counter.
2. A relocatable term or expression must not enter into a multiplication, division, or logical operation. For example: RT-RT*10 is invalid, while (RT-RT)*10 is valid.
3. The result of a relational operation is absolute regardless of the relocatability of the terms or expressions used in the operation.
4. Relocatable terms or expressions used in a relational operation are considered absolute. The relocatability attribute is disregarded. Thus RT<LT>RT is valid even if the two terms or expressions do not appear in the same control section.
5. If an expression that enters into a relational operation has multiple relocation attributes, an error indicator is given and the operation is performed as if the value of the expression were absolute, for example: RT<EQ>RT_1+RT_2.

Relocatable Expressions

A relocatable expression reduces to a single relocatable value. A relocatable expression may contain relocatable terms alone, or in combination with absolute terms, provided the following conditions are met.

1. All the relocatable terms but one must be paired, or be involved in a relational operation.

2. The leftover relocatable term must not be directly preceded by a minus sign.

3. No relocatable term may enter into a multiplication, division, or logical operation.

Examples of Absolute and Relocatable Expressions

The following examples illustrate absolute and relocatable expressions. A is an absolute term; BR1 and CR1 are relocatable terms within the current control section. XR2 is a relocatable term in a control section different from that in which BR1 and CR1 are defined. Examples of absolute and relocatable expressions are:

Absolute Expressions	Relocatable Expressions
A-BR1+CR1	A*A+BR1+XR2-CR1
A	BR1
BR1+A-CR1	BR1+CR1-*
BR1+(XR2<LT>CR1)+A-CR1	(BR1<EQ>XR2)*10+CR1

CODING FORM

Figure 2-12 illustrates a source code form provided for convenience to the programmer and the keypunch operator. Since assembler statements are free form, the various fields (name, operation, operand, and comment) need not begin in any specified column. The only restrictions are that the fields appear in the sequence shown, be separated by one or more spaces, and the name field begin in column 1. If the name field is omitted, the operation field can begin in any column after column 1.

All statements are contained in columns 1 through 72. Columns 73 through 80 are reserved for identification and statement sequencing. Thus, column 1 is called the begin column and column 72 the end column. The standard begin and end columns can, however, be altered by the ICTL assembler statement. (This statement is described in *Chapter 6. Program Control Statements*.)

MEMOREX

Assembler Coding Form

Punching Instructions

Graphic

Punch

Date_____ Page_____ of_____

Programmer_____

Program_____

NAME	OPERATION	OPERAND	IDENTIFICATION
1 2 3 4 5 6 7 8	9 10 11 12 13 14 15 16 17	18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

Figure 2-12. Source Code Form

NAME FIELD

If the name field is included, it must begin in the begin column (column 1). If the name field is omitted, the begin column must be blank. A name cannot be more than eight characters long after substitution, and cannot be continued on the next line.

OPERATION FIELD

The operation field begins after the first blank column. More than one blank column can separate the name field and the operation field; however, within the operation field, blanks are not allowed. The operation field after substitution cannot be more than eight characters. It cannot be continued.

OPERAND FIELD

Operands identify or describe data used in a statement. An operand may be actual data, a storage address, a register number, a field length, mask, etc. A comma must separate operands; embedded blanks or spaces are not allowed, except in strings.

COMMENT FIELD

To include comments on the same line as an existing statement, the programmer must begin the comment at least one space after the operand field. A comment cannot extend past the end column (column 72). An asterisk (*) or a period and asterisk (.*) starting in the begin column indicate a comment statement. The asterisk is used for commentary notes on the source program; the period and asterisk are used for comments in a macro definition. Comments do not affect the assembled program, but are printed in the assembly listing.

The following example illustrates the name, operation, operand, and comment fields. LOAD REGISTER on the first line is a comment because a space separates it from the last operand. The asterisk on the second line in column 1 specifies that all characters on that line are comments.

```
NAME      OPERATION      OPERAND
1 2 3 4 5 6 7 8 | 9 | 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
BEGIN           LOD            TAG,1 LOAD REGISTER
*                              1 FROM TAG
```

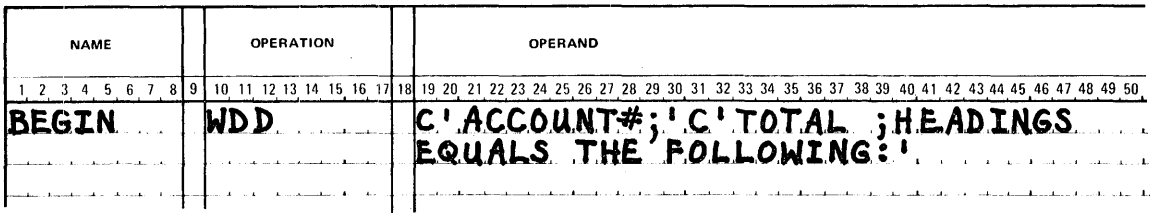
IDENTIFICATION-SEQUENCE FIELD

The identification-sequence field is used for identification or sequencing of statements. This optional field is normally contained in positions 73-80. The contents of the field appear in the source listing. The programmer may verify the statement sequence by the use of the ISEQ assembler statement (discussed in Chapter 6).

STATEMENT CONTINUATION

A semicolon (;) indicates that a statement is continued to the next line. The first non-blank character in the next line is the start of the continuation line. Therefore, a blank cannot be the first character of a continuation line. Only one continuation line is allowed for source statements, except for macro instructions, prototype statements, and GBL statements. Any characters after a semicolon are ignored by the assembler and considered as comments. A semicolon preceded by an escape character retains its literal value. The first non-blank character of a continuation line cannot be a semicolon; nor can a continuation line be entirely blank.

The operands in the following example are two character constants: ACCOUNT; and TOTAL EQUALS THE FOLLOWING. The first semicolon retains its literal value because it is preceded by an escape character. The second semicolon indicates to the assembler that the second operand is continued on the next line beginning with the first non-blank character. HEADINGS, which appears after the semicolon, is a comment. Note that the blank between TOTAL and EQUALS must be coded on the first line, not in the continuation line.



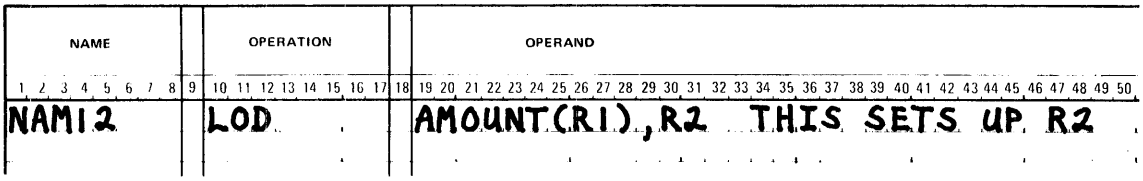
3. MACHINE INSTRUCTIONS

SOURCE STATEMENTS

Machine instruction source statements consist of:

- Name field (optional)
- Mnemonic operation code
- Operand field
- Comment field (optional)

An example of a machine instruction source statement is given below. (The data flow of most machine instructions is operand 1→operand 2.)



INSTRUCTION ALIGNMENT AND CHECKING

All machine instructions are aligned by the assembler on even-byte boundaries. The assembler advances the current location counter the amount necessary to ensure correct alignment of the assembled instructions. The contents of the area between the prealignment location counter and the postalignment location counter is unchanged. All expressions that specify storage addresses are checked to ensure appropriate alignment for the instruction format in which they are used.

OPERANDS AND SUBOPERANDS

Machine instructions have 0, 1, or 2 operands. Operands are written as a single operand, or as an operand with 1 or 2 suboperands. The possible formats of an operand are shown below.

op

op(subop)

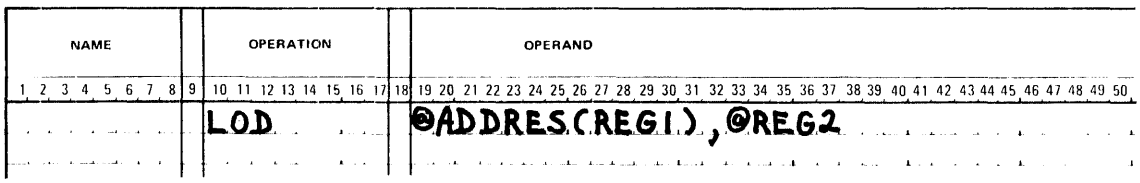
op(,subop)

op(subop,subop)

Operands specify immediate values, memory locations, or general register numbers, while suboperands specify explicit lengths or index registers.

If indexing is not desired for an instruction, the suboperand used for indexing is omitted. General register zero cannot be used by machine instructions as an index. Its specification as an index is flagged as an error.

The at-sign (@) in the first character position of an operand specifies indirect addressing of a memory address or general register, as in the following example.



NAME AND LENGTH ATTRIBUTES

Any machine instruction statement may be named by a symbol, to which other assembler statements can refer. The value attribute of the symbol is the address of the leftmost byte assigned to the assembled instruction.

The length attribute may be 2, 4, 6, or 8, depending upon how many bytes are used for an instruction.

The length field of 6 or 8 byte instruction formats can be explicit or implied. To imply a length, the length suboperand is omitted. The omission indicates that the length field is either the length attribute of the expression specifying the effective address, or the length of the literal.

The length attribute of an expression is the length attribute of the leftmost (or only) term in the expression.

NOTATION USED TO DESCRIBE MACHINE INSTRUCTIONS

The source formats of the operands are defined using the following symbols.

Op Code	The operation codes are presented in hexadecimal (00 through FF).
R	Absolute expression specifying a general register number, 0-7. The register may be used as a sending or receiving field (0-7), or as an index register (1-7 only).
E	Absolute expression specifying an extended register, 0-15 (for RDX and WRX only).
M	Absolute or relocatable expression that specifies a memory address, 0-65,535.

I — An absolute expression specifying an immediate value; the value varies depending on the instruction. The value may represent an amount used in an arithmetic operation, a shift count, a skip count, or a bit number.

L — Absolute expression specifying a field length, usually 0-255, but longer for some instructions. For certain instructions the length of an operand field may be defined in the instruction. The length specified in the instruction overrides any previous field length definition, but is only in effect for that instruction.

@ — An at-sign (@) in a source operand indicates indirect addressing, an optional feature. For the instructions in which a register is a sending or receiving field, the at-sign indicates indirect addressing for R_1 or R_2. If a field in memory is the sending or receiving field, the at-sign indicates indirect addressing of M_1 or M_2.

( ) — Index registers and field lengths are optional; they are enclosed in parentheses in a source operand. A source operand using both an indexing and a field length specification would be represented like this: $M_1(L_1,R_1)$. The comma in the parentheses must not only be coded when both the length and index register are used, but also if the second operand is used, as follows: $M_1(L_1)$ or $M_1(,R_1)$. This enables the assembler to distinguish between the two specifications in parentheses.

• — A bullet following a mnemonic indicates the operands are byte-addressable; all other operands are word-addressable only.

An R, M, I, or L in source operand 1 is identified as R_1, M_1, I_1, or L_1; in source operand 2 they are identified as R_2, M_2, I_2, or L_2.

The two major operand fields must be separated by a comma; no blanks are allowed anywhere in the operand fields.

Remember that the at-sign and any designations in parentheses (field length and index registers) are almost always optional; if any of these designations are not optional, this fact will be noted. Data flow is usually operand 1 to operand 2, unless stated otherwise.

SUMMARY OF MACHINE INSTRUCTIONS

The MRX 40/50 System machine instruction set is divided into two major categories: General-Purpose instructions and System instructions. General-purpose instructions are the instructions needed to solve most data processing problems using a defined software system. System instructions are specialized instructions used to interpret and alter a software system.

The General-Purpose instructions may be used at any time; the System instructions require certain preconditions and cautions. For information on using the System instructions refer to the publication **7200 or 7300 Processing Unit Reference.**

Within these two major categories, the instructions are divided into functional groups, and these functional groups are listed in alphabetical order, as shown in the following table.

General Purpose Instructions	System Instructions
Arithmetic	Control
Bit-Oriented	I/O
Boolean Logic	
Branching	
Compare	
Control	
Data Conversion	
Data Transfer	
Shift	
Optional: Floating Point	

The instructions in each functional group are listed alphabetically by mnemonic. This rule holds for all instructions except for logical pairs or groups of instructions — these instructions are listed alphabetically according to the first instruction of the pair. For instance, PAKX (Pack) will be followed by UNPX (Unpack), and SB (Skip Back Unconditional) will be followed by SF (Skip Forward Unconditional).

GENERAL-PURPOSE INSTRUCTIONS

Arithmetic

Mnemonic	Name	Code	Lgth	Operands
ADD	Add Memory-Register	A2	4	$@M_1(R_1),@R_2$
ADDD	Add Direct	B2	4	$I_1(R_1),@R_2$
ADDI	Add Immediate	32	2	$I_1,@R_2$
ADDK	Added Packed Decimal •	52	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
ADDM	Add Memory-Memory	62	6	$@M_1(R_1),@M_2(R_2)$
ADDR	Add Register-Register	22	2	$@R_1,@R_2$
ADDT	Add Two-Word	72	4	$@M_1(R_1),@R_2$
DIV	Divide Memory-Register	A9	4	$@M_1(R_1),@R_2$
DIVD	Divide Direct	B9	4	$I_1(R_1),@R_2$
DIVK	Divide Packed Decimal •	7C	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
DIVI	Divide Immediate	39	2	$I_1,@R_2$
DIVM	Divide Memory-Memory	69	6	$@M_1(R_1),@M_2(R_2)$
DIVR	Divide Register-Register	29	2	$@R_1,@R_2$
MPY	Multiply Memory-Register	A8	4	$@M_1(R_1),@R_2$
MPYD	Multiply Direct	B8	4	$I_1(R_1),@R_2$
MPYI	Multiply Immediate	38	2	$I_1,@R_2$
MPYK	Multiply Packed Decimal •	5B	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
MPYM	Multiply Memory-Memory	68	6	$@M_1(R_1),@M_2(R_2)$
MPYR	Multiply Register-Register	28	2	$@R_1,@R_2$
SUB	Subtract Memory-Register	A3	4	$@M_1(R_1),@R_2$
SUBD	Subtract Direct	B3	4	$I_1(R_1),@R_2$
SUBI	Subtract Immediate	33	2	$I_1,@R_2$
SUBK	Subtract Packed Decimal •	53	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
SUBM	Subtract Memory-Memory	63	6	$@M_1(R_1),@M_2(R_2)$
SUBR	Subtract Register-Register	23	2	$@R_1,@R_2$
SUBT	Subtract Two-word	73	4	$@M_1(R_1),@R_2$
ZADK	Zero and Add Decimal •	50	8	$M_1(L_1,R_1),M_2(L_2,R_2)$

Bit-Oriented Instructions

IBIT	Invert Bit •	BF	4	$@M_1(R_1),I_2$
ROFR	Reverse Off-Bit	6F	2	$@R_1,@R_2$
RONR	Reverse On-Bit	6D	2	$@R_1,@R_2$
SBIT	Set Bit •	BC	4	$@M_1(R_1),I_2$
RBIT	Reset Bit •	BD	4	$@M_1(R_1),I_2$
TBIT	Test Bit •	BE	4	$@M_1(R_1),I_2$
TOFR	Test for Off-Bit	6E	2	$@R_1,@R_2$
TONR	Test for On-Bit	6C	2	$@R_1,@R_2$

Boolean Logic Instructions

AND	Logical Product Memory-Register	A5	4	$@M_1(R_1),@R_2$
ANDD	Logical Product Direct	B5	4	$I_1(R_1),@R_2$

Boolean Logic Instructions (Continued)

Mnemonic	Name	Code	Lgth	Operands
ANDI	Logical Product Immediate	35	2	I_1,@R_2
ANDM	Logical Product Memory-Memory	65	6	@M_1(R_1),@M_2(R_2)
ANDR	Logical Product Register-Register	25	2	@R_1,@R_2
EOR	Exclusive OR Memory-Register	A6	4	@M_1(R_1),@R_2
EORD	Exclusive OR Direct	B6	4	I_1(R_1),@R_2
EORI	Exclusive OR Immediate	36	2	I_1,@R_2
EORM	Exclusive OR Memory-Memory	66	6	@M_1(R_1),@M_2(R_2)
EORR	Exclusive OR Register-Register	26	2	@R_1,@R_2
IOR	Inclusive OR Memory-Register	A7	4	@M_1(R_1),@R_2
IORD	Inclusive OR Direct	B7	4	I_1(R_1),@R_2
IORI	Inclusive OR Immediate	37	2	I_1,@R_2
IORM	Inclusive OR Memory-Memory	67	6	@M_1(R_1),@M_2(R_2)
IORR	Inclusive OR Register-Register	27	2	@R_1,@R_2

Branching Instructions

B	Branch (post-indexing)	ED	4	@M_1(R_1)
BA1	Branch Add One	E4	4	@M_1(R_1),@R_2
BA2	Branch Add Two	E5	4	@M_1(R_1),@R_2
BCF	Branch on Condition Register False	E9	4	@M_1(R_1),I_2
BCT	Branch on Condition Register True	E8	4	@M_1(R_1),I_2
BCH	Branch Uncond. (pre-indexing)	EC	4	@M_1(R_1)
BOF	Branch if Bit Off	E2	4	@M_1(R_1),I_2
BON	Branch if Bit On	E3	4	@M_1(R_1),I_2
BR	Branch to Address in Register	EB	2	@R_1
BRN	Branch if Register is Not Zero	E1	4	@M_1(R_1),@R_2
BRZ	Branch if Register is Zero	E0	4	@M_1(R_1),@R_2
BS1	Branch Subtract One	E6	4	@M_1(R_1),@R_2
BS2	Branch Subtract Two	E7	4	@M_1(R_1),@R_2
BSR	Branch and Save Return	EA	4	@M_1(R_1),@R_2
SB	Skip Back - Uncond.	BB	2	I_1
SF	Skip Forward - Uncond.	BA	2	I_1
SCFB	Skip on Condition False - Back	4B	2	I_1,I_2
SCFF	Skip on Condition False - Forward	49	2	I_1,I_2
SCTB	Skip on Condition True - Back	4A	2	I_1,I_2
SCTF	Skip on Condition True - Forward	48	2	I_1,I_2
SRMB	Skip if Reg. Minus - Back	47	2	I_1,R_2
SRMF	Skip if Reg. Minus - Forward	46	2	I_1,R_2
SRPB	Skip if Reg. Plus - Back	45	2	I_1,R_2
SRPF	Skip if Reg. Plus - Forward	44	2	I_1,R_2
SRNB	Skip if Reg. Not Zero - Back	43	2	I_1,R_2
SRNF	Skip if Reg. Not Zero - Forward	42	2	I_1,R_2
SRZB	Skip if Reg. Zero - Back	41	2	I_1,R_2
SRZF	Skip if Reg. Zero - Forward	40	2	I_1,R_2

Compare Instructions

Mnemonic	Name	Code	Lgth	Operands
CBY	Compare Byte Memory-Register •	F9	4	$@M_1(R_1),@R_2$
CBYM	Compare Byte Memory-Memory •	6B	6	$@M_1(R_1),@M_2(R_2)$
CMP	Compare Memory-Register	A1	4	$@M_1(R_1),@R_2$
CMPD	Compare Direct	B1	4	$I_1(R_1),@R_2$
CMPI	Compare Immediate	31	2	$I_1,@R_2$
CMPK	Compare Packed Decimal •	51	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
CMPM	Compare Memory-Memory	61	6	$@M_1(R_1),@M_2(R_2)$
CMPR	Compare Register-Register	21	2	$@R_1,@R_2$
CMPT	Compare Two-Word	71	4	$@M_1(R_1),@R_2$
CMPX	Compare Characters •	55	8	$M_1(L_1,R_1),M_2(L_2,R_2)$

Control Instructions

General Purpose Control instructions can be used at any time without preconditions; compare with System Control instructions.

NOP	No Operation	EE	4	Blank or $@M_1(R_1),R_2$
RDX	Read Extended Register	F0	2	E_1,R_2
SR	Service Request	13	2	$@I_1$

Data Conversion Instructions

CVB	Convert to Binary •	AA	4	$@M_1(R_1),R_2$
CVBT	Convert to Binary Two-Word •	AA	4	$@M_1(R_1),R_2$
CVD	Convert to Decimal •	AB	4	$@M_1(R_1),R_2$
CVDT	Convert to Decimal Two-Word •	AB	4	$@M_1(R_1),R_2$
EDTX	Packed Decimal/Alpha Edit •	57	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
PAKX	Pack •	58	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
UNPX	Unpack •	59	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
TRNX	Translate •	56	8	$M_1(R_1),M_2(L_2,R_2)$

Data Transfer Instructions

CLDR	Condition Register Load	2B	2	$@R_1$
CSTR	Condition Register Store	2A	2	$@R_1$
INV	Inverse Move Memory-Register	A4	4	$@M_1(R_1),@R_2$
INVD	Inverse Move Direct	B4	4	$I_1(R_1),@R_2$
INVI	Inverse Move Immediate	34	2	$I_1,@R_2$
INVM	Inverse Move Memory-Memory	64	6	$@M_1(R_1),@M_2(R_2)$
INVR	Inverse Move Register-Register	24	2	$@R_1,@R_2$
LOD	Load Memory-Register	A0	4	$@M_1(R_1),@R_2$
LODB	Load Byte •	F7	4	$@M_1(R_1),@R_2$
LODD	Load Direct	B0	4	$I_1(R_1),@R_2$
LODI	Load Immediate	30	2	$I_1,@R_2$
LODT	Load Two-Word	70	4	$@M_1(R_1),@R_2$
MOVB	Move Byte •	6A	6	$@M_1(R_1),@M_2(R_2)$

Data Transfer Instructions (Continued)

Mnemonic	Name	Code	Lgth	Operands
MOVL	Move Long •	5A	8	$M_1(L_1,R_1),M_2(R_2)$
MOVM	Move Memory-Memory	60	6	$@M_1(R_1),@M_2(R_2)$
MOVR	Move Register-Register	20	2	$@R_1,@R_2$
MOVX	Move Characters •	54	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
PSTR	Program Address Store	3A	2	$@R_1$
STO	Store Memory-Register	FA	4	$@M_1(R_1),@R_2$
STOB	Store Byte •	F8	4	$@M_1(R_1),@R_2$
STOT	Store Two-Word	FB	4	$@M_1(R_1),@R_2$

Shift Instructions

ARDI	Arithmetic Right Double Shift - Immediate	5F	2	I_1,R_2
ARDR	Arithmetic Right Double Shift - By Register	3F	2	$@R_1,R_2$
ARSI	Arithmetic Right Single Shift - Immediate	4F	2	I_1,R_2
ARSR	Arithmetic Right Single Shift - By Register	2F	2	$@R_1,R_2$
LLDI	Logical Left Double Shift - Immediate	5C	2	I_1,R_2
LLDR	Logical Left Double Shift - By Register	3C	2	$@R_1,R_2$
LLSI	Logical Left Single Shift - Immediate	4C	2	I_1,R_2
LLSR	Logical Left Single Shift - By Register	2C	2	$@R_1,R_2$
LRDI	Logical Right Double Shift - Immediate	5D	2	I_1,R_2
LRDR	Logical Right Double Shift - By Register	3D	2	$@R_1,R_2$
LRSI	Logical Right Single Shift - Immediate	4D	2	I_1,R_2
LRSR	Logical Right Single Shift - By Register	2D	2	$@R_1,R_2$
RLDI	Rotating Left Double Shift - Immediate	5E	2	I_1,R_2
RLDR	Rotating Left Double Shift - By Register	3E	2	$@R_1,R_2$
RLSI	Rotating Left Single Shift - Immediate	4E	2	I_1,R_2
RLSR	Rotating Left Single Shift - By Register	2E	2	$@R_1,R_2$
SHFK	Shift Packed Decimal •	3B	6	$M_1(L_1,R_1),I_2(R_2)$

Floating Point Instructions (Optional)

ADDF	Add Floating Point	86	4	$@M_1(R_1),R_2$
CMPF	Compare Floating Point	87	4	$@M_1(R_1)$
DIVF	Divide Floating Point	89	4	$@M_1(R_1),R_2$
FLT	Convert Fixed to Float	82	2	$@R_1$
FLTT	Convert Fixed to Float Two Word	82	2	$@R_1$
INT	Convert Float to Fixed	81	2	$@R_1,R_2$
INTT	Convert Float to Fixed Two Word	81	2	$@R_1,R_2$
LODF	Load Floating Point Register	84	4	$@M_1(R_1),R_2$
MPYF	Multiply Floating Point	88	4	$@M_1(R_1),R_2$
NEGF	Negate Floating Point Register	80	2	

Floating Point Instructions (Optional) (Continued)

Mnemonic	Name	Code	Lgth	Operands
STOF	Store Floating Point Register	8A	4	@$M_1(R_1)$
SUBF	Subtract Floating Point	85	4	@$M_1(R_1),R_2$

SYSTEM INSTRUCTIONS

Privileged and restricted classes; consult 7200 or 7300 Processing Unit Reference manual for information on the use of these system instructions

Control Instructions

Mnemonic	Name	Code	Lgth	Operands
CTB	Clear Tie-Breaker Register	12	2	I_1
TST	Test and Set Tie-Breaker Register	11	2	I_1
BCM	Branch to Control Memory	EF	2	R_1,I_2
RAR	Read Any Register	FE	4	$I_1(R_1)$,@R_2
WAR	Write Any Register	FE	4	$I_1(R_1)$,@R_2
RRO	Read Register - Option Register	FD	4	$I_1(R_1)$,@R_2
WRO	Write Register - Option Register	FD	4	$I_1(R_1)$,@R_2
SAR	Save All Registers	FF	4	$M_1(R_1),I_2$ or $M_1(R_1)$,@R_2
RSAR	Restore All Registers	FF	4	$M_1(R_1),I_2$ or $M_1(R_1)$,@R_2
SBA	Set Busy/Active Register	10	2	I_1,I_2 or @R_1,I_2
RBA	Reset Busy/Active Register	10	2	I_1,I_2 or @R_1,I_2
SCN	Set Control Register	14	2	I_1,I_2 or @R_1,I_2
RCN	Reset Control Register	14	2	I_1,I_2 or @R_1,I_2
SPM	Set Privileged Mode Register	15	2	I_1,I_2 or @R_1,I_2
RPM	Reset Privileged Mode Register	15	2	I_1,I_2 or @R_1,I_2
WRX	Write Extended Register	F0	2	E_1,R_2

I/O Instructions

DIO	Disc Input/Output	F2	2	@R_1,R_2
INP	Input from I/O Register	F5	2	I_1,@R_2
OUT	Output to I/O Register	F6	2	I_1,@R_2
RDC	Communications Input/Output	F3	2	
WRC	Communications Output	F4	2	R_1,R_2
SIO	System Input/Output	F1	2	@R_1,R_2

SUMMARY OF EXTENDED MNEMONICS

The assembler provides extended mnemonic codes which allow unconditional skips, and conditional skips and branches to be written in a symbolic form that is easier to use than standard machine instructions. The assembler translates the extended mnemonic codes into machine instruction object code.

Extended mnemonic codes for skip instructions do not specify the forward (F) or backward (B) direction of the skip. Thus, the extended mnemonic, S, can be used instead of the SF or SB machine instruction. The assembler determines the direction of the skip for the S instruction from the memory address or immediate value in the operand. For example, S DOG skips to the address, DOG, whether DOG is before or after the present location counter.

Extended mnemonic codes for branch and skip instructions that test the condition register specify the condition in the mnemonic itself rather than in the operand for example. SOV ADDRS skips to ADDRS if overflow has occurred. The standard machine instruction names the direction and the bit status in the mnemonic, and the actual bit number tested in the operand. Thus, the extended mnemonic SOV 4 is the same as the standard instruction SCTF 4,0. (Bit 0 of the condition register is the overflow bit.)

The extended mnemonic codes are grouped as follows:

- Address Coded Skips
- After Arithmetic Instructions
- After Compare Instructions – Arithmetic Test
- After Compare Instructions – Logical Test
- After Decimal Instructions
- After PAKX Instruction
- After TBIT Instruction
- Conditional Register Test

Just as for the standard instructions, indirect addressing and indexing are optional for the extended mnemonic codes.

EXTENDED MNEMONIC CODES

Address-Coded Skips

Extended Code		Machine Instruction		Meaning
S	M_1 or I_1	SF	I_1	Skip forward or backward
		SB	I_1	
SRZ	M_1,R_2 or I_1,R_2	SRZF	I_1,R_2	Skip if reg. is zero, forward or
		SRZB	I_1,R_2	backward
SRN	M_1,R_2 or I_1,R_2	SRNF	I_1,R_2	Skip if reg. is non-zero forward or
		SRNB		backward
SRP	M_1,R_2 or I_1,R_2	SRPF	I_1,R_2	Skip if reg. is plus, forward or
		SRPB	I_1,R_2	backward
SRM	M_1,R_2 or I_1,R_2	SRMF	I_1,R_2	Skip if reg. is minus, forward or
		SRMB	I_1,R_2	backward

For S, the I_1 value = -255 through +255; for all other extended mnemonics in this category, I_1 = -15 through +15.

For SF and SB, the I_1 value = 0-255; for all other regular instructions in this category I_1 = 0-15.

After Arithmetic Instructions

BOV	$@M_1(R_1)$	BCT	$@M_1(R_1),0$	Branch if overflow
BNV	$@M_1(R_1)$	BCF	$@M_1(R_1),0$	Branch if no overflow
BCY	$@M_1(R_1)$	BCT	$@M_1(R_1),3$	Branch if carry
BNC	$@M_1(R_1)$	BCF	$@M_1(R_1),3$	Branch if no carry
SOV	M_1 or I_1	SCTF	$I_1,0$	Skip if overflow
		SCTB	$I_1,0$	
SNV	M_1 or I_1	SCFF	$I_1,0$	Skip if no overflow
		SCFB	$I_1,0$	
SCY	M_1 or I_1	SCTF	$I_1,3$	Skip if carry
		SCTB	$I_1,3$	
SNC	M_1 or I_1	SCFF	$I_1,3$	Skip if no carry
		SCFB	$I_1,3$	

I_1 = -15 through +15 for the extended instructions. I_1 = 0-15 for the regular instructions.

After Compare Instructions – Arithmetic Test

The arithmetic test tests the result of a signed arithmetic compare between operand 1 and operand 2. In the following table, 1 and 2 under Meaning refer to the signed values of operands 1 and 2.

Extended Code		Machine Instruction		Meaning
BGT	$@M_1(R_1)$	BCT	$@M_1(R_1),1$	Branch if 1 GT 2
BLT	$@M_1(R_1)$	BCT	$@M_1(R_1),2$	Branch if 1 LT 2
BGE	$@M_1(R_1)$	BCF	$@M_1(R_1),2$	Branch if 1 GE 2
BLE	$@M_1(R_1)$	BCF	$@M_1(R_1),1$	Branch if 1 LE 2
BEQ	$@M_1(R_1)$	BCT	$@M_1(R_1),3$	Branch if 1 EQ 2
BNE	$@M_1(R_1)$	BCF	$@M_1(R_1),3$	Branch if 1 NE 2

After Compare Instructions – Arithmetic Test (Continued)

Extended Code		Machine Instruction		Meaning
SGT	M_1 or I_1	SCTF	I_1,1	Skip if 1 GT 2
		SCTB	I_1,1	
SLT	M_1 or I_1	SCTF	I_1,2	Skip if 1 LT 2
		SCTB	I_1,2	
SGE	M_1 or I_1	SCFF	I_1,2	Skip if 1 GE 2
		SCFB	I_1,2	
SLE	M_1 or I_1	SCFF	I_1,1	Skip if 1 LE 2
		SCFB	I_1,1	
SEQ	M_1 or I_1	SCTF	I_1,3	Skip if 1 EQ 2
		SCTB	I_1,3	
SNE	M_1 or I_1	SCFF	I_1,3	Skip if 1 NE 2
		SCFB	I_1,3	

I_1 = -15 through +15 for extended instructions. I_1 = 0-15 for regular instructions.

After Compare Instructions – Logical Test

The logical test tests the results of an unsigned arithmetic (logical) compare between operand 1 and operand 2. In the following table, 1 and 2 under Meaning refer to the unsigned values of operands 1 and 2. CMPX and all variations of the CBY instruction always yield a logical result.

Extended Code		Machine Instruction		Meaning
BLGT	@$M_1(R_1)$	BCT	@$M_1(R_1)$,5	Branch if 1 GT 2
BLLT	@$M_1(R_1)$	BCT	@$M_1(R_1)$,6	Branch if 1 LT 2
BLGE	@$M_1(R_1)$	BCF	@$M_1(R_1)$,6	Branch if 1 GE 2
BLLE	@$M_1(R_1)$	BCF	@$M_1(R_1)$,5	Branch if 1 LE 2
BLEQ	@$M_1(R_1)$	BCT	@$M_1(R_1)$,7	Branch if 1 EQ 2
BLNE	@$M_1(R_1)$	BCF	@$M_1(R_1)$,7	Branch if 1 NE 2
SLGT	M_1 or I_1	SCTF	I_1,5	Skip if 1 GT 2
		SCTB	I_1,5	
SLLT	M_1 or I_1	SCTF	I_1,6	Skip if 1 LT 2
		SCTB	I_1,6	
SLGE	M_1 or I_1	SCFF	I_1,6	Skip if 1 GE 2
		SCFB	I_1,6	
SLLE	M_1 or I_1	SCFF	I_1,5	Skip if 1 LE 2
		SCFB	I_1,5	
SLEQ	M_1 or I_1	SCTF	I_1,7	Skip if 1 EQ 2
		SCTB	I_1,7	
SLNE	M_1 or I_1	SCFF	I_1,7	Skip if 1 NE 2
		SCFB	I_1,7	

I_1 = -15 through +15 for the extended instructions. I_1 = 0-15 for the regular instructions.

After Decimal Instructions

BKP	@$M_1(R_1)$	BCT	@$M_1(R_1)$,1	Branch if plus
BKM	@$M_1(R_1)$	BCT	@$M_1(R_1)$,2	Branch if minus

After Decimal Instructions (Continued)

Extended Code		Machine Instruction		Meaning
BKZ	@$M_1(R_1)$	BCT	@$M_1(R_1)$,3	Branch if zero
SKP	M_1 or I_1	SCTF	I_1,1	Skip is plus
		SCTB	$I_1$1	
SKM	M_1 or I_1	SCTF	I_1,2	Skip if minus
		SCTB	I_1,2	
SKZ	M_1 or I_1	SCTF	I_1,3	Skip if zero
		SCTB	I_1,3	

I_1 = -15 through +15 for the extended instructions. I_1 = 0-15 for the regular instructions.

After PAKX Instruction

BID	@$M_1(R_1)$	BCT	@$M_1(R_1)$,4	Branch if invalid digit
BNI	@$M_1(R_1)$	BCF	@$M_1(R_1)$,4	Branch if no invalid digit
SID	M_1 or I_1	SCTF	I_1,4	Skip if invalid digit
		SCTB	I_1,4	
SNI	M_1 or I_1	SCFF	I_1,4	Skip if no invalid digit
		SCFB	I_1,4	

I_1 = -15 through +15 for the extended instructions. I_1 = 0-15 for the regular instructions.

After TBIT Instruction

BBS	@$M_1(R_1)$	BCT	@$M_1(R_1)$,0	Branch if bit is set
BBR	@$M_1(R_1)$	BCF	@$M_1(R_1)$,0	Branch if bit is reset
SBS	M_1 or I_1	SCTF	I_1,0	Skip if bit is set
		SCTB	I_1,0	
SBR	M_1 or I_1	SCFF	I_1,0	Skip if bit is reset
		SCFB	I_1,0	

I_1 = -15 through +15 for the extended instructions. I_1 = 0-15 for the regular instructions.

Condition Register Test

SCF	M_1,I_2 or I_1,I_2	SCFF	I_1,I	Skip if bit spec. by I_2 is off
		SCFB	I_1,I	
SCT	M_1,I_2 or I_1,I_2	SCTF	I_1,I	Skip if bit spec. by I_2 is on
		SCTB	I_1,I	

I_1 = -15 through +15 and I_2 = 0-15 for the extended instructions. I_1 and I_2 = 0-15 for the regular instructions.

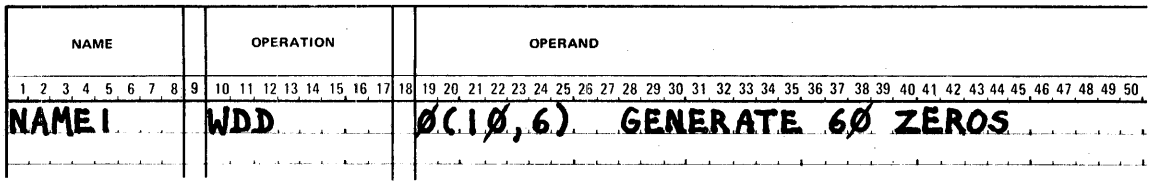
4. ASSEMBLER INSTRUCTION SOURCE STATEMENTS — OVERVIEW

Assembler statements are requests to the assembler to perform certain operations during the assembly. Some statements, such as WDD and BDD, generate data, while others, such as EQU and SPACE, are effective only at assembly time. A summary of assembler statements can be found in Appendix E.

Assembler instruction source statements consist of:

- Name field (usually optional)
- Mnemonic operation code
- Operand field (optional for some statements)
- Comment field (optional)

An example of an assembler instruction source statement is:



The following is a list of assembler statements with a short explanation for each statement.

Program Sectioning and Linking Statements

CSECT Identifies the beginning or continuation of a control section

ENTRY Identifies relocatable symbols defined in the current program that are used in another program

EXTRN Identifies relocatable symbols used in the current program that are defined in another program

COM Identifies the beginning or continuation of a common control section

Program Control Statements

ORG Sets the location counter

END Ends the assembly

PUNCH Writes data in a user-defined file

LTORG	Assembles literals in a pool
ICTL	Specifies nonstandard input format
ISEQ	Checks the lines of code for the correct sequence
ALIGN	Sets the current location counter to a storage boundary

Linkage Editor Map Address Directive Statement

SEG	Defines load-module segment

Symbol and Data Definition Statements

EQU	Defines a symbol and assigns values and attributes to it
WDD	Defines word aligned data (in bytes)
BDD	Defines byte aligned data (in bytes)
WRS	Reserves word aligned storage (in words)
BRS	Reserves byte aligned storage (in bytes)
FORM	Defines bit-oriented formats (in storage bytes)

Listing Control Statements

TITLE	Identifies the listing
EJECT	Starts a new page
SPACE	Inserts blank lines
PRINT	Specifies the details to be printed

Macro Definition Statements

MACRO	Begin macro definition
MEXIT	Conditional exit from macro definition
MEND	End macro definition
MNOTE	Macro message

Conditional Assembly Statements

SETA	Assigns arithmetic values to set symbols
SETC	Assigns character values to set symbols
GBLA	Defines a SETA symbol as global
GBLC	Defines a SETC symbol as global
ADO	Sets up a source statement generation loop
AGO	Specifies a branch to another statement
ANOP	Specifies an assembly no-operation statement

5. PROGRAM SECTIONING AND LINKING STATEMENTS

The programmer can divide a lengthy or complex program into control sections to make it more manageable and easier to debug. Each section is assigned a unique name. The operating system treats each section as an independent, relocatable routine that can be executed alone or linked with others.

During assembly, the assembler creates an index of all assigned control section names. At load time, the Linkage Editor uses the index to link the various control sections into a single storage module, from which the connected sections can be executed as a complete program.

The assembler mnemonics and functions of the program sectioning and linking statements are:

CSECT Identifies the beginning or continuation of a control section.

ENTRY Identifies relocatable symbols defined in the current program that are used in another program.

EXTRN Identifies relocatable symbols used in the current program that are defined in another program.

COM Identifies the beginning or continuation of a common control section.

(The reserved symbolic segment tag, $SYSEG, is also explained in this section.)

CSECT – IDENTIFY CONTROL SECTION

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

Name	Operation	Operand
Symbol or blank	CSECT	Not used – ignored by the assembler

If a symbol appears in the name field, it is the name of the control section; otherwise, an unnamed control section is defined. The symbol in the name field represents the address of the first byte of the control section. It has a length attribute of 1.

To preclude the generation of an unnamed CSECT section; the CSECT statement must precede all statements except the following: macro and FORM definitions, listing control statements, conditional assembly statements, ICTL and ISEQ statements, EXTRN and ENTRY statements, PUNCH statements, and comments.

If the assembler encounters a statement other than these before a CSECT statement, an unnamed CSECT statement is generated.

All statements following the CSECT statement are assembled as part of that control section until another CSECT or COM statement is encountered. A control section can be interrupted and then resumed by inserting another CSECT statement with the same name, as in the following example. Control section ONE includes all code between the first ONE CSECT card and the TWO CSECT card plus all code following the second ONE CSECT card. Unnamed control sections can also be resumed.

```
ONE     CSECT
          .
          .      } Control section ONE
          .
TWO     CSECT
          .
          .      } Control section TWO
          .
ONE     CSECT
          .
          .      } Control section ONE
          .
```

SYMBOLIC LINKAGE STATEMENTS – ENTRY AND EXTRN

The symbolic linkage statements, ENTRY and EXTRN, allow a symbol to be defined in one program and referred to in another program. The program defining the symbol uses the ENTRY statement; the program referencing the symbol uses the EXTRN statement. In both instances, the assembler provides the linkage editor with the information to resolve the symbolic linkage.

ENTRY – IDENTIFY ENTRY POINT SYMBOL

The ENTRY statement specifies which relocatable symbols defined in the current program can be accessed by other programs. The format of the ENTRY statement is:

Name	Operation	Operand
Sequence symbol or blank	ENTRY	One or more relocatable symbols separated by comma

The symbols in the operand field may be used as operands by other programs. Control sections named in CSECT or COM statements are automatically considered entry points and do not have to be listed in an ENTRY statement. In the following example, the ENTRY statement identifies SUB1 and SUB2 as entry points to the program:

NAME	OPERATION	OPERAND
	ENTRY	SUB1,SUB2

EXTRN – IDENTIFY EXTERNAL SYMBOL

The EXTRN statement identifies relocatable symbols that can be used in the current program, although they are defined elsewhere. The format of the EXTRN statement is:

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

Symbols named in the operand field cannot appear in an ENTRY statement in the same program. The combined total of control section and external symbols in the same program cannot exceed 252.

The example in Figure 5-1 shows how two programs, PROGA and PROGB, use EXTRN and ENTRY, so PROGA can use symbols defined in PROGB. EXTRN in PROGA identifies FETCH as a symbolic address that is defined in another program. ENTRY in PROGB defines FETCH as an entry point in PROGB. Thus PROGA can use FETCH as an operand without first defining it.

NAME	OPERATION	OPERAND
PROGA	CSECT	
	EXTRN	FETCH
	.	
	.	
	.	
	B	FETCH
	.	
	.	
	.	
	END	
PROGB	CSECT	
	ENTRY	FETCH
	.	
	.	
	.	
FETCH	LODB	TAGE,R1
	.	
	.	
	.	
	END	

Figure 5-1. Example of EXTRN and ENTRY

COM – DEFINE COMMON CONTROL SECTION

The COM statement identifies the beginning or continuation of a common control section. The format of the COM statement is:

Name	Operation	Operand
Symbol or blank	COM	Not used – ignored by the assembler

If a symbol appears in the name field, it is the name of the common storage area; otherwise, an unnamed common storage area is defined. An unnamed COM control section does not have the same name as an unnamed CSECT section. If two COM statements with the same name appear in the same program, the second is a continuation of the first. When COM statements are assembled, the common location assignment starts at zero. The symbol in the name field represents the address of the first byte of the control section. It has a length attribute of 1.

Since a COM control section's primary function is to define the structure of a storage area used by more than one program, no binary data can be generated by statements within the COM section. However, pertinent storage information such as address assignment, relocatability, and length attributes are retained and assembled in the normal way, without binary output.

Data can be stored in a common storage area during assembly only by a CSECT control section in a different assembly, in which case the CSECT control section must use the same name as the common area defined and reserved by COM statements. After the programs are assembled and the Linkage Editor has performed the necessary linkage, data can be retrieved, stored, checked, and manipulated in the common area by any cognizant program currently being executed.

No more than one CSECT control section may be included in a set of COM control sections identified by the same name in the same load module.

The example in Figure 5-2 shows how two programs communicate information through the common area. Each program must know the other's plan for structuring and using the common storage area. If program PROGA is to pass information to PROGB, PROGB must know the location in COMMON to which PROGA will pass the data. In this example, information is passed through COMMON at location TAGE. In PROGA, the statement STO TAG5,R2 stores the contents of register 2 at location TAG5. After the branch is made to FETCH, PROGB loads the contents from TAGE (which is the same location as TAG5 in PROGA) into its register and checks to see if it is equal to the hexadecimal constant: 'AF'. If a true comparison is found, a branch is made to PROGC. Note that when the Linkage Editor is called to link the programs at load time, each program is linked to COMMON.

```
NAME      OPERATION      OPERAND
PROGA     CSECT
          ENTRY          PROGC
          EXTRN          FETCH
            .
            .
          LOD            TAG1,R1
          STO            TAG2,R1
          MOVX           TAG4,TAG5
PROG      LODD           "AF1D,R2
          STO            TAG5,R2
          B              FETCH
            .
            .
COMMON    COM
TAG1      WDD            X'FF'(,2)
TAG2      WDD            Ø(8)
TAG3      WDD            Ø(4,3)
TAG4      BRS            1Ø
TAG5      BRS            1Ø
            .
            .
          END
```

```
NAME      OPERATION      OPERAND
PROGB     CSECT
          ENTRY          FETCH
          EXTRN          PROGC
            .
            .
FETCH     LODB           TAGE,R1
          CBY            =X'AF',R1
          BEQ            PROGC
            .
            .
COMMON    CSECT
TAGA      WDD            X'FF'(,2)
TAGB      WDD            Ø(8)
TAGC      WDD            Ø(4,3)
TAGD      BRS            1Ø
TAGE      BRS            1Ø
TAGF      WDD            1(2,4)
TAGG      BRS            2Ø
TAGH      BRS            1Ø
            .
            .
          END
```

Figure 5-2. Example of the COM Statement

RESERVED SYMBOLIC SEGMENT NAME – $SYSEG

The reserved symbolic segment name, $SYSEG, is a relocatable segment designator which makes extended addressing possible. A reserved name has a special meaning to the system and should not be used as a symbol for any purpose other than its special meaning.

Symbolic segment tags are used for address referencing across program (or storage) segments. However, a user whose program is limited to one segment need not use segment tags. To make all system interfaces (I/O requests and service requests) compatible, addresses specified in these interfaces must have an associated segment tag.

When a single-segment program is written, the assembler assigns the global system name, $SYSEG, to the first (and only) segment in the program. $SYSEG is automatically entered in the symbol table by the assembler, and thus becomes a reserved identifier.

The name $SYSEG is used as a default value in all system macros containing address parameters. The user must concern himself with $SYSEG only if he is coding system interfaces directly without using the regular system macros.

$SYSEG cannot be used in a multi-term expression. It can only be used as an operand in a BDD, WDD, or FORM instruction. If it is used in a FORM instruction, the size of the corresponding definition field must be 8 bits in length and start on a byte boundary. The length attribute of $SYSEG is 1.

6. PROGRAM CONTROL STATEMENTS

The assembler mnemonics and functions of the program control statements are:

ORG	Sets the location counter
END	Ends the assembly
PUNCH	Writes data in a user-defined sequential disk file.
LTORG	Inserts the accumulated literal pool, starting at the current location counter.
ICTL	Specifies nonstandard input format.
ISEQ	Checks the lines of code for the correct sequence.
ALIGN	Sets the current location counter to a storage boundary address.

ORG – SET LOCATION COUNTER

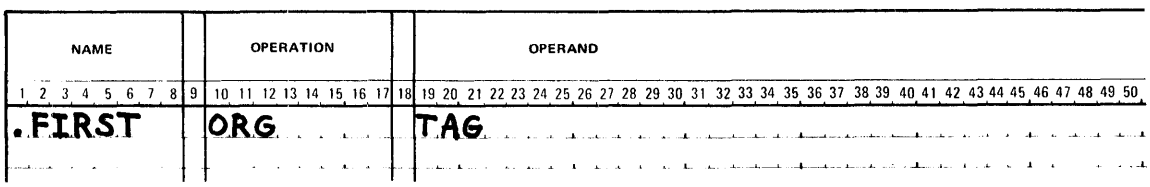
The ORG statement alters the setting of the location counter. The format of the ORG statement is:

Name	Operation	Operand
Sequence symbol or blank	ORG	Relocatable expression or blank

Symbols in the expression must be 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, or to its previous high count, if the operand field is blank.

Since the location counter points to a storage location that is to receive the next line of assembled code, altering its setting permits a programmer to return to a previous location in his program and change its contents. In this way, an area can be redefined during assembly, changing data definitions to meet various requirements in the program.

An example of an ORG statement is:



If the location named TAG is twelve bytes from the beginning of the control section, this statement sets the location counter to 000C. Statements that follow the ORG statement are then assembled beginning at address 000C. The information previously assembled at these locations is lost, although the name tags remain unchanged and cannot be duplicated.

The ORG statement can also be used in direct reference to the location counter. The format of the operand is *-n or *+n. The asterisk specifies the present location counter, and n specifies the number of bytes. The following statement decreases the present location counter by 60:

```
NAME      OPERATION      OPERAND
.SECD     ORG            *-60
```

An ORG statement without an operand returns the location counter to its previous high count, as in the following statement. If the counter has already reached its previous high, an ORG without an operand has no effect.

```
NAME      OPERATION      OPERAND
.THRD     ORG
```

The following example shows an ORG with a location counter reference and an ORG without an operand. If the location counter is 0050 when ORG *-20 is encountered, the count is decreased by 20 bytes to 003C. Then if the ORG without an operand is encountered before the counter reaches its previous high of 0050, the count is returned to 0050.

```
NAME      OPERATION      OPERAND
          .
          .
          .
          ORG            *-20
          .
          .
          .
          ORG
```

For an example of an application of the ORG statement, refer to Figure 8-3 in Chapter 8.

END – END ASSEMBLY

The END statement terminates the assembly of a program. The format of the END statement is:

Name	Operation	Operand
Blank	END	Ordinary symbol or blank

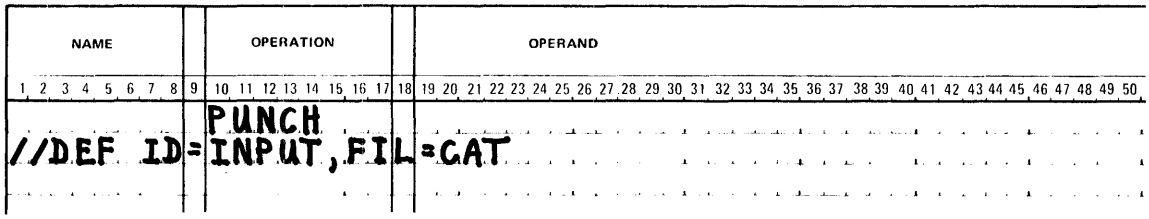
An ordinary symbol in the operand field specifies the point to which control is to be transferred when loading is complete. The ordinary symbol must identify a symbolic address in the current assembly. Substitution is invalid on the END statement. Continuation is ignored on the END statement.

PUNCH – WRITE TO FILE

The PUNCH statement writes data in a user-defined sequential disk file. The format of the PUNCH statement is:

Name	Operation	Operand
Sequence symbol or blank	PUNCH	Not used

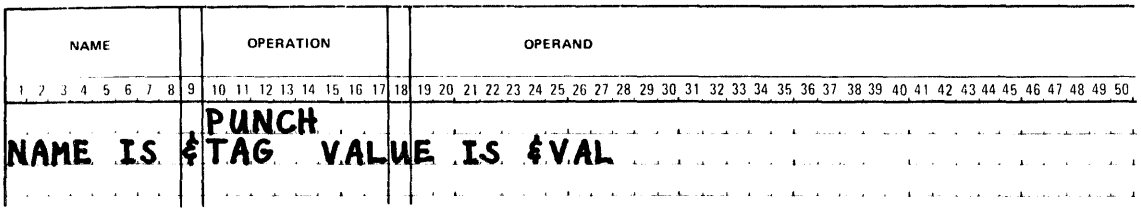
The PUNCH statement precedes the line of code that is to be written in the file. The line of code can be in any format and it cannot be continued. In the following example, //DEF ID=INPUT,FIL=CAT is written in a file defined by the user:



Variable symbols in the line of code are resolved before the record is written to the file. If substitution results in a record that exceeds the standard statement size (columns 1 through 72) or nonstandard size specified by an ICTL statement, excess characters are truncated on the right.

The PUNCH statement does not lend itself to a fixed field format, because as values of various lengths are substituted for a variable symbol, the position of the subsequent fields is shifted.

In the following example, values are substituted for the variable symbols $TAG and $VAL before the record is written.



If $TAG is equal to AB and $VAL is equal to 75, the record written in the file is: NAME IS AB VALUE IS 75.

If $TAG is equal to C'THIS IS A MESSAGE' and $VAL is equal to 10, the record written in the file is: NAME IS C'THIS IS A MESSAGE' VALUE IS 10.

Notice that the statement VALUE IS. . .is moved to the right to make room for the first statement NAME IS. . . .

A fixed field format may be established by using a variable symbol with as many characters (including the ampersand) as the substituted value.

LTORG – BEGIN LITERAL POOL

The LTORG statement assembles previously defined literals into a single area called a literal pool. All preceding literals, back to the beginning of the program, or back to the last LTORG statement, are assembled at the next word boundary. If a LTORG statement is not used, all literals are assembled after the first control section. Literals that appear after the last LTORG statement are also assembled after the first control section.

The format of the LTORG statement is:

Name	Operation	Operand
Symbol or blank	LTORG	Not used

A symbol in the name field represents the address of the first byte of the literal pool. The length attribute of the symbol is 1.

ICTL – INPUT FORMAT CONTROL

The ICTL statement specifies that statements in a program begin and end in columns other than the standard columns 1 and 72. The format of the ICTL statement is:

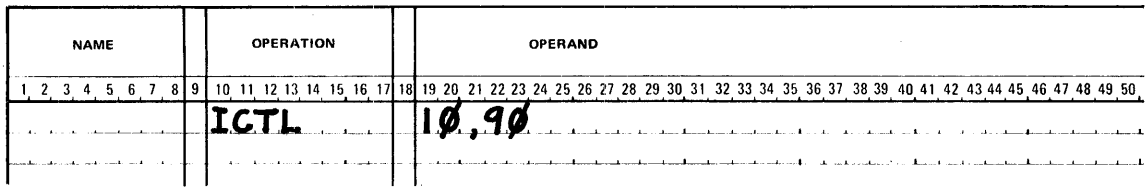
Name	Operation	Operand
Not used	ICTL	Two decimal arithmetic constants separated by a comma

Both decimal constants are required. The first constant specifies the beginning column (1-40). The second specifies the end column (41-120). A comma must separate the two constants.

If the source program does not contain an ICTL statement, the standard format (columns 1 and 72) is used. Because system macros are always processed in the standard format, they are not affected by the ICTL statement.

Since the ICTL statement establishes a new format, it must be the first statement in the program and can be used only once. The ICTL statement cannot be continued.

The following statement establishes column 10 as the begin column and column 90 as the end column.



ISEQ – INPUT SEQUENCE CHECKING

The ISEQ statement specifies that all subsequent statements are to be checked or not checked for correct sequence. The format of the ISEQ statement is:

Name	Operation	Operand
Blank	ISEQ	Blank, or two decimal arithmetic constants, separated by a comma

If the operand field is used, both decimal constants must be specified. The first constant specifies the column number of the leftmost character of the sequence field; the second constant specifies the column number of the rightmost character of the sequence field. A comma must separate the two operands.

Sequence checking begins with the first line following the ISEQ statement and continues up to an ISEQ statement without an operand. Checking can be resumed with another ISEQ statement. If each succeeding sequence field is not higher in value than that of the preceding statement, a sequence error message is generated. A sequence error does not terminate the assembly.

The columns identified by the ISEQ statement must be between columns 73-80 if the standard begin and end columns are used, or outside the begin and end columns defined by an ICTL statement.

Sequence checking is only performed on statements contained in the source program. Macro definitions in a macro library or lines generated by a macro instruction are not checked. Lines with a blank sequence field are always considered to be in the correct order.

The operand field of an ISEQ statement may not contain a reference to a variable symbol. The ISEQ statement cannot be continued.

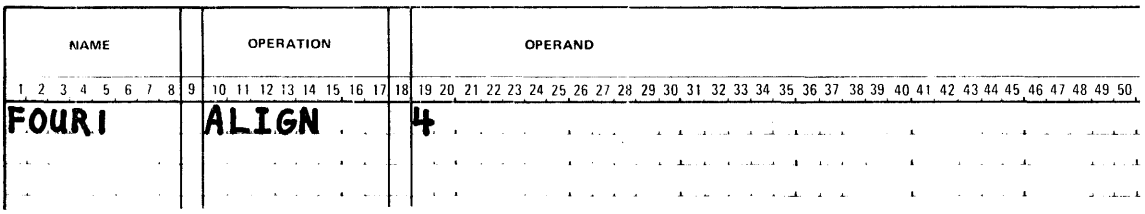
ALIGN – ALIGN LOCATION COUNTER

The ALIGN statement sets the value of the location counter to an address determined by a value in the operand field; the assembler updates the counter to the next highest address which is a multiple of the expressed value in the operand. The format of the ALIGN statement is:

Name	Operation	Operand
Symbol or blank	ALIGN	Absolute arithmetic expression

The operand can be an expression to be evaluated by the assembler; however, all symbols must be previously defined.

In the following example, the location counter is set to the next highest multiple of 4 addresses. If the location counter is at 1009 when the statement is encountered, it is set to 100C. However, if the location counter is already set to an address which is a multiple of the operand value (1008, 100C, 1010, etc.), the counter is not changed. After alignment, the address of the location counter is assigned the symbolic name in the name field. In the example, FOUR1 is equal to 100C. The length attribute is always 1.



If the counter is set to an address exceeding $65{,}535_{10}$, an error message is generated and alignment occurs. The location counter is set to the value exceeding $65{,}535_{10}$. An address of $65{,}600_{10}$ will set the location counter to 64_{10}.

7. LINKAGE-EDITOR MAP DIRECTIVE — SEG

The Linkage Editor map directive, SEG, may be used as an assembly language statement. The assembler does not process the statement, but simply writes the directives, in their source form, on the output file.

All SEG statements in an assembly must immediately precede the END statement. The only statements that can appear between the SEG statement and the END statement are the conditional assembly statements SETA, SETC, ADO, AGO, ANOP, and macro instructions generating these statements. A SEG statement cannot be continued and no substitution is performed.

The function and format of the SEG statement is described in the publication **MRX/OS Program Library Services Reference.**

8. SYMBOL AND DATA DEFINITION STATEMENTS

The assembler mnemonics and functions of the symbol and data definition statements are:

EQU	Defines a symbol and assigns values and attributes to it.
WDD	Defines word-aligned data (in bytes).
BDD	Defines byte-aligned data (in bytes).
WRS	Reserves word-aligned storage (in words).
BRS	Reserves byte-aligned storage (in bytes).
FORM	Defines bit-oriented data formats (in storage bytes).

EQU – EQUATE

The EQU statement assigns values and attributes to a symbol. The format of the EQU statement is:

Name	Operation	Operand
Ordinary or variable symbol	EQU	Expression

The expression in the operand field can be absolute or relocatable; however, all symbols 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. When that term is a location counter reference (*) or an arithmetic constant, the length attribute is 1. The value attribute of the symbol is the value of the expression. When the newly defined symbol is referenced in later statements, it has all the attributes assigned by the EQU statement.

The EQU statement can equate symbols to register numbers, immediate data or other arbitrary values, as shown in the first two examples of Figure 8-1.

The EQU statement can also equate symbols to frequently used or complex expressions, so that the programmer can use the symbol rather than an entire expression, as shown in the last two examples of Figure 8-1. Note that all symbols in the expression must be previously defined.

```
NAME                OPERATION          OPERAND
RØ                  EQU                Ø
* REGISTER NUMBER

MSG                 EQU                C'INCORRECT LABEL'
* IMMEDIATE DATA:CHARACTER CONSTANT

FREQ                EQU                B+25/C

CONF                EQU                ((8+B)/7*M)+85-NAM
```

Figure 8-1. Examples of EQU Statements

WDD AND BDD – WORD AND BYTE DEFINED DATA

The WDD and BDD statements define aligned data in storage. The WDD statement aligns the data constant defined in the operand at word boundaries; the BDD statement aligns data at byte boundaries. Both statements have the same basic format:

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

The name field reflects the address after alignment.

Operands must be separated by commas and have the format:

a

a(b,c)

a(b)

a(,c)

Where:

a = A data value to be generated (required); any valid expression. If the data value is a forward symbolic reference, and no explicit size is coded, it is assumed to be a two-byte value.

b = A length specification (in bytes); a positive predefined absolute expression. If omitted, the length specification is the implied size of the expression. If the explicit size of a forward symbolic reference is one byte, and if the symbol resolves to an arithmetic or string of more than one byte, a warning diagnostic is issued.

c = A repetition factor; a positive predefined absolute expression. If omitted, a repetition of one is assumed.

If the length specification or the repetition factor is zero, no data is generated, but the location counter is aligned on the specified word or byte boundary.

Examples of WDD and BDD statements are shown in Figure 8-2. In the last example, if M = N, the term 35 is generated. If M ≠ N, no data is generated, but the location counter is set on a word address, which is given the name NAM4.

```
NAME      OPERATION   OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
NAM1      WDD         C'VALUE IS'(8)
NAM2      BDD         =X'24B'
NAM3      BDD         7(,B+1)
NAM4      WDD         35(2,M<EQ>N)
```

Figure 8-2. Examples of WDD and BDD Statements

The value attribute of the symbol in the name field is the address of the leftmost byte after alignment. The length attribute is the length in bytes (specified or implied) of the first (or only) data field in the operand.

Omitted operands, signified by a comma without a data value, indicate a zero byte or word. The last data value in a string of multiple operands must be a specified data value, not an omitted operand.

Consider the following example. Notice that an arithmetic constant, such as 45, uses two bytes.

Statement	Generated Data	
WDD ,12,,,45	00 00	Word 1
	00 0C	Word 2
	00 00	Word 3
	00 00	Word 4
	00 2D	Word 5

If the data value is a relocatable expression other than $SYSEG, the length specification and the repetition factor have the following restrictions:

1. The length attribute must be resolved to two bytes. If the length is not specified, two bytes are assumed.

2. Alignment must be on a word boundary.

For $SYSEG, the length attribute is 1 and alignment must be on an odd boundary.

Literals (which are always relocatable) in a WDD or BDD statement require special consideration. If a literal term is used in the WDD or BDD statement, the implied length and repetition attributes are (2,1). If other specifications are included, they refer to the literal term itself, but not to the symbol defined in the name field. Consider this statement:

```
NAME      OPERATION      OPERAND
B1        WDD            =C'ABCD'(5,2)
```

The length attribute of the literal C'ABCD'(5,2) is 10; but the length attribute of the symbol B1 is 2, because the operand is a relocatable term. In all other cases, the symbol in the name field receives the length attribute of the first data field in the operand.

If the location counter (*) is referenced in the operand field, the value attribute of the symbol in the name field replaces the operand. For example, TAG WDD *,*,* generates three words of data, each assigned the value attribute of TAG. If TAG is equal to to 1004, then 1004 1004 1004 is generated.

For example, the following statement specifies that two 8-byte fields of all zeros are to be generated. TAG represents the address of the first byte of generated data.

```
NAME      OPERATION      OPERAND
TAG       WDD            Ø(4,2),Ø(4,2)
```

If the location counter is pointing to 00FE, TAG is equal to 00FE and the storage locations are as follows. (The last byte is 010D or TAG+15.)

Locations	Contents	
00FE	00 00	First operand
0100	00 00	
0102	00 00	
0104	00 00	
0106	00 00	Second operand
0108	00 00	
010A	00 00	
010C	00 00	

If the location counter is pointing to an odd-byte address when a WDD statement is encountered, the assembler automatically updates the counter to the next word boundary and does not affect the contents of the odd-byte address.

In the preceding example, if the location counter is pointing to 00FD, the assembler updates the counter to 00FE and the contents of 00FD are unchanged. The storage locations are as follows.

Locations	Contents	
00FC	xx xx	xx indicates that the contents are
00FE	00 00	residual data
0100	00 00	
0102	00 00	
.	. .	
.	. .	
.	. .	
.	. .	
010C	00 00	

TAG, then is equal to 00FE. However, if TAG BDD 0(4,2), 0(4,2) is specified rather than TAG WDD 0(4,2),0(4,2) and the location counter points to 00FD, the first zero byte is stored at address 00FD and the last byte at address 010C. TAG is equal to 00FD.

If multiple operands are specified, each operand is word-aligned for the WDD statement, or byte-aligned for the BDD statement, as in the example:

Statement	Generated Data	
WDD TAG1,X'13',X'05',TAG2	TAG1	Word 1
	13 xx	Word 2
	05 xx	Word 3
	TAG2	Word 4

Where:

TAG1 = a two-byte relocatable tag

xx = contents are residual data

TAG2 = a two-byte relocatable tag

WRS AND BRS – WORD AND BYTE RESERVE STORAGE

The WRS and BRS statements reserve storage without preset data. The format of the WRS and BRS statement is:

Name	Operation	Operand
Any symbol or blank	WRS or BRS	Absolute arithmetic expression

The symbol in the name field is the address of the first byte, and has the length attribute (number of bytes) of the storage area.

The operand of the WRS statement specifies the number of words to reserve; the operand of the BRS statement specifies the number of bytes. Symbols used in the operand field must be previously defined, and when evaluated must equal a positive, absolute arithmetic value. If the operand is zero, the location counter is aligned on the specified word or byte boundary.

In the following example, WRS and BRS each reserve 800 bytes of storage. The first byte of the WBUFF area begins on a word boundary, while no distinction between odd or even address bytes is made for BBUFF.

```
NAME      OPERATION      OPERAND
BBUFF     BRS            800
WBUFF     WRS            400
```

In Figure 8-3 the example shows the use of an ORG statement in conjunction with WDD and BRS. After the second ORG statement, the value of TAG1(29,1) is: CONTENTS OF ABCD ARE INVALID.

```
NAME      OPERATION      OPERAND
TAG1      WDD            C'CONTENTS OF'
TAG2      BRS            5
TAG3      WDD            C'ARE INVALID.'
                  .
                  .
                  .
          ORG            TAG2      RETURNS CTR TO TAG2
          WDD            C'ABCD'   INSERTS CHAR'S AT TAG2
          ORG                      CTR SET TO PREVIOUS HIGH
                  .
                  .
                  .
          END
```

Figure 8-3. Example of an ORG Statement with WDD and BRS

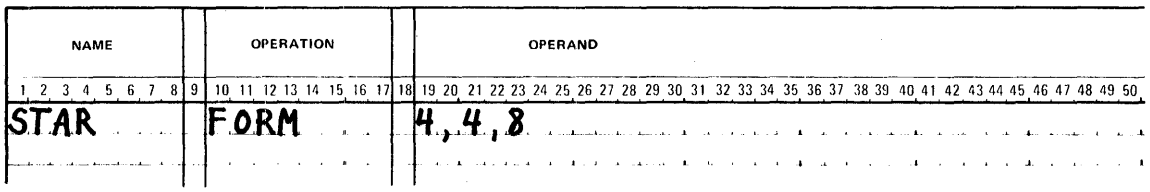
FORM – DEFINE DATA FORMAT

The FORM *definition* statement defines a symbolic name to be used as a mnemonic in the operation field of a subsequent statement, and specifies the size (in bits) and storage alignment of each operand to be used with the mnemonic. The format of the FORM definition statement is:

Name	Operation	Operand
Ordinary symbol	FORM	One or more positive arithmetic expressions separated by commas

The name field which is required, defines the mnemonic operation code for a FORM instruction statement. Expressions in the operand field must equal a number between 1 and 255. Symbols in the operand field must be previously defined.

The FORM definition statement in the following example defines a mnemonic, STAR, and specifies that its first operand is assigned four bit positions; the second, four also; and the third, eight bit positions.



FORM – INSTRUCTION STATEMENT

The FORM *instruction* statement specifies the data to be generated according to the format defined by the corresponding FORM definition statement. The format of the FORM instruction statement is:

Name	Operation	Operand
Any symbol or blank	FORM name	Exp,exp,. . .,exp

The FORM name in the operation field must be previously defined in a FORM definition statement.

The operand field may contain any valid expressions, separated by commas. No alignment is performed before data generation. The values specified in the operand field of the FORM instruction are matched by position to the fields defined in the operand of the corresponding FORM definition statement.

Missing operands (signified by contiguous commas) are replaced with zeros. If the number of operands in the instruction does not match the number specified in the definition, an error message is generated.

If the symbol in the name field of a FORM definition is a mnemonic used in more than one type of instruction, the assembler assigns attributes to the instruction according to the following hierarchy:

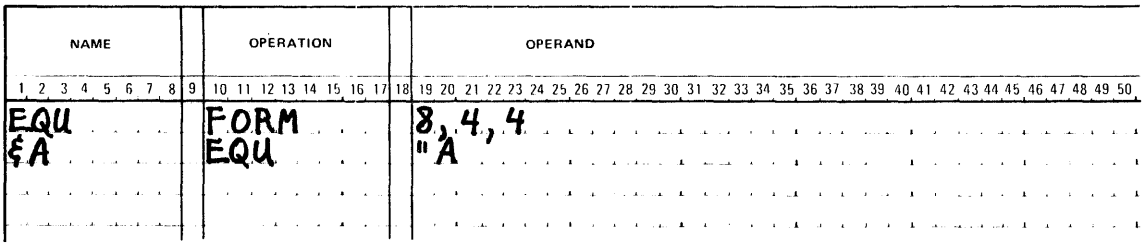
1. Machine and assembler instructions.

2. User macros and user FORM instructions within user macros.

3. User FORM instructions outside macros.

4. System macros and FORM instructions within system macros.

For example, if the programmer codes the following statements, the assembler treats the statement &A EQU "A as an assembly language statement (1. above) rather than a FORM instruction.



If the value of an expression is relocatable and not $SYSEG, the following conditions must be met; or a relocation error occurs and the expression is made absolute:

1. The size of the corresponding FORM definition field must be 16 bits in length.

2. The field must begin on a word boundary.

If the operands of a FORM instruction statement do not use a complete byte, the remainder of the byte is unchanged. For example, the FORM definition statement, SIGN FORM 1,5,3 specifies that 9 bit positions are required for the operands. When a FORM instruction, such as .A1 SIGN A<EQ>B,"B,0 calls this definition, the assembler uses two full bytes, but the last 7 bit positions of the second byte are zeros.

If $SYSEG is used as an operand in a FORM instruction, the size of the corresponding FORM definition field must be 8 bits and start on a byte boundary.

PADDING AND TRUNCATION RULES FOR FORM STATEMENTS

Padding and truncation is done according to the following rules:

1. Hexadecimal values are right-justified with zero fill on the left. If the actual data is larger than the defined field, the data is truncated on the left.

2. Alphanumeric character constants are left-justified with blank fill on the right. If the actual data is larger than the defined field, the data is truncated on the right.

3. Packed decimal values are right-justified with zero fill on the left. If the actual data is larger than the defined field, the data is truncated on the left.

4. Zoned decimal values are right-justified with zero fill on the left. If the actual data is larger than the defined field, the data is truncated on the left.

5. Integer string values and arithmetic values are right- justified with the sign propagated on the left. If the actual data is larger than the defined field, the data is truncated on the left and the sign is lost.

6. Floating point values are expanded with zero fill on the right. If the actual data is larger than the defined field, the value is truncated on the right.

The examples in Figure 8-4 illustrate certain padding and truncation rules. The first operand of TAB1, a hexadecimal 5 (0101), is truncated on the left and the two rightmost bits (01) are inserted in the 2-bit field defined by 2 in the FORM definition statement. If the value attribute of B in the second operand is less than C, the single bit position established in the corresponding definition statement is set to a binary 1. If B is greater than C, the bit is set to a binary 0.

The first A character constant (1100 0001 in EBCDIC) in the second TABLE statement is truncated on the right, and the remaining two leftmost bits (11) are assigned to location TAB2. Note, however, that all eight bits (1100 0001) of the second A character constant in the fourth operand position are retained at location TAB2+1.

```
NAME      OPERATION   OPERAND
1 2 3 4 5 6 7 8 | 9 | 10 11 12 13 14 15 16 17 | 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
TABLE     FORM      2,1,5,8,16,8
TAB1      TABLE     "5,B<LT>C,N+1,"FØ,ADDR,BYTE
TAB2      TABLE     C'A',,,C'A'
```

Figure 8-4. Examples of Padding and Truncation for Form Statements

The following examples show a possible use of the FORM statement — redefining instructions to create a new language closer to English. In the first example, the "MOVE" instruction generates a MOVM machine instruction, using a FORM statement and a series of Equates. This corresponds to: MOVM BUFFERB(R2),@BUFFERA(R3). Assuming that BUFFERA is at address 63FA, the code generated is:

602B

63FA

63FC

NAME	OPERATION	OPERAND
DIRECT	EQU	Ø
INDIRECT	EQU	1
R2	EQU	2
R3	EQU	3
MEMORY	EQU	X'6Ø'
MOVE	FORM	8,1,3,1,3,16,16
BUFFERA	WDD	C' '(2)
BUFFERB	WDD	C'5Ø'
	MOVE	MEMORY,DIRECT,R2,INDIRECT,R3,;
		BUFFERB,BUFFERA

The next example shows a BRANCH FORM statement used alone to generate a BR machine instruction, or together with ROUTINE to generate a BSR machine instruction. Assuming that SUBROUT1 is at 2F3A, the code generated is:

EA07

2F3A

The last statement of this example, BRANCH REGISTER,R7, generates the code: EB07.

NAME	OPERATION	OPERAND
SAVERTN	EQU	X'EA'
R7	EQU	7
REGISTER	EQU	X'EB'
BRANCH	FORM	8,8
ROUTINE	FORM	16
	BRANCH	SAVERTN,R7
	ROUTINE	SUBROUT1
SUBROUT1	EQU	*
	BRANCH	REGISTER,R7

A detailed example of the FORM instruction can be found in Appendix F.

9. LISTING CONTROL STATEMENTS

The listing control statements control the printing of the lines of code generated by the assembler. The statements themselves are used only for the source listing and are not carried over to the object program.

The assembler mnemonics and functions of the listing control statements are:

TITLE Identifies the listing.

EJECT Starts a new page.

SPACE Inserts blank lines.

PRINT Specifies the details to be printed.

TITLE, SPACE, and EJECT statements do not appear in the source listing.

TITLE – IDENTIFY LISTING

The TITLE statement specifies the program ID and the heading to be printed on each page of the listing. The format of the TITLE statement is:

Name	Operation	Operand
Symbol or blank	TITLE	Character string constant

The first symbol (except a sequence symbol) in the name field of any TITLE statement is printed in the program ID area at the top of each page of the entire listing. The name field of all other TITLE statements must contain a sequence symbol or a blank.

The character string in the operand field specifies the heading for each page. Any variable symbols in the character string are resolved. The resolved character string must not be more than 90 characters.

When the TITLE statement appears in a source program, the assembler ejects the current page and prints the specified title on the top of the next page. This title is printed on the top of each page until another TITLE statement appears.

In the following example, PGM1 and FIRST SUBROUTINE is printed on all sheets of the listing until another TITLE statement appears:

NAME	OPERATION	OPERAND
PGM1	TITLE	C'FIRST SUBROUTINE'

Assume that this heading has been printed on six sheets, and the following TITLE statement is encountered while the sixth sheet is being printed.

NAME	OPERATION	OPERAND
.SECD	TITLE	C'SECOND SUBROUTINE'

The assembler halts the printing and ejects the sixth sheet. The new heading, SECOND SUBROUTINE, is printed on the seventh sheet. Printing then continues with the seventh sheet. Note that a sequence symbol in the name field does not change the program ID.

EJECT – START NEW PAGE

The EJECT statement ejects the remainder of the page and resumes the printing at the top of the next page. The format of the EJECT statement is:

Name	Operation	Operand
Sequence symbol or blank	EJECT	Not used – ignored by the assembler

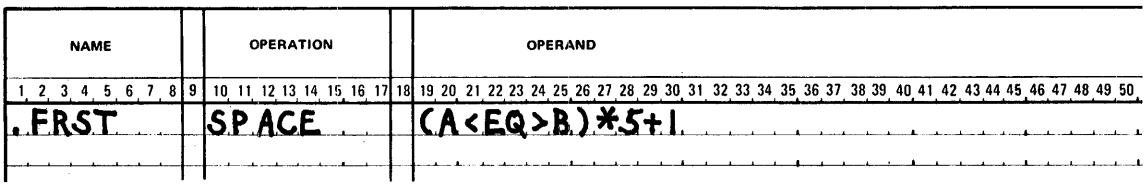
The EJECT statement can be used to separate routines in a program listing. Two successive EJECT statements leave the remainder of the current sheet plus the entire next sheet blank. A TITLE statement immediately followed by an EJECT statement produces a page that is blank except for the heading specified in the TITLE statement.

SPACE – INSERT BLANK LINES

The SPACE statement inserts one or more blank lines in a listing. The format of the SPACE statement is:

Name	Operation	Operand
Sequence symbol or blank	SPACE	Absolute arithmetic expression

The assembler evaluates the expression in the operand field to determine the number of lines to leave blank. Symbols in the expression must be previously defined. If the number of lines to be left blank exceeds the number left on the current page, the SPACE statement functions as an EJECT statement, and the next line is printed on the top of the next page. An example of a SPACE statement is:



PRINT – PRINT OPTIONAL DATA

The PRINT statement controls the extent of the printing. The PRINT statement cannot appear within a macro. The format of the PRINT statement is:

Name	Operation	Operand
Sequence symbol or blank	PRINT	One to four operands separated by commas

The operands, which may appear in any order, specify the amount of data to be printed. The two choices for each operand are:

1. ON — A listing is printed.
   OFF — No listing is printed.

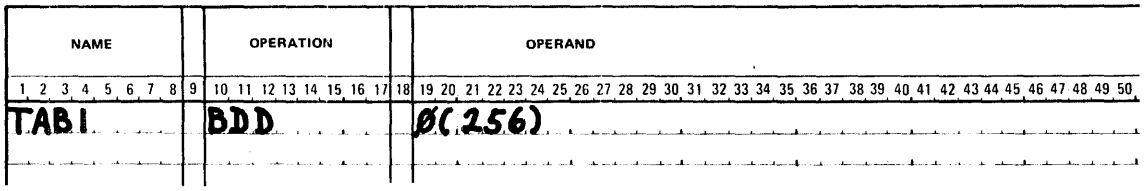
2. GEN — All statements generated from a macro instruction are printed.
   NOGEN — Only the macro instruction is printed.

3. DATA — Generated data is printed in full.
   NODATA — Only the leftmost eight bytes of generated data are printed.

4. COND — Conditional statements are printed.
   NOCOND — Conditional statements are not printed.

The operand cannot contain a variable symbol. Each condition specified by the operands remains in force until it is changed by another PRINT statement. A program can contain any number of PRINT statements. If a PRINT statement does not appear in a program, or until it does appear, the assembler assumes the following conditions: PRINT ON,NODATA,GEN,NOCOND.

For example, the following statement requests the assembler to assemble 256 bytes of zeros:



If the operand DATA is included in the last PRINT statement, the listing contains all 256 bytes of zeros. If the last PRINT statement included the operand NODATA, only eight bytes of zeros are listed.

If the operands of a PRINT statement are contradictory (such as OFF and DATA), the assembler determines the printing according to the following priority:

1. ON or OFF
2. GEN or NOGEN
3. DATA or NODATA, and COND or NOCOND

When NOCOND is specified, conditional statements with errors are printed unless OFF is in force, or the statement occurs in a macro when NOGEN is in force.

Note that the line numbers generated will not be in sequence when NOGEN or NOCOND is specified. Jumps in the line numbers indicate that statements were not listed due to NOGEN or NOCOND. However, if the suppressed statements contain errors, the error messages still refer to lines which do not appear in the listing.

10. MACRO LANGUAGE AND CONDITIONAL ASSEMBLY STATEMENTS

The macro language and conditional assembly statements are so closely related that it is difficult to explain either concept by itself. Thus, this section is divided into two subsections. The first explains the basic structure and application of the macro language. Sufficient information is supplied to write a simple macro definition and instruction. The second subsection deals with specific conditional assembly statements and their use within more complex macros and assemblies.

MACRO LANGUAGE

The macro language provides a convenient way to generate a desired sequence of assembler statements many times in one or more programs. The principal features are the macro definition and the macro instruction. The macro definition is a composite piece of coding which serves as a prototype for generating source statements. The macro instruction is a single statement which calls the macro definition for assembly and assigns values to the variable symbols in the macro definition.

The macro definition is written only once; the macro instruction is written each time a programmer wants to generate the desired sequence of statements. This facility can help simplify the coding of a program and reduce the chance of coding errors.

Macro definitions must appear in a source program before all PUNCH statements and statements pertaining to the first control section; consequently, only EJECT, PRINT, SPACE, TITLE, ICTL, ISEQ, and comment statements can validly precede the first macro definition. All of these statements except ICTL can appear between macro definitions.

A macro definition cannot appear within a macro definition; however, one macro can call another macro.

MACRO DEFINITION

A macro definition has four parts:

1. Header statement
2. Prototype statement
3. Model statements
4. Termination statement

HEADER STATEMENT

The header statement identifies the beginning of a macro definition. It must be the first statement of a macro definition. The format of the header statement is:

Name	Operation	Operand
Blank	MACRO	Blank — ignored by the assembler

PROTOTYPE STATEMENT

The prototype statement specifies the mnemonic operation code and the format of macro instructions that call the macro definition. The prototype statement must be the second statement of every macro definition. The format of the prototype statement is:

Name	Operation	Operand
Symbolic parameter or blank	Mnemonic operation code	0 — 35 symbolic parameters

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

The mnemonic operation code in the operation field is the macro name used to call the macro definition for assembly. This code must not be used in another macro definition, nor can it be a recognized mnemonic of a machine or assembler instruction.

The operand field may contain 0 through 35 symbolic parameters, separated by commas. The first four characters of a symbolic parameter should not be &SYS. Comments can appear only if symbolic parameters are present.

The symbolic parameters in the name field and the operand field represent variable values that are supplied by the programmer when he calls the macro for assembly. Subsequent model statements use these symbolic parameters.

The operands in a prototype statement can be positional or keyword. Keyword and positional operands cannot both be used in the same prototype statement.

Positional operands require that the operands of a macro instruction be written in the same order as the corresponding symbolic parameters of the prototype statement. Positional operands cannot have a default value and must begin with an ampersand (&) followed by one to seven alphanumeric characters, the first of which must be alphabetic. Examples of positional operands are &PAR1, &P, &P2.

Keyword operands may appear in any order in the macro definition and the macro instruction, because the parameters are recognized by the keyword, not by the order or position of the symbolic parameter in the prototype statement. Keyword operands may be assigned default values. A default value must be a standard value, not a variable symbol. Keyword operands are similar to positional operands, except that keyword operands are immediately followed by an equal sign and optionally followed by a standard value. If a value is not assigned, the default value is null. Examples of keyword operands are: &PAM1=22,&P=C'ABCD',&PAM3=.

The length of a prototype statement can be any number of lines. Continued lines must end with a semicolon. A semicolon may not be the first character of a continuation line.

MODEL STATEMENTS

Model statements are the macro definition statements from which the assembler language statements are generated. Zero or more model statements may follow the prototype statement. In the use of special characters, model statements must follow the same rules as macro instruction operands. The rules pertaining to special characters in macro instructions are discussed under *Special Characters in a Macro Instruction* later in this chapter. Model statements must also follow the normal continuation line rules, and statements generated from model statements must not require more than 160 characters. Only generated statements appear in the listing. The format of the model statement is:

Name	Operation	Operand
Any symbol or blank	Instruction or variable symbol	Any symbols or terms

The name field may be blank, or may contain a symbol. Because a sequence symbol inside a macro definition is local to that definition, the same sequence symbol can be used in another macro or outside the macro definition. However, within the same macro definition, a sequence symbol can be generated only once. Thus, a sequence symbol may be used in more than one model statement, provided that the statements which duplicate it are skipped due to conditional assembly. Note that because prototype statements are not generated, the symbol in the name field of a prototype statement can be duplicated and generated in a model statement. The characters * and . * or a sequence symbol cannot be substituted for a variable symbol in the begin position of a model statement. If the model statement is an inner macro instruction, the name field must follow the rules for macro instructions.

A variable symbol can be concatenated with other characters in the name field.

The operation field may contain a machine instruction, an assembler instruction, a macro instruction, a form instruction, or a variable symbol. However, the following assembler instruction mnemonics cannot be used in the operation field of a model statement: MACRO, PRINT, ISEQ, ICTL, and END. Variable symbols cannot be used to generate an ADO, AGO, ANOP, SETA, SETC, PUNCH, MEXIT, GBLA, GBLC, MNOTE, CSECT, COM, MACRO, SEG, PRINT, ISEQ, ICTL, or END mnemonic, or macro instruction mnemonic operation codes.

The operand field may contain ordinary, variable, or sequence symbols, or other terms, depending upon the instruction in the operation field. A symbolic parameter in the operand field of a model statement must first be defined in the prototype statement. Comments can appear after the last operand. No substitution is made for variable symbols in a comment.

A model statement may be a comment statement. An asterisk in the begin column indicates that the entire line is a comment statement. The assembler converts a model comment statement into an assembler language comment statement.

The programmer may also write comment 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 comment (.*comment).

The line following a PUNCH model statement is the only exception to model statement format rules. Format rules do not apply because the line contains output data, and all characters between the begin and end positions are in free format. Substitution is performed for all variable symbols between the begin and end positions.

Relationship of Model and Prototype Statements: Figure 10-1 illustrates the relationship between model statements and the prototype statement. Notice that the symbolic parameters &TAG1, &TAG2, &TAG3, and &TAG4 are defined in the prototype statement before they are used in the model statements.

```
NAME                OPERATION           OPERAND
1 2 3 4 5 6 7 8 9   10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
                    MACRO
&BEGIN              DIVIDE              &TAG1,&TAG2,&TAG3,&TAG4
&BEGIN              STO                 SAVE,5
                    LOD                 &TAG1,4
                    LOD                 &TAG2,5
                    DIV                 &TAG3,5
                    STO                 &TAG4,5
                    LOD                 SAVE,5
                    MEND
```

Figure 10-1. Macro Definition

TERMINATION STATEMENT

The termination statement specifies the end of a macro definition. It can appear only once within a macro definition and must be the last statement of the definition. The format of the termination statement is:

Name	Operation	Operand
Sequence symbol or blank	MEND	Blank — ignored by the assembler

MACRO INSTRUCTION

The macro instruction performs the following functions:

1. Calls the macro definition for assembly.

2. Assigns values to the symbolic parameters in the macro definition.

The macro instruction closely parallels the prototype statement. The values in the macro instruction are equated to the symbolic parameters in the prototype statement. The format of the macro instruction is:

Name	Operation	Operand
Ordinary symbol, sequence symbol, or blank	Mnemonic operation code	0 — 35 operands

The name field may be blank, or it may contain an ordinary symbol or a sequence symbol. An ordinary symbol is defined in the assembly only if a symbolic parameter is both in the name field of the prototype statement and in the name field of a model statement. If the model statement which has the symbolic parameter in its name field is skipped as a result of conditional assembly, the statement is not generated. Consequently, the ordinary symbol in the name field of the macro instruction is also not generated.

A sequence symbol in the name field of the macro instruction is never carried over to the name field of the generated model statement. This sequence symbol can only be used in the operand field of an AGO statement -- never as the second operand of an ADO statement.

The mnemonic operation code in the operand field must be previously defined in a macro definition or in the system macro library.

The number of operands in the operand field may not exceed the number of operands in the prototype statement. An operand can have up to 127 characters.

POSITIONAL OPERANDS

If the prototype statement has positional operands, the operands of the macro instruction correspond to the symbolic parameters in the prototype statement, and the operand values are applied to the symbolic parameters in sequence. If the macro instruction has fewer operands than the prototype statement, the unmatched symbolic parameters in the prototype are assigned null values, not blanks. Two contiguous commas in the operand field indicate an omitted operand, which is also assigned a null value. The examples in Figure 10-2 illustrate positional operands.

```
Prototype:   &LABEL   POS   &TAB1,&TAB2,&TAB3      Values:   &TAB1 = 42
Instruction: .FIRST   POS   42,15,63                         &TAB2 = 15
                                                             &TAB3 = 63
Instruction: .SEC     POS   16,,2                  Values:   &TAB1 = 16
                                                             &TAB2 = null
                                                             &TAB3 = 2
```

Figure 10-2. Macro Instruction -- Positional Operands

KEYWORD OPERANDS

If the prototype statement has keyword operands, the macro instruction must also have keyword operands. The instruction keyword, which is all characters before the equal sign, must directly match the keyword in the prototype operand, except that the instruction keyword operand is not preceded by an ampersand. The value following the equal sign in the macro instruction is assigned to its corresponding symbolic parameter. Symbolic parameters not matched by a macro instruction operand retain their default value. The examples in Figure 10-3 illustrate keyword operands.

```
Prototype:   &LBL    KEY   &PARM1=5,&PARM2=,&PARM3=C'VALUE'   Values:  &PARM1 = 100
Instruction: .THD    KEY   PARM3=C'PRICE',PARM1=100                    &PARM2 = null
                                                                       &PARM3 = C'PRICE'
Instruction: .FOUR   KEY   PARM1=ABCD,PARM2="7B                 Values:  &PARM1 = ABCD
                                                                       &PARM2 = "7B
                                                                       &PARM3 = C'VALUE'
```

Figure 10-3. Macro Instruction – Keyword Operands

SPECIAL CHARACTERS IN A MACRO INSTRUCTION

A macro instruction operand may consist of any combination of up to 127 characters, provided the syntactical rules for the following special characters are observed. These characters have special meanings in a macro instruction operand.

- escape character (#)
- ampersand
- apostrophe
- parentheses
- comma
- semicolon
- blank

Escape Character

The first character after an escape character retains its literal value. Therefore, if a special character is to retain its literal value, it must be preceded by an escape character. For example, the following operand field has three operands:

```
#(C#.B,#,#, ,XYZ
 1     2  3
```

Escape characters are part of the operand value and are carried over to the model statements.

Ampersand

An ampersand followed by a letter indicates the start of a variable symbol. During assembly, the current value is substituted for all variable symbols found in a macro instruction. In the following example, only &V1 is substituted.

```
C'A&V1#&V2',C'#&V3#&V4',AB+X'&V1
```

Apostrophe

An apostrophe immediately preceded by the letter C signifies the start of a character string constant. The character string constant extends to the next apostrophe. Parentheses, commas, and blanks lose their special meanings when they are enclosed in a character string constant, as in the example:

```
C'A,BC(   539'
```

The comma, blanks, and left parenthesis are part of the character string constant. In any other context, the apostrophe retains its literal value.

Parentheses

An operand beginning with a left parenthesis and ending with a right parenthesis signifies an operand sublist. Within a sublist, the comma is a suboperand separator, not an operand separator. Special characters, other than the comma, retain their special meanings within a sublist. Parentheses that do not specify an operand sublist retain their literal value. Examples of operands with parentheses are:

A,(B,C)	Two operands: [A] [(B,C)]
A(B,C)	One operand — operand does not begin with a left parenthesis
(A+B)/2(R,4)	One operand — operand does not begin and end with matching parentheses

Comma

The comma separates operands or suboperands, unless the comma is inside a character string constant, preceded by an escape character, or follows a left parenthesis that is not the opening parenthesis of a sublist. In these three cases, the comma retains its literal value. Examples of operands with commas are:

A,B,C	Three operands: [A] [B] [C]
A #,B #,C	One operand: comma preceded by #
B(C,4+K	One operand: comma preceded by (

Semicolon

The semicolon indicates line continuation. Characters following a semicolon are considered comments. The semicolon cannot be the first character of a continuation line. The current line continues with the first nonblank character of the next line, as in the following example:

A,BHCD,52,;HERE CON	HERE is a comment. The four operands are: [A] [BHCD] [52] and [CON].

Blank

A blank signifies the end of the operand field. Characters following a blank are considered comments.

SUBLISTS IN MACRO INSTRUCTIONS

To group a number of suboperands as a single symbolic parameter value, the macro instruction operand is written as a sublist. A sublist consists of one or more suboperands separated by commas and enclosed in parentheses. Each suboperand has form identical to an operand. The entire sublist including the parentheses is considered one macro instruction operand.

Omitted suboperands have a null value. However, the operand ( ) is considered a character string, not a sublist with all the suboperands omitted. The operand limit of 127 characters applies to the entire sublist. Examples of valid sublists are:

(A,2,4,16)	Four suboperands
(A+40,B(2,6),(N,M),240/C+X)	Four suboperands

SUBLISTS IN MODEL STATEMENTS

In a macro instruction operand, a sublist can assign a set of values to a single symbolic parameter of the macro definition. Any model statement may reference the entire symbolic parameter, but only a SETA, SETC, or ADO statement can reference the suboperands of the symbolic parameter.

The format used to reference a suboperand is: symbolic parameter (n).

The subscript n is an arithmetic set expression that refers to the position of the suboperand being referenced. For example, &LIST(2) references the second suboperand of the symbolic parameter, &LIST.

If the sublist (A,,C20,'192',(N,M)) is a macro instruction operand corresponding to the symbolic parameter &FIEL D, the values assigned to the suboperands are:

FIEL D(0) = null

FIELD(1) = A

FIELD(2) = null

FIELD(3) = C20

FIELD(4) = '192'

FIELD(5) = (N,M)

FIELD(6) through FIELD(n) = null

A subscripted reference to an omitted sublist element, such as FIELD(2) above, is assigned a null value. If the macro instruction operand is a simple operand, rather than a sublist, the macro instruction can refer to the operand value by the symbolic parameter without a subscript, or by the symbolic parameter with a subscript of 1. All other subscripted references have a null value.

Suboperands of sublists within a sublist (for example, N in FIELD(5) above) cannot be referenced in a model statement.

SUBSTRING NOTATION

Substring notation allows the programmer to reference part of a macro instruction operand, a SET symbol, or a character string constant. Substring notation consists of a variable symbol or a character string constant, immediately followed by two arithmetic set expressions separated by a comma and enclosed in parentheses, as in the examples:

&NAME(2,4)

C'AB24CFG'(&P1,2)

The first expression indicates the position of the first character to be included in the substring. The second expression indicates the number of consecutive characters (beginning with the character indicated by the first expression) to be included in the substring.

If the substring specifies more characters than are in the macro instruction operand or in the character string constant, only the number of available characters are supplied. If the starting character is outside the range of the string, a null character is supplied.

Substring notation has the following limitations:

1. Substrings can only be specified for character string constants, macro instruction operands, and SET symbols.

2. A substring of a sublist cannot be specified.

3. A substring reference can only be used as a term in a conditional assembly statement.

Consider the example in Figure 10-4.

```
Prototype:   &LABEL    WRITE   &P1,&P2,&P3          Values:   &P1 = 24
Instruction: .FIRST    WRITE   24,C'ABCDE',734A               &P2(3,3) = ABC
                                                              &P3(1,2) = 73
```

Figure 10-4. Examples of Substring Notation

CONCATENATION OF VARIABLE SYMBOLS

Variable symbols in model statements can be concatenated so that the name field, operation field, or operand field can be modified by a macro instruction. Thus, the mnemonic instruction DIV can become DIVA, DIVB, or DIVC, and the operand FIELD can become FIELDA, FIELDB, or FIELDC.

The example in Figure 10-5 illustrates concatenated operation codes and operands in a macro definition.

```
NAME      OPERATION   OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

          MACRO DEFINITION
                    MACRO
&BEGIN    DIVIDE    &T1,&T2,&T3,&T4,&T5,&T6,&T7
&BEGIN    STO       SAVE,5
          LOD       &T1&T2,4
          LOD       &T1&T3,5
          DIV&T7    &T5&T4,5
          STO       &T1&T6,5
          LOD       SAVE,5
          MEND
          MACRO INSTRUCTION
HERE      DIVIDE    FIELD,B,D,3,,A,R
          MACRO EXPANSION
HERE      STO       SAVE,5
          LOD       FIELDB,4
          LOD       FIELDD,5
          DIVR      3,5
          STO       FIELDA,5
          LOD       SAVE,5
```

Figure 10-5. Concatenation of Variable Symbols

NESTING OF MACROS

A model statement in a macro definition may be a macro instruction that calls another macro. Model statements used as macro instructions are called inner macro instructions. Macro instructions not used as model statements are called outer macro instructions.

Rules for inner macro instructions are the same as for outer macro instructions. Symbolic parameters in an inner macro instruction are replaced by the corresponding characters of the outer macro instructions. Sublists of an outer macro instruction cannot be passed as a sublist to an inner macro instruction.

An outer macro instruction is a first level macro instruction. The first inner macro instruction is a second level instruction, the second inner macro instruction is a third level, and so on. Five levels of macro instructions are allowed. Within each level, any number of macro instructions may be used.

Figure 10-6 demonstrates the inner and outer macro instructions and the various levels of instructions.

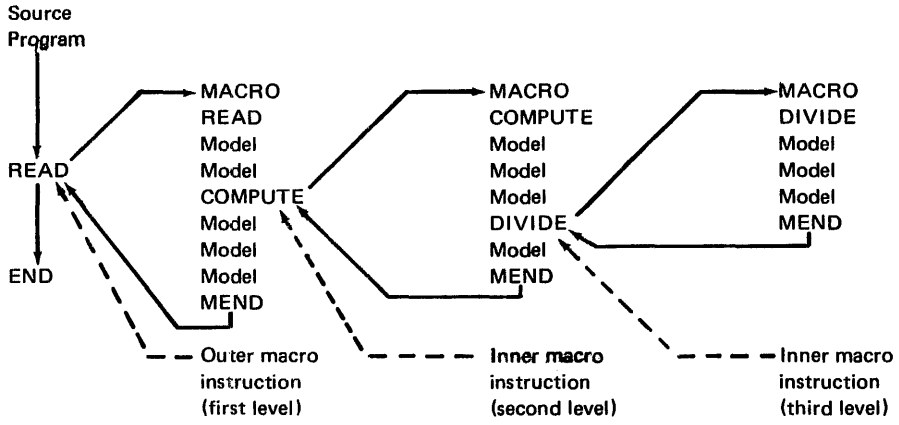


Figure 10-6. Nesting of Macros

MNOTE – GENERATE ERROR MESSAGE

The MNOTE statement is used only in a macro definition to generate an error message. The format of the MNOTE statement is:

Name	Operation	Operand
Sequence symbol or blank	MNOTE	Severity code, message or message only

The severity code may be W — warning error, or F — fatal error. If the first character is not a W or an F followed by a comma, the statement is treated as a warning error.

The message may be any EBCDIC characters. The line may not be continued, and no substitution is performed. Examples of MNOTE statements are:

Statement	Generated Message
MNOTE W,message	W MNOTE *message*
MNOTE F,message	F MNOTE *message*
MNOTE Fmessage	W MNOTE *Fmessage*
MNOTE message	W MNOTE *message*

MEXIT – ALTERNATE TERMINATION FOR MACRO DEFINITION

The MEXIT statement indicates an alternate termination point for a macro definition. The format of the MEXIT statement is:

Name	Operation	Operand
Sequence symbol or blank	MEXIT	Blank — ignored by the assembler

When an MEXIT statement is processed, the next statement processed by the assembler is the statement immediately following the macro instruction that called the macro. If MEXIT is skipped due to conditional assembly, MEXIT is ignored. The MEXIT statement cannot replace MEND as the final statement of a macro definition.

SYSTEM VARIABLE SYMBOLS – &SYSNDX AND &SYSECT

The system variable symbols, &SYSNDX and &SYSECT, are automatically assigned values by the assembler. These symbols can be used only in the name, operation, and operand fields of statements in macro definitions. They may not be defined as symbolic parameters, nor can they be assigned values by SET statements.

&SYSNDX

Assigned an original value of 0001 for the first macro instruction processed, &SYSNDX is increased by 1 for each inner or outer macro that is processed. Thus, the value of &SYSNDX represents the current number of macro calls processed up to and including the current call.

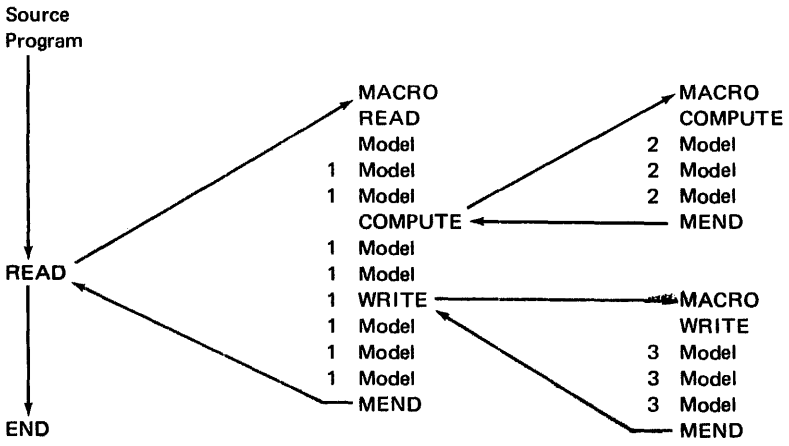
The value of &SYSNDX always remains constant within a macro level, regardless of how many inner macros are called. When the assembler returns to a given macro (from an inner macro), the value of &SYSNDX in effect is the value it had when the macro was first called.

Figure 10-7 illustrates the changing value of &SYSNDX. The numbers to the left of the model statements indicate the value of &SYSNDX as each statement is processed.

The variable symbol &SYSNDX can be concatenated with other characters to form unique names in statements generated from the same model statements. Thus duplicate names can be avoided when a macro is called more than once.

&SYSNDX as an arithmetic term in a SETA expression produces an arithmetic value. In other contexts, the value of &SYSNDX is a four digit number, including leading zeros.

The example in Figure 10-8 shows how &SYSNDX can be used. All statements in the example are part of the same macro, which is called by the eighth macro instruction processed during the assembly.



The numbers to the left of the statements indicate the value of &SYSNDX at that particular point in the program.

Figure 10-7. Using &SYSNDX with Inner and Outer Macros

Model Statements:

&SET1	SETA	&SYSNDX+10	Values:	&SYSNDX+10 = 8 + 10 = 18 (in the SETA statement)
&SET2	SETC	&SYSNDX		&SYSNDX = 0008 (in the SETC statement)
L&SYSNDX	FORM	2,6		L&SYSNDX = L0008 (in the concatenated symbol)

Figure 10-8. Examples of &SYSNDX

&SYSECT

The variable symbol &SYSECT represents the name of the control section in which a macro instruction appears. The value assigned to &SYSECT is the name of the last CSECT or COM statement that precedes the current macro instruction. If no CSECT or COM statement appears before the macro instruction, &SYSECT has the value of two blanks.

In any given macro definition, the value of &SYSECT is constant, that is, the name of the last CSECT or COM statement before the macro instruction. During nested macro calls, &SYSECT in an inner model statement refers to the last active CSECT or COM statement in the next outer macro definition. Note the value of &SYSECT in Figure 10-9.

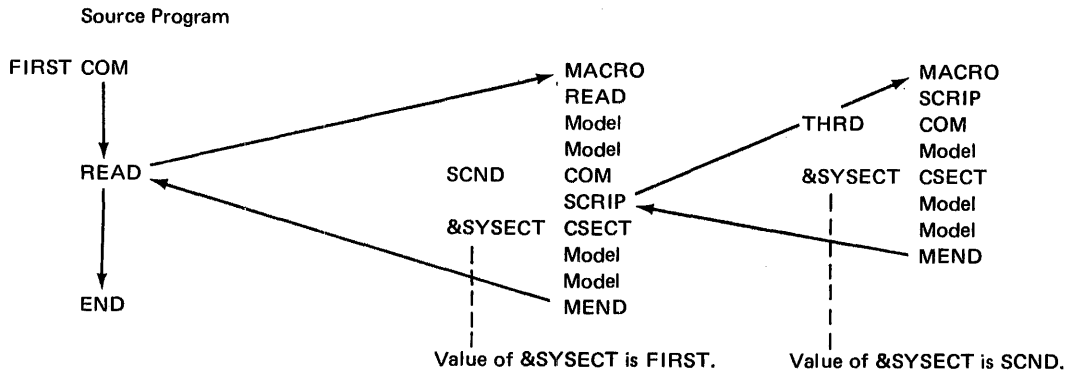


Figure 10-9. Example of &SYSECT

CONDITIONAL ASSEMBLY STATEMENTS

Conditional assembly allows the programmer to specify assembler language statements which may or may not be assembled, depending upon conditions evaluated at assembly time. These conditions are usually tests of values, which may be defined, set, changed, or tested during assembly. Thus, different sequences of statements can be generated from the same macro definition.

Almost all conditional assembly statements can be used inside or outside macros, although their primary use is inside macros. The macro language itself can be considered a type of conditional assembly.

The conditional assembly statements are:

SETA	Assigns arithmetic values to set symbols.
SETC	Assigns character values to set symbols.
GBLA	Defines a SETA statement as global.

GBLC	Defines a SETC statement as global.
ADO	Sets up a source statement generation loop.
AGO	Specifies a branch to another statement; skipped statements are not assembled.
ANOP	Specifies an assembly no-operation statement.

SET STATEMENTS

The SET statements assign arithmetic and character values to set symbols which can then be referenced in subsequent source statements. When the defined symbol appears in a subsequent statement, the assembler replaces the symbol with the assigned value.

If two SET statements assign different values to the same set symbol, the last value assigned to the symbol is the value currently in effect.

SET statements can appear within or outside a macro definition; however, a set symbol defined within a macro is local to that macro unless it is specifically declared global.

A set symbol defined outside a macro can be referenced for its assigned value from anywhere in the source program. The same variable symbol may not be used as a symbolic parameter and as a set symbol within the same macro definition; nor can the same variable symbol be used in a SETA and a SETC statement, if the symbols are defined within the same scope.

SETA – ASSIGN ARITHMETIC VALUE TO SET SYMBOL

The SETA statement assigns arithmetic values to set symbols. The format of the SETA statement is:

Name	Operation	Operand
Set symbol	SETA	Arithmetic set expression

The set symbol in the name field may not be generated as a result of substitution, that is, the name field must be explicitly coded.

The expression in the operand field is evaluated as a 16-bit arithmetic value which is assigned to the set symbol in the name field.

An arithmetic set expression may consist of one term or an arithmetic combination of terms. The procedure used to evaluate an arithmetic set expression is the same as that used to evaluate arithmetic expressions in assembler language statements. The only difference between the two expressions lies in the terms that are allowed. The terms that may be used in an arithmetic set expression are:

Arithmetic Terms	Example
Arithmetic constants	2463 or "F0F2
SETA symbols	&S1
&SYSNDX references	&SYSNDX
Count attribute references	K'&P1
Number attribute references	N'&P2

Character Terms	Example
Symbolic parameters, including sublist and substring references	&P3(&P1)
SETC symbols, including substring references	&S
Substrings of character constants	C'ABCDE'(2,3)
&SYSECT references	&SYSECT
Character constants	C'ABC'

Any character term except a character constant may appear as a single term, or be used as an operand in arithmetic and logical operations, provided the resultant character string contains only numeric characters. When used in this context, the value of the character string, that is, its decimal equivalent, may not exceed 65,535.

Any character term may be used as an operand in a relational operation. In this context, the term is always treated as a character constant, even if it consists solely of numeric characters.

Since conditional assembly statements represent the tools for source statement generation, they cannot themselves be the object of substitution, nor may they be concatenated. For example, if the character constant C'&P1' appears as a term in an arithmetic set expression, its value is &P1 regardless of the actual value of &P1.

The following are examples of *valid* SETA statements if &P1 and &P2 are symbolic parameters with values (64,4,ABC) and AB246C, respectively.

```
&S1     SETA    8
&S2     SETA    "4A
&S3     SETA    &S1+&S2
&S4     SETA    &P1(1)+&P2(&P1(2),2)
&S5     SETA    (&P1(3)<EQ>C'ABCDE'(1,3)+&P2(4,1))/2
&S6     SETA    &P2(3,3)<EQ>C'246'+N'&P1
```

Examples of *invalid* arithmetic set expressions are:

A+2	Reference to ordinary symbol
=C'AB'	Literal term
2+L'BETA	Length attribute reference
&P1(1)<EQ>64	&P1(1) is considered a character string, the second operand in the relation must be coded as a character term, that is, C'64'
P'-240'	String constant other than a character constant
C'ABC'	Character constant not used in a relation
&P1&P2	Concatenation of symbols

SETC – ASSIGN CHARACTER VALUE TO SET SYMBOL

The SETC statement assigns a character value to a set symbol. The assigned character value can be passed with the set symbol to another statement operand. The format of the SETC statement is:

Name	Operation	Operand
Set Symbol	SETC	Character term or arithmetic set expression

The set symbol in the name field may not be generated as a result of substitution, that is, the name field must be explicitly coded.

The operand field may contain an arithmetic set expression or one of the following character terms:

Term	Example
Symbolic parameter, including sublist and substring references	&PARM2(3)
SETC symbol, including substring references	&SET1
Substring of a character constant	C'ABCDE'(2,3)
&SYSECT reference	&SYSECT
Character constant	C'ABCD'
&SYSNDX reference	&SYSNDX

Only one operand is allowed. The maximum size of an assigned character value is 16 bytes. If a larger value appears, only the leftmost 16 bytes are assigned by the assembler.

When &SYSNDX is used as a single character term, the value is a string of four characters, including leading zeros. If the operand specified is an arithmetic set expression, the arithmetic value is converted to a 16-bit constant.

Examples of *valid* SETC operands are:

```
C'24BK'
&P1
&P1(2,4)
&P1(&P2(2,1))
&SETC
&SYSECT
&SYSNDX
(&SYSNDX+10-&P1(2,4)/2)*&P2(2)<EQ>C'FID'
```

Examples of *invalid* SETC operands are:

X'124'	String constant other than a character constant
C'AB2'+4	Character constant used as arithmetic operand
ALPHA	Ordinary symbol
L'ALPHA	Length attribute

GBLA AND GBLC – GLOBAL ARITHMETIC AND CHARACTER SET SYMBOLS

Local set symbols are made global (available outside the macro) by a GBLA or GBLC statement. The format of the GBLA or GBLC statement is:

Name	Operation	Operand
Blank	GBLA and GBLC	1-35 set symbols, separated by commas

Global statements must appear immediately after a prototype statement or after another global statement. Any number of continuation lines may be used.

When a set symbol is declared global, its assigned value is available to statements in the main program, but not to other macro definitions. To be available to other macro definitions, the set symbols must also be declared global in those macro definitions.

If set symbols have not been assigned values outside this macro definition or by a previous call to this definition, the global statement assigns an initial value of 0 to SETA symbols and a null character value to SETC symbols. If the set symbols have an assigned value, the global statements do not affect the value of the set symbols.

ADO – ITERATIVE RETURN

The ADO statement sets up a loop between the ADO statement and a subsequent statement identified in the ADO statement. The format of the ADO statement is:

Name	Operation	Operand
Sequence symbol, set symbol, or blank	ADO	Positive arithmetic set expression, sequence symbol

The sequence symbol in the name field can only be referenced by an AGO statement.

The operand field must have two operands, separated by a comma. The first operand is a positive arithmetic set expression that indicates the number of iterations to be executed. The second operand is a sequence symbol that specifies the last statement of the loop. This sequence symbol must appear in the name field of a statement that follows, not precedes, the ADO statement. The sequence symbol cannot be in the name field of another ADO statement, a macro instruction, or an AGO statement.

If the set expression is equal to zero, control is transferred to the statement named in the second operand, and no intervening code is included. However, if the first operand is a positive number, the assembler subtracts 1 from the first operand and includes in the assembly all the code from the ADO statement to the statement named in the second operand. If the first operand is zero after the subtraction, the assembler does not return to the ADO statement but processes the next statement. If the first operand is not zero after the subtraction, the assembler again subtracts 1 from the first operand and repeats the loop.

Notice that when the first operand initially is zero, the jump is made to the statement named in the second operand. When the first operand is not initially zero, the indicated number of loops is performed and a jump is made to the first statement after the statement named in the second operand. If the first operand is invalid, the value is set to 1.

The name field may contain a sequence symbol, a set symbol, or a blank. If the name field contains a set symbol, the symbol is initially assigned a value of 1. For each iteration, this value increases by one while the first operand of ADO decreases by one. Thus, in the following example, after 10 iterations, the first operand is 0 and &COUNT is 10.

```
NAME        OPERATION    OPERAND
&COUNT      ADO          10,..END
```

The preceding statement performs a function equivalent to the following three statements. During the first iteration, &COUNT equals 1 and the first operand (10) is decreased by one. After 10 iterations, &COUNT equals 10 and the first operand equals 0. Control is then transferred to the first statement following the .END statement.

```
NAME        OPERATION    OPERAND
&COUNT      SETA         0
            ADO          10,..END
&COUNT      SETA         &COUNT+1
```

The ADO statement can be used to include or exclude code from the assembly, depending upon the value of the first operand, as in the following example. If the variable symbol &LABEL is equal to YES, the first operand of the ADO statement is 1, and all intervening code is included in the assembly. If, however, &LABEL is not equal to YES, the first operand is zero and all intervening code is excluded.

```
NAME        OPERATION    OPERAND
&SETI       ADO          &LABEL<EQ>C'YES',..A1
.A1         ANOP
```

NESTING OF ADO STATEMENTS

As many as six ADO statements can occur within a primary ADO loop. Such nesting of ADO statements increases the total number of iterations in geometric progression. If five iterations are performed by an outer ADO loop, an inner ADO with five iterations increases the number of iterations to 25. An inner ADO statement cannot reference a statement outside one referenced by the outer ADO.

The following statements produce a 5 x 5 matrix containing all products of the ordinary numbers 1 through 5.

```
NAME      OPERATION      OPERAND
&COL      ADO            5,.END1
&ROW      ADO            5,.END
.END      WDD            &COL*&ROW
.END1     ANOP
```

During the first iteration of the outer ADO (named &COL), &COL equals 1 and &ROW increases by 1 with each iteration of the &ROW ADO statement until &ROW equals 5. During this first iteration, the statement .END WDD &COL *&ROW produces the values 1, 2, 3, 4, and 5, as shown in the first column of values in this matrix.

First Pass of the Outer ADO

		&COL
		1
&ROW	1	1
	2	2
	3	3
	4	4
	5	5

During the second iteration of the outer ADO, &COL equals 2 and &ROW again equals 1 through 5. The second column of values is produced.

Second Pass of the Outer ADO

		&COL	
		1	2
&ROW	1	1	2
	2	2	4
	3	3	6
	4	4	8
	5	5	10

This process continues until the outer ADO has performed five iterations, at which time the 5 x 5 matrix contains the following values:

		&COL				
		1	2	3	4	5
	1	1	2	3	4	5
	2	2	4	6	8	10
&ROW	3	3	6	9	12	15
	4	4	8	12	16	20
	5	5	10	15	20	25

AGO – UNCONDITIONAL BRANCH

The AGO statement transfers control to a statement named in its operand field. Statements between the AGO statement and the statement to which the jump is made are not included in the assembly. The format of the AGO statement is:

Name	Operation	Operand
Sequence symbol or blank	AGO	Sequence symbol

The sequence symbol in the operand field must appear in the name field of a statement following, not preceding, the AGO statement. An AGO statement can transfer control out of an ADO loop. If an AGO statement transfers control into the range of an ADO loop, the source statements are processed as if no ADO loop exists.

The AGO statement cannot jump into or out of a macro. An example of the AGO statement appears with the description of the ANOP statement in the following text.

ANOP – LABEL DEFINITION

The ANOP statement identifies a statement area to which a jump can be made. Because the ANOP statement is used for name field identification only, it has no operands and no operation is performed. The format of the ANOP statement is:

Name	Operation	Operand
Sequence symbol	ANOP	Not used – ignored by the assembler

If the programmer wants to use an ADO or AGO statement to branch to another statement, he must place a sequence symbol in the name field of the statement he wishes to branch to. However, if the name field already has an ordinary symbol or a variable symbol, a sequence symbol cannot be placed in the name field. To solve this problem, the programmer can place an ANOP statement before the statement he wishes to branch to, and branch to the ANOP

statement. This arrangement has the same effect as branching to the statement immediately after the ANOP statement.

In Figure 10-10, the AGO statement performs an unconditional branch to the ANOP statement. All intervening code is excluded from the assembly.

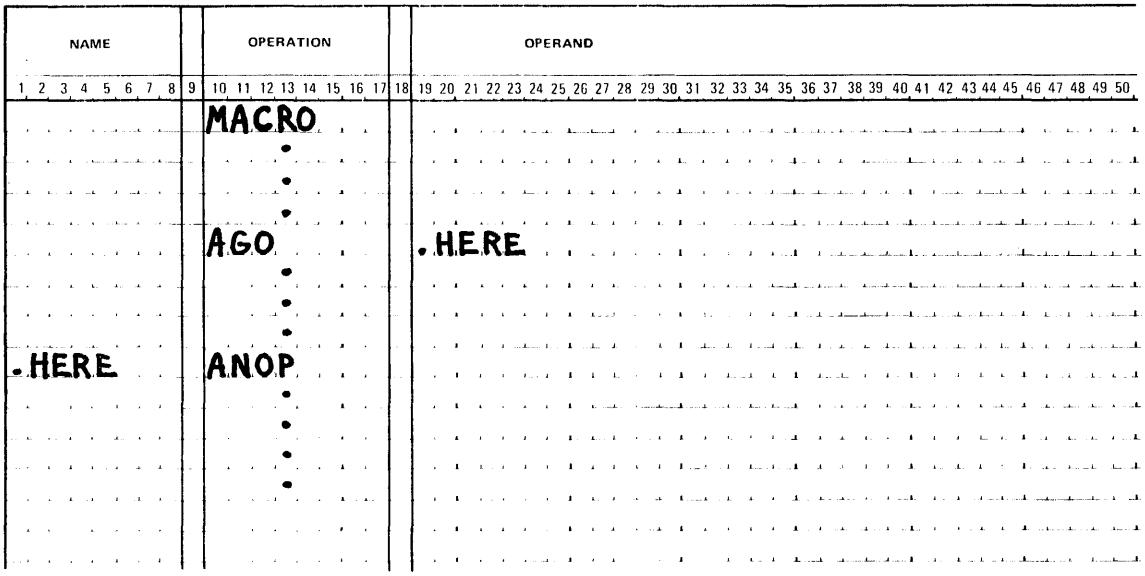


Figure 10-10. Example of the AGO Statement

COUNT AND NUMBER ATTRIBUTES

Count and number attributes are unique to macro instruction operands. These attributes can be referred to only in set expressions used as operands in conditional assembly statements within macro definitions.

COUNT ATTRIBUTE

The count attribute is a value equal to the number of characters in the macro instruction operand after substitution of variable symbols in the operand. All characters in the operand, including escape characters, are included in this count. If the operand is a sublist, the count attribute includes the embracing parentheses and the commas within the sublist. The count attribute of an omitted operand is zero.

If an operand contains variable symbols, the characters that replace the variable symbol, rather than the variable symbols, determine the count attribute.

The notation for the count attribute is K' immediately followed by the symbolic parameter that corresponds to the operand.

The examples in Figure 10-11 illustrate the preceding rules.

Symbolic Parameter	Macro Instruction Operand	Count Attribute	Count Value
&PAR	ALPHA	K'&PAR	5
&PAR1	(JUNE,JULY,AUGUST)	K'&PAR1	18
&PAR2	2(10,12)	K'&PAR2	8
&PAR3	C'AB##3'	K'&PAR3	9

Figure 10-11 Examples of the Count Attribute

To reference the count attribute of a suboperand, K' followed immediately by a subscripted symbolic parameter must be used, as in the example: K'&PAR(3). K'&PAR(3) refers to the count attribute of the third suboperand of the symbolic parameter &PAR.

NUMBER ATTRIBUTE

The number attribute is a value equal to the number of suboperands in an operand sublist. If the operand in the macro instruction is not a sublist, the value of the number attribute is one. The number attribute of an omitted operand is zero.

The notation for the number attribute is N' immediately followed by the symbolic parameter that corresponds to the operand, as in the example: N'&PAR.

In the following macro instruction, the count attribute of the operand is 19, while the number attribute is 3:

```
NAME      OPERATION      OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
          REPRO          (INPUT,OUTPUT,LIST)
```

The following example has three operands, each with a number attribute of 1; their count attributes, however, are 5, 6, and 4, respectively.

```
NAME      OPERATION      OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
          REPRO          INPUT,OUTPUT,LIST
```

11. CONTROL LANGUAGE STATEMENTS

The Control Language for the assembler must provide the following four basic services:

1. Call the assembler into execution
2. Specify the assembly options
3. Define source input, object output, source output, and macro library input files
4. Obtain the source program from the card reader, library member, or spooled input

The Control Language statements for the assembler are explained below. For the exact format of the statements, consult the **MRX/OS Control Language Services, Extended Reference** manual.

Control Language Statement	Parameter	Description
//EXEC PGM=	ASM	Calls the assembler into execution.
//PAR	Keyword options	Specifies the assembly options in free form.
//PAR IMEM=	Input-member-name	Specifies the name of the input source module on the library. If omitted, the source is assumed to be a non-partitioned sequential data file (such as a spooled input file).

Control Language Statement	Parameter	Description
//PAR OMEM1=	Output-member-name	Specifies the name of the relocatable object module on the library. If omitted and OBJECT=YES or COND, the assembly is aborted.
//PAR OMEM2=	Punch-indicator	1-10 alphabetic characters to indicate that PUNCH output is expected. If omitted, no PUNCH output is produced.
//PAR MAXSIZ=	1-5 decimal digits	Specifies the approximate number of source lines generated in the program. If omitted, the default value is the SYSGEN parameter, usually 1000.
//PAR LIST=	{YES / NO}	Specifies whether the source program is to be listed: YES List source program NO Omit listing If omitted, the default parameter is YES.
//PAR XREF=	{YES / NO}	Specifies whether a cross-reference list is to be generated: YES Generate cross-reference list NO Omit cross reference list If omitted, the default parameter is YES.
//PAR ERROR=	{YES / NO}	Specifies whether warning errors are to be listed: YES List warning errors NO Omit listing If omitted, the default parameter is YES. Fatal errors are always listed.
//PAR OBJECT=	{YES / NO / COND}	Specifies under what conditions a relocatable object module is to be generated: YES Module is generated unconditionally NO Module is not generated COND Module is generated if no fatal error occurred If omitted, the default parameter is YES. If the option, YES or COND, is selected, the OMEM1 option must be specified.

Control Language Statement	Parameter	Description
//DEF ID=	File identifier	Defines the source input, object output, source output, list output, and macro library input files.
//DEF ID=	INPUT	Source input file
//DEF ID=	OUTPUT1	Relocatable output file: must be 252 bytes, blocked 1. The file must be a partitioned data file. The device must be disc. CSD=YES
//DEF ID=	OUTPUT2	Source punch file identifier: must be 80 bytes, blocked 1. The file must be a non-partitioned sequential data set. CSD=YES
//DEF ID=	LIST	List output file: must be 132 bytes, blocked 1. The file must be a sequential data set. CSD=NO. The file is written with the first character being a "native" mode control character for a printer.
//DEF ID=	MACLIB MACLIB1 MACLIB2 . . . MACLIB9	Macro library input: the files must be partitioned data sets, CSD format, 80 byte records, blocked 1.

The INPUT and LIST files must always be defined; the OUTPUT1 and OUTPUT2 files are optional. //DEFINE statements may also include keyword parameters to identify the file name or the device with which the file is associated.

System macros are cataloged on the $SYSMACLIB file. Additional macro libraries may be specified by the user in a hierarchical search. The search is controlled by //DEFINE cards; the hierarchy is as follows:

```
ID=MACLIB    (Highest priority)
ID=MACLIB1
ID=MACLIB2
ID=MACLIB3
ID=MACLIB4
ID=MACLIB5
ID=MACLIB6
ID=MACLIB7
ID=MACLIB8
ID=MACLIB9    (Lowest priority)
```

MACLIB is optional; if it is not supplied, the default is $SYSMACLIB, and this file will be the highest priority in the search. $SYSMACLIB can be used at a lower priority by specifying it as the filename on the //DEFINE card for a lower priority library, and naming some other library on the //DEFINE card for ID=MACLIB. The remaining libraries, MACLIB1 through MACLIB9, must be specified if they are to be searched, and must appear in the order shown above. Any break in the sequence of these libraries terminates the search. For example, if cards are supplied in order for MACLIB1, MACLIB2, and MACLIB4, the search will terminate with MACLIB2, since MACLIB3 is missing.

The search is a true hierarchical search; that is, if a macro is called that resides on a low priority library, the search for the macro starts at the highest priority library.

If the source program is to be read from the card reader, the source program card deck must be preceded by a //DATA FIL=SYSCRD statement and be terminated by a /* card.

The cards in Figure 11-1 illustrate the Control Language statements to specify:

- Source input from cards
- Relocatable output to a specified library member
- List output to printer
- Cross-reference output

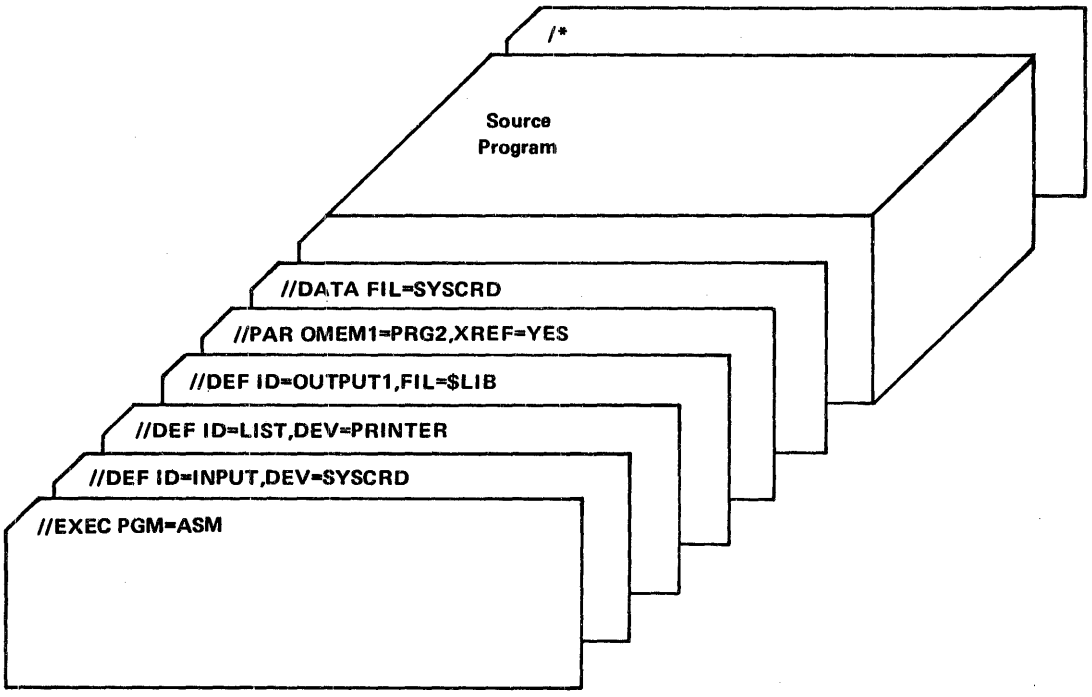


Figure 11-1. Example of Control Language Statements

The cards in Figure 11-2 illustrate the Control Language statements to specify:

- Source input from spooled input file
- Relocatable output to a specified library member
- List output to printer
- Cross-reference output

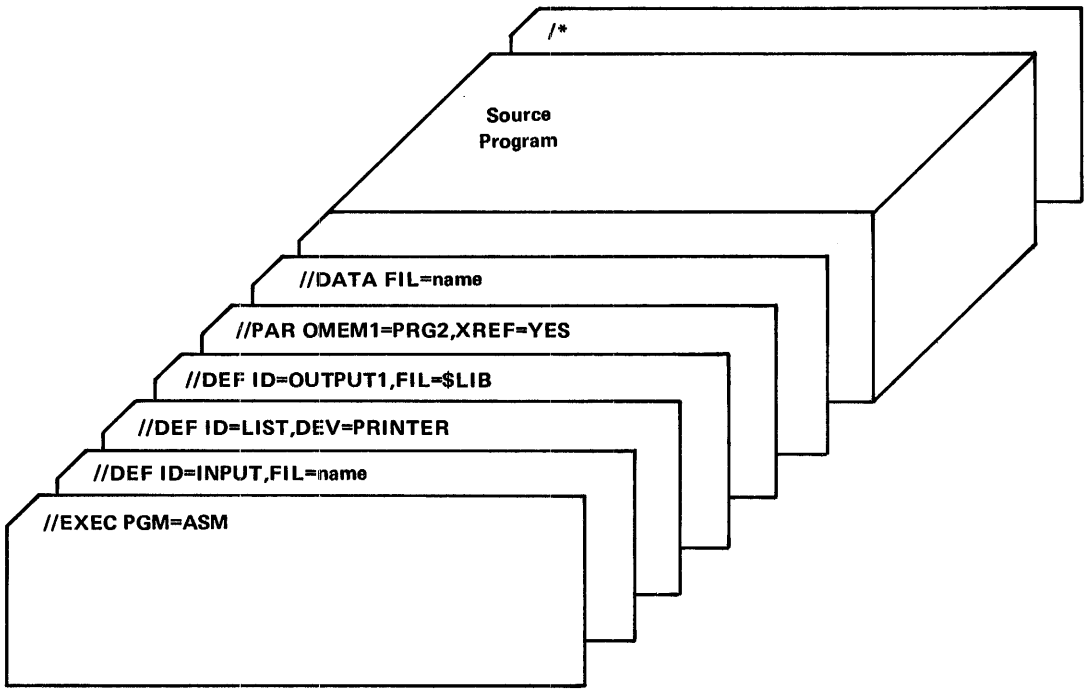


Figure 11-2. Example of Control Language Statements

The cards in Figure 11-3 illustrate the Control Language statements to specify:

- Source input from a library file
- Relocatable output to a specified library member
- List output to printer
- Cross-reference output

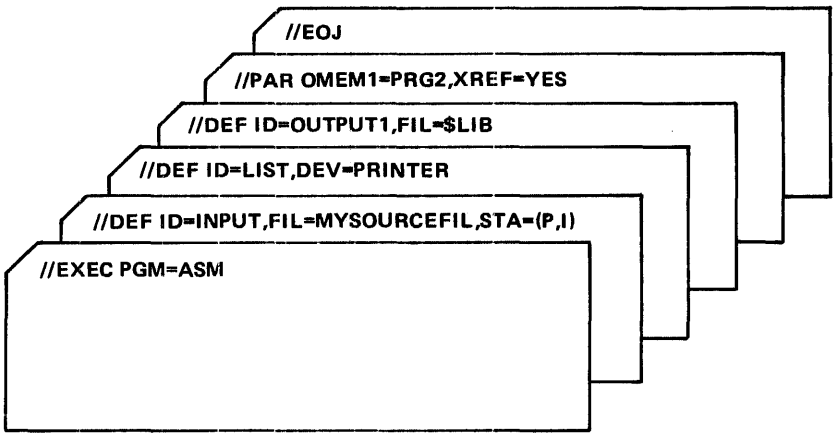


Figure 11-3. Example of Control Language Statements

To place the intermediate files on non-shared resource discs, any or all of the cards in Figure 11-4 may be included in the job control stream:

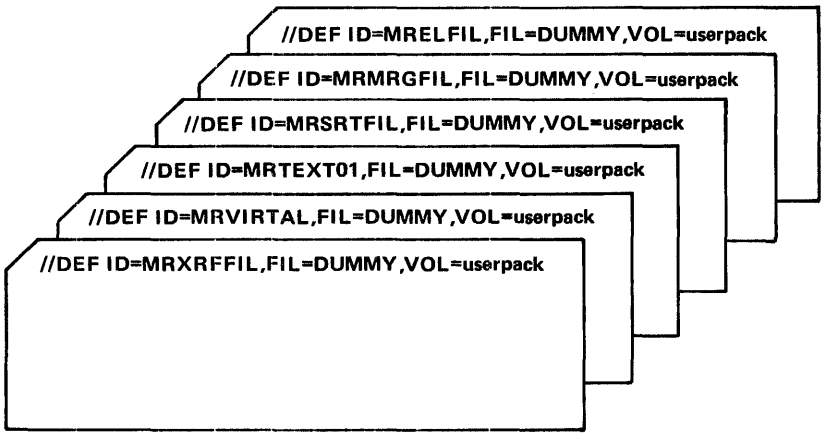


Figure 11-4. Placing Files on Disk — Example

The file MRTEXT01 has the highest traffic and would be the best candidate for performance enhancement. The file MRXRFFIL is used in macro generation and cross reference generation programs. The files MRSRTFIL and MRMRGFIL are used for cross reference. The file MRVIRTAL is used primarily for symbol table overflow. The file MRELFIL is used for error output.

A. EBCDIC REPRESENTATION

EBCDIC		
Hex Code	Graphic	Card Code
00	NUL	12-0-1-8-9
01	SOH	12-1-9
02	STX	12-2-9
03	ETX	12-3-9
04	PF	12-4-9
05	HT	12-5-9
06	LC	12-6-9
07	DEL	12-7-9
08		12-8-9
09		12-1-8-9
0A	SMM	12-2-8-9
0B	VT	12-3-8-9
0C	FF	12-4-8-9
0D	CR	12-5-8-9
0E	SO	12-6-8-9
0F	SI	12-7-8-9
10	DLE	12-11-1-8-9
11	DC1	11-1-9
12	DC2	11-2-9
13	DC3	11-3-9
14	RES	11-4-9
15	NL	11-5-9
16	BS	11-6-9
17	IL	11-7-9
18	CAN	11-8-9
19	EM	11-1-8-9
1A	CC	11-2-8-9
1B		11-3-8-9
1C	IFS	11-4-8-9
1D	IGS	11-5-8-9
1E	IRS	11-6-8-9
1F	ITB(IUS)	11-7-8-9
20	DS	11-0-1-8-9
21	SOS	0-1-9
22	FS	0-2-9
23		0-3-9
24	BYP	0-4-9
25	LF	0-5-9
26	EOB/ETB	0-6-9
27	ESC/PRE	0-7-9
28		0-8-9
29		0-1-8-9
2A	SM	0-2-8-9
2B		0-3-8-9
2C		0-4-8-9
2D	ENQ	0-5-8-9
2E	ACK	0-6-8-9

EBCDIC		
Hex Code	Graphic	Card Code
2F	BEL	0-7-8-9
30		12-11-0-1-8-9
31		1-9
32	SYN	2-9
33		3-9
34	PN	4-9
35	RS	5-9
36	UC	6-9
37	EOT	7-9
38		8-9
39		1-8-9
3A		2-8-9
3B		3-8-9
3C	DC4	4-8-9
3D	NAK	5-8-9
3E		6-8-9
3F	SUB	7-8-9
40	SP	No punches
41		12-0-1-9
42		12-0-2-9
43		12-0-3-9
44		12-0-4-9
45		12-0-5-9
46		12-0-6-9
47		12-0-7-9
48		12-0-8-9
49		12-1-8
4A	¢	12-2-8
4B	—	12-3-8
4C	<	12-4-8
4D	(	12-5-8
4E	+	12-6-8
4F	:	12-7-8
50	&	12
51		12-11-1-9
52		12-11-2-9
53		12-11-3-9
54		12-11-4-9
55		12-11-5-9
56		12-11-6-9
57		12-11-7-9
58		12-11-8-9
59		11-1-8
5A	!	11-2-8
5B	$	11-3-8
5C	*	11-4-8
5D	)	11-5-8

EBCDIC		
Hex Code	Graphic	Card Code
5E	;	11-6-8
5F		11-7-8
60	-	11
61	/	0-1
62		11-0-2-9
63		11-0-3-9
64		11-0-4-9
65		11-0-5-9
66		11-0-6-9
67		11-0-7-9
68		11-0-8-9
69		0-1-8
6A		12-11
6B	,	0-3-8
6C	%	0-4-8
6D	_	0-5-8
6E	>	0-6-8
6F	?	0-7-8
70		12-11-0
71		12-11-0-1-9
72		12-11-0-2-9
73		12-11-0-3-9
74		12-11-0-4-9
75		12-11-0-5-9
76		12-11-0-6-9
77		12-11-0-7-9
78		12-11-0-8-9
79		1-8
7A	:	2-8
7B	#	3-8
7C	@	4-8
7D	'	5-8
7E	=	6-8
7F	"	7-8
80		12-0-1-8
81	a	12-0-1
82	b	12-0-2
83	c	12-0-3
84	d	12-0-4
85	e	12-0-5
86	f	12-0-6
87	g	12-0-7
88	h	12-0-8
89	i	12-0-9
8A		12-0-2-8
8B		12-0-3-8
8C		12-0-4-8
8D		12-0-5-8
8E		12-0-6-8
8F		12-0-7-8
90		12-11-1-8
91	j	12-11-1
92	k	12-11-2
93	l	12-11-3
94	m	12-11-4

EBCDIC		
Hex Code	Graphic	Card Code
95	n	12-11-5
96	o	12-11-6
97	p	12-11-7
98	q	12-11-8
99	r	12-11-9
9A		12-11-2-8
9B		12-11-3-8
9C		12-11-4-8
9D		12-11-5-8
9E		12-11-6-8
9F		12-11-7-8
A0		11-0-1-8
A1		11-0-1
A2	s	11-0-2
A3	t	11-0-3
A4	u	11-0-4
A5	v	11-0-5
A6	w	11-0-6
A7	x	11-0-7
A8	y	11-0-8
A9	z	11-0-9
AA		11-0-2-8
AB		11-0-3-8
AC		11-0-4-8
AD		11-0-5-8
AE		11-0-6-8
AF		11-0-7-8
B0		12-11-0-1-8
B1		12-11-0-1
B2		12-11-0-2
B3		12-11-0-3
B4		12-11-0-4
B5		12-11-0-5
B6		12-11-0-6
B7		12-11-0-7
B8		12-11-0-8
B9		12-11-0-9
BA		12-11-0-2-8
BB		12-11-0-3-8
BC		12-11-0-4-8
BD		12-11-0-5-8
BE		12-11-0-6-8
BF		12-11-0-7-8
C0		12-0
C1	A	12-1
C2	B	12-2
C3	C	12-3
C4	D	12-4
C5	E	12-5
C6	F	12-6
C7	G	12-7
C8	H	12-8
C9	I	12-9
CA		12-0-2-8-9
CB		12-0-3-8-9

EBCDIC		
Hex Code	Graphic	Card Code
CC		12-0-4-8-9
CB		12-0-5-8-9
CE		12-0-6-8-9
CF		12-0-7-8-9
D0		11-0
D1	J	11-1
D2	K	11-2
D3	L	11-3
D4	M	11-4
D5	N	11-5
D6	O	11-6
D7	P	11-7
D8	Q	11-8
D9	R	11-9
DA		12-11-2-8-9
DB		12-11-3-8-9
DC		12-11-4-8-9
DD		12-11-5-8-9
DE		12-11-6-8-9
DF		12-11-7-8-9
E0		0-2-8
E1		11-0-1-9
E2	S	0-2
E3	T	0-3
E4	U	0-4
E5	V	0-5

EBCDIC		
Hex Code	Graphic	Card Code
E6	W	0-6
E7	X	0-7
E8	Y	0-8
E9	Z	0-9
EA		11-0-2-8-9
EB		11-0-3-8-9
EC		11-0-4-8-9
ED		11-0-5-8-9
EE		11-0-6-8-9
EF		11-0-7-8-9
F0	0	0
F1	1	1
F2	2	2
F3	3	3
F4	4	4
F5	5	5
F6	6	6
F7	7	7
F8	8	8
F9	9	9
FA		12-11-0-2-8-9
FB		12-11-0-3-8-9
FC		12-11-0-4-8-9
FD		12-11-0-5-8-9
FE		12-11-0-6-8-9
FF		12-11-0-7-8-9

B. OBJECT FORMATS OF MACHINE INSTRUCTIONS

The notation used to describe the source and object format in Appendixes A and B is as follows (a = absolute, r = relocatable expression).

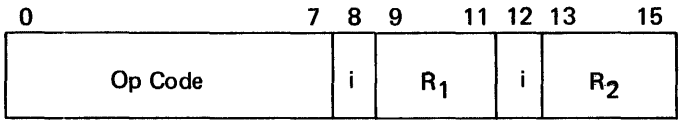
Op Code	Hexadecimal 00-FF
R	General register, 0-7. (a)
E	Extended register, 0-15. (a)
M	Memory address, 0-65,535. (a or r)
I	Immediate value-arithmetic value, shift count, skip count, or bit number. (a)
L	Field length, 0-255 (for MOVL, 0-65,535). (a)
( )	Parentheses enclose index registers and field lengths, both of which are optional.
•	A bullet following an instruction name indicates the operands are byte-addressable; other operands are word-addressable only.

Bits 8 and 12 of the object instructions are used in almost every instruction to convey information to the computer concerning that instruction. If these bits are not interpreted in any way, they are shaded; otherwise, the following symbols are used to define bits 8 and 12.

i	Indirect addressing indicator; for direct addressing i=0, for indirect addressing i=1. Indirect addressing is indicated by the programmer.
f	A sub-function indicator; indicates a function that the operation code alone cannot do. These function bits are set by the assembler.
1,0	If bit 8 or 12 must be a 1 or a 0 for a particular instruction, the bit will be shown as a 1 or 0. These bits are set by the assembler; if the wrong bit state appears in the object instruction, a no-operation occurs.

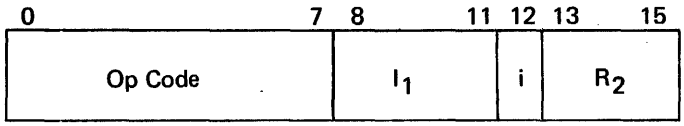
TWO BYTE INSTRUCTIONS

Op Code	Mnemonic Code	Description	Word/Operand Format
22	ADDR	Add Register-Register	
23	SUBR	Subtract Register-Register	
29	DIVR	Divide Register-Register	
28	MPYR	Multiply Register-Register	
25	ANDR	Logical Product Register-Register	
26	EORR	Exclusive OR Register-Register	
27	IORR	Inclusive OR Register-Register	
21	CMPR	Compare Register-Register	
20	MOVR	Move Register-Register	
24	INVR	Inverse Move Register-Register	
6F	ROFR	Reverse Off-Bit	
6D	RONR	Reverse On-Bit	
6E	TOFR	Test for Off-Bit	
6C	TONR	Test for On-Bit	



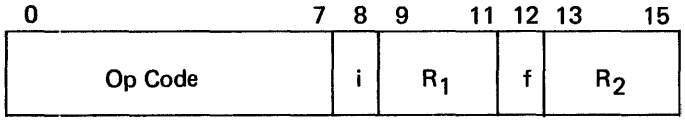
Source Operands: $@R_1,@R_2$

32	ADDI	Add Immediate	
33	SUBI	Subtract Immediate	
39	DIVI	Divide Immediate	
38	MPYI	Multiply Immediate	
35	ANDI	Logical Product Immediate	
36	EORI	Exclusive OR Immediate	
37	IORI	Inclusive OR Immediate	
31	CMPI	Compare Immediate	
30	LODI	Load Immediate	
34	INVI	Inverse Move Immediate	
F5	INP	Input from I/O Register	
F6	OUT	Output to I/O Register	



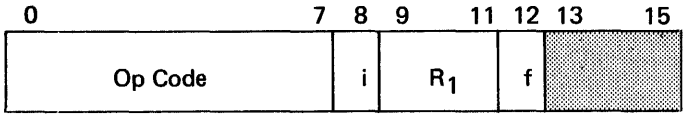
Source Operands: $I_1,@R_2$

81	INT	Convert float to fixed
81	INTT	Convert float to fixed two-word



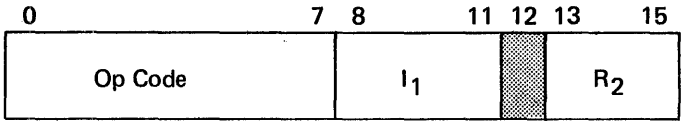
Source Operands: $@R_1,R_2$

82	FLT	Convert fixed to float
82	FLTT	Convert fixed to float two-word

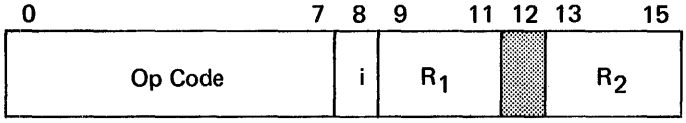


Source Operands: $@R_1$

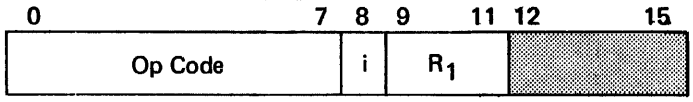
Op Code	Mnemonic Code	Description	Word/Operand Format
5F	ARDI	Arithm. Right Double Shift Immediate	
4F	ARSI	Arithm. Right Single Shift Immediate	
5C	LLDI	Logical Left Double Shift Immediate	
4C	LLSI	Logical Left Single Shift Immediate	
5D	LRDI	Logical Right Double Shift Immediate	
4D	LRSI	Logical Right Single Shift Immediate	
5E	RLDI	Rotating Left Double Shift Immediate	
4E	RLSI	Rotating Left Single Shift Immediate	
47	SRMB	Skip if Register is Minus Backward	
46	SRMF	Skip if Register is Minus Forward	
43	SRNB	Skip if Register Not Zero Backward	
42	SRNF	Skip if Register Not Zero Forward	
45	SRPB	Skip if Register is Plus Backward	
44	SRPF	Skip if Register is Plus Forward	
41	SRZB	Skip if Register is Zero Backward	
40	SRZF	Skip if Register is Zero Forward	
3F	ARDR	Arithm. Right Double Shift By Register	
2F	ARSR	Arith. Right Single Shift By Register	
3C	LLDR	Logical Left Double Shift By Register	
2C	LLSR	Logical Left Single Shift By Register	
3D	LRDR	Logical Right Double Shift By Register	
2D	LRSR	Logical Right Single Shift By Register	
3E	RLDR	Rotating Left Double Shift By Register	
2E	RLSR	Rotating Left Single Shift By Register	
F2	DIO	Disk I/O	
F1	SIO	System I/O	



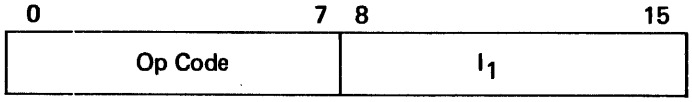
Source Operands: I_1,R_2



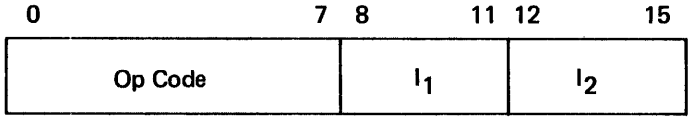
Source Operands: @R_1,R_2

Op Code	Mnemonic Code	Description	Word/Operands Format
EB	BR	Branch to Address in Register	
2B	CLDR	Condition Register Load	

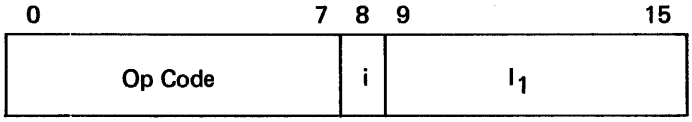
Source Operands: @R_1

BB	SB	Skip Unconditional Backward	
BA	SF	Skip Unconditional Forward	

Source Operands: I_1

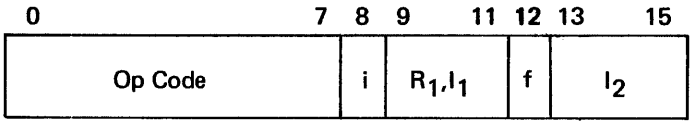
4B	SCFB	Skip on Condition Register False - Back	
49	SCFF	Skip on Condition Register False - Forward	
4A	SCTB	Skip on Condition Register True - Back	
48	SCTF	Skip on Condition Register True - Forward	

Source Operands: I_1,I_2

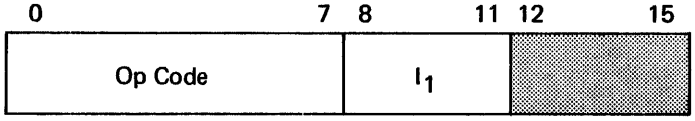
13	SR	Service Request	

Source Operand: @I_1

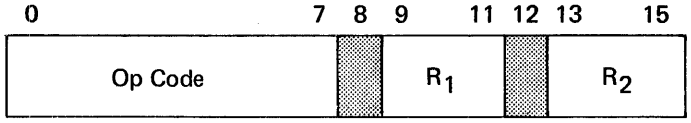
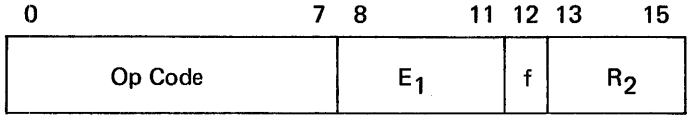
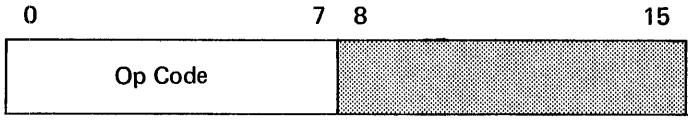
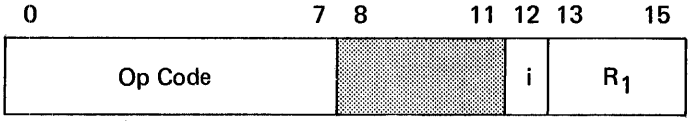
10	RBA	Reset Busy/Active Register	
10	SBA	Set Busy/Active Register	
14	RCN	Reset Control Register	
14	SCN	Set Control Register	
15	RPM	Reset Privileged Mode Register	
15	SPM	Set Privileged Mode Register	



Source Operands: @R_1,I_2 or I_1,I_2

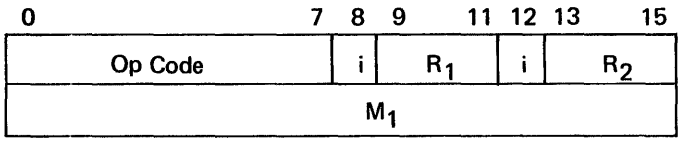
12	CTB	Clear Tie-Breaker Register	
11	TST	Test and Set Tie-Breaker Register	

Source Operand: I_1

Op Code	Mnemonic Code	Description	Word/Operand Format
EF F4	BCM WRC	Branch to Control Memory Communications Output Command	 Source Operands: R_1,R_2
F0 F0	RDX WRX	Read Extended Register Write Extended Register	 Source Operands: E_1,R_2
F3 80	RDC NEGF	Communications I/O Negate Floating Point	 Source Operands: none
2A 3A	CSTR PSTR	Condition Register Store Program Address Store	 Source Operand: @R_1

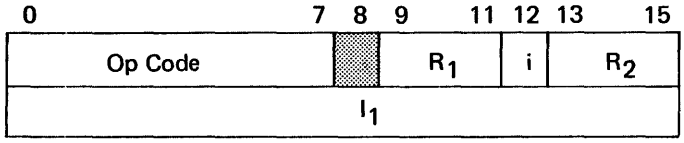
FOUR BYTE INSTRUCTIONS

Op Code	Mnemonic Code	Description	Word/Operand Format
A2	ADD	Add	
72	ADDT	Add Two-Word	
A3	SUB	Subtract	
73	SUBT	Subtract Two-Word	
A9	DIV	Divide	
A8	MPY	Multiply	
A5	AND	Logical Product	
A6	EOR	Exclusive OR	
A7	IOR	Inclusive OR	
F9	CBY	Compare Byte •	
A1	CMP	Compare	
71	CMPT	Compare Two-Word	
A0	LOD	Load, Memory-Register	
F7	LODB	Load Byte Memory-Register •	
70	LODT	Load Two-Word	
FA	STO	Store Register-Memory	
F8	STOB	Store Byte Register-Memory •	
FB	STOT	Store Two-Word	
A4	INV	Inverse Move Memory-Register	
E4	BA1	Branch Add 1	
E5	BA2	Branch Add 2	
E1	BRN	Branch if Register is Not Zero	
E0	BRZ	Branch if Register is Zero	
E6	BS1	Branch Subtract 1	
E7	BS2	Branch Subtract 2	
EA	BSR	Branch and Save Return	
EE	NOP	No Operation	



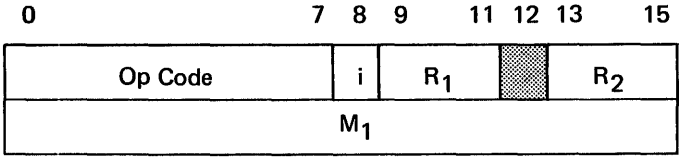
Source Operands: $@M_1(R_1),@R_2$

B2	ADDD	Add Direct	
B3	SUBD	Subtract Direct	
B9	DIVD	Divide Direct	
B8	MPYD	Multiply Direct	
B5	ANDD	Logical Product Direct	
B6	EORD	Exclusive OR Direct	
B7	IORD	Inclusive OR Direct	
B1	CMPD	Compare Direct	
B0	LODD	Load Direct	
B4	INVD	Inverse Move Direct	

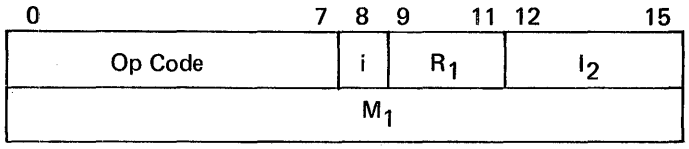
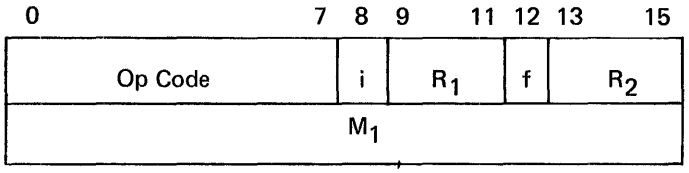
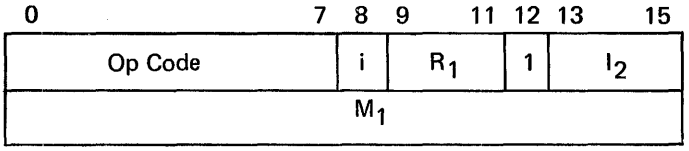
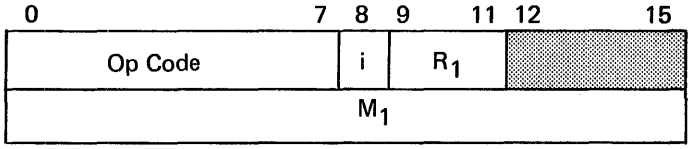
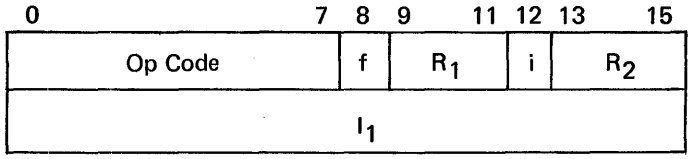


Source Operands: $I_1(R_1),@R_2$

84	LODF	Load floating point	
86	ADDF	Add floating point	
85	SUBF	Subtract floating point	
88	MPYF	Multiply floating point	
89	DIVF	Divide floating point	



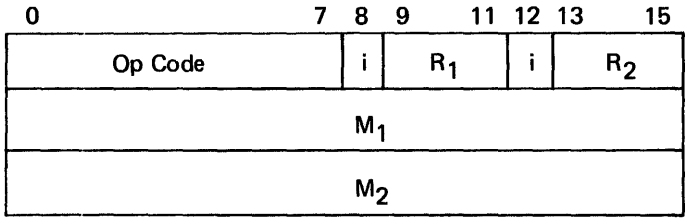
Source Operands: $@M_1(R_1),R_2$

Op Code	Mnemonic Code	Description	Word/Operand Format
E9	BCF	Branch on Condition Register - False	
E8	BCT	Branch on Condition Register - True	
E2	BOF	Branch if Bit is Off	
E3	BON	Branch if Bit is On	Source Operands: $@M_1(R_1),I_2$
AA	CVB	Convert to Binary •	
AA	CVBT	Convert to Binary Two-Word o	
AB	CVD	Convert to Decimal •	
AB	CVDT	Convert to Decimal Two-Word •	Source Operands: $@M_1(R_1),R_2$
BF	IBIT	Invert Bit •	
BD	RBIT	Reset Bit •	
BC	SBIT	Set Bit •	
BE	TBIT	Test Bit •	Source Operands: $@M_1(R_1),I_2$
ED	B	Branch-Post Indexing	
EC	BCH	Branch-Pre Indexing	
87	CMPF	Compare floating point	
8A	STOF	Store floating point	Source Operands: $@M_1(R_1)$
FE	RAR	Read Any Register	
FE	WAR	Write Any Register	
FF	RSAR	Restore All Registers	
FF	SAR	Save All Registers	
FD	RRO	Read Register Options	
FD	WRO	Write Register Options	Source Operands: $I_1(R_1),@R_2$

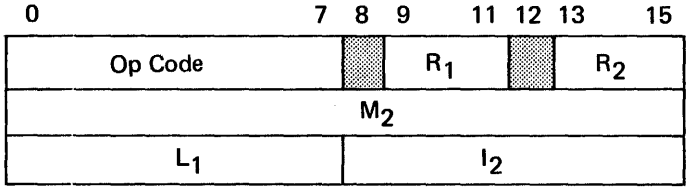
SIX BYTE INSTRUCTIONS

Op Code	Mnemonic Code	Description
62	ADDM	Add Memory-Memory
63	SUBM	Subtract Memory-Memory
69	DIVM	Divide Memory-Memory
68	MPYM	Multiply Memory-Memory
65	ANDM	Logical Product Memory-Memory
66	EORM	Exclusive OR Memory-Memory
67	IORM	Inclusive OR Memory-Memory
6B	CBYM	Compare Byte Memory-Memory •
61	CMPM	Compare Memory-Memory
6A	MOVB	Move Byte Memory-Memory •
60	MOVM	Move Word Memory-Memory
64	INVM	Inverse Move Memory-Memory

Word/Operand Format



Source Operands: $@M_1(R_1),@M_2(R_2)$

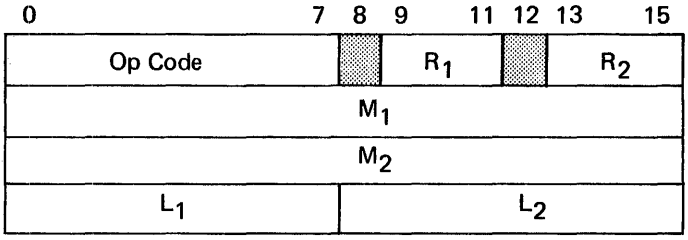


3B	SHFK	Shift Packed Decimal •

Source Operands: $M_1(L_1,R_1),I_2(R_2)$

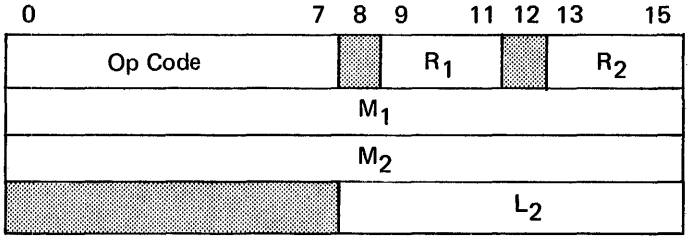
EIGHT BYTE INSTRUCTIONS

Op Code	Mnemonic Code	Description	Word/Operand Format
52	ADDK	Add Packed Decimal •	
53	SUBK	Subtract Packed Decimal •	
51	CMPK	Compare Packed Decimal •	
55	CMPX	Compare Characters •	
54	MOVX	Move Characters •	
58	PAKX	Pack •	
59	UNPX	Unpack •	
50	ZADK	Zero and Add Decimal •	
57	EDTX	Packed Decimal/Alpha Edit •	
7C	DIVK	Divide Packed Decimal •	
5B	MPYK	Multiply Packed Decimal •	



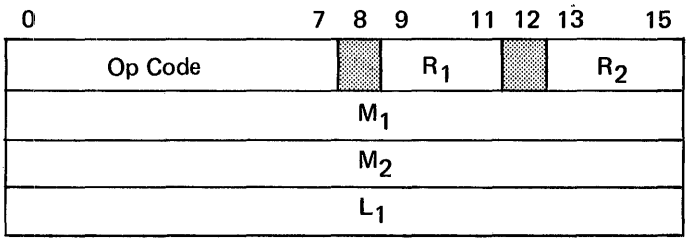
Source Operands: $M_1(L_1,R_1),M_2(L_2,R_2)$

56	TRNX	Translate •	



Source Operands: $M_1(R_1),M_2(L_2,R_2)$

5A	MOVL	Move Long •	



Source Operands: $M_1(L_1,R_1),M_2(R_2)$

C. ALPHABETICAL LIST OF MNEMONICS

This appendix lists all machine mnemonic codes and extended mnemonic codes in alphabetical order. Also included are the hexadecimal function codes, the instruction size in bytes, and the configuration of the source operand. An asterisk in the function code column indicates an extended mnemonic code. The symbols used in the operand configuration are the same as in the preceding lists.

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
ADD	A2	4	@$M_1(R_1)$,@R_2
ADDD	B2	4	$I_1(R_1)$,@R_2
ADDF	86	4	@$M_1(R_1)$,R_2
ADDI	32	2	I_1,@R_2
ADDK	52	8	$M_1(L_1,R_1)$,$M_2(L_2,R_2)$
ADDM	62	6	@$M_1(R_1)$,@$M_2(R_2)$
ADDR	22	2	@R_1,@R_2
ADDT	72	4	@$M_1(R_1)$,@R_2
AND	A5	4	@$M_1(R_1)$,@R_2
ANDD	B5	4	$I_1(R_1)$,@R_2
ANDI	35	2	I_1,@R_2
ANDM	65	6	@$M_1(R_1)$,@$M_2(R_2)$
ANDR	25	2	@R_1,@R_2
ARDI	5F	2	I_1,R_2
ARDR	3F	2	@R_1,R_2
ARSI	4F	2	I_1,R_2
ARSR	2F	2	@R_1,R_2
B	ED	4	@$M_1(R_1)$
BA1	E4	4	@$M_1(R_1)$,@R_2
BA2	E5	4	@$M_1(R_1)$,@R_2
BBR	*		@$M_1(R_1)$
BBS	*		@$M_1(R_1)$
BCF	E9	4	@$M_1(R_1)$,I_2
BCH	EC	4	@$M_1(R_1)$
BCM	EF	2	R_1,R_2
BCT	E8	4	@$M_1(R_1)$,I_2
BCY	*		@$M_1(R_1)$
BEQ	*		@$M_1(R_1)$
BGE	*		@$M_1(R_1)$
BGT	*		@$M_1(R_1)$
BID	*		@$M_1(R_1)$
BKM	*		@$M_1(R_1)$
BKP	*		@$M_1(R_1)$
BKZ	*		@$M_1(R_1)$
BLE	*		@$M_1(R_1)$
BLEQ	*		@$M_1(R_1)$
BLGE	*		@$M_1(R_1)$

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
BLGT	*		$@M_1(R_1)$
BLLE	*		$@M_1(R_1)$
BLLT	*		$@M_1(R_1)$
BLNE	*		$@M_1(R_1)$
BLT	*		$@M_1(R_1)$
BNC	*		$@M_1(R_1)$
BNE	*		$@M_1(R_1)$
BNI	*		$@M_1(R_1)$
BNV	*		$@M_1(R_1)$
BOF	E2	4	$@M_1(R_1),I_2$
BON	E3	4	$@M_1(R_1),I_2$
BOV	*		$@M_1(R_1)$
BR	EB	2	$@R_1$
BRN	E1	4	$@M_1(R_1),@R_2$
BRZ	E0	4	$@M_1(R_1),@R_2$
BS1	E6	4	$@M_1(R_1),@R_2$
BS2	E7	4	$@M_1(R_1),@R_2$
BSR	EA	4	$@M_1(R_1),@R_2$
CBY	F9	4	$@M_1(R_1),@R_2$
CBYM	6B	6	$@M_1(R_1),@M_2(R_2)$
CLDR	2B	2	$@R_1$
CMP	A1	4	$@M_1(R_1),@R_2$
CMPD	B1	4	$I_1(R_1),@R_2$
CMPF	87	4	$@M_1(R_1)$
CMPI	31	2	$I_1,@R_2$
CMPK	51	8	$M_1(L_1,R_1),M_1(L_2,R_2)$
CMPM	61	6	$@M_1(R_1),@M_2(R_2)$
CMPR	21	2	$@R_1,@R_2$
CMPT	71	4	$@M_1(R_1),@R_2$
CMPX	55	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
CSTR	2A	2	$@R_1$
CTB	12	2	I_1
CVB	AA	4	$@M_1(R_1),R_2$
CVBT	AA	4	$@M_1(R_1),R_2$
CVD	AB	4	$@M_1(R_1),R_2$
CVDT	AB	4	$@M_1(R_1),R_2$
DIO	F2	2	$@R_1,R_2$
DIV	A9	4	$@M_1(R_1),@R_2$
DIVD	B9	4	$I_1(R_1),@R_2$
DIVF	89	4	$@M_1(R_1),R_2$
DIVI	39	2	$I_1,@R_2$

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
DIVK	7C	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
DIVM	69	6	$@M_1(R_1),@M_2(R_2)$
DIVR	29	2	$@R_1,@R_2$
EDTX	57	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
EOR	A6	4	$@M_1(R_1),@R_2$
EORD	B6	4	$I_1(R_1),@R_2$
EORI	36	2	$I_1,@R_2$
EORM	66	6	$@M_1(R_1),@M_2(R_2)$
EORR	26	2	$@R_1,@R_2$
FLT/FLTT	82	2	$@R_1$
IBIT	BF	4	$@M_1(R_1),I_2$
INP	F5	2	$I_1,@R_2$
INT/INTT	81	2	$@R_1,R_2$
INV	A4	4	$@M_1(R_1),@R_2$
INVD	B4	4	$I_1(R_1),@R_2$
INVI	34	2	$I_1,@R_2$
INVM	64	6	$@M_1(R_1),@M_2(R_2)$
INVR	24	2	$@R_1,@R_2$
IOR	A7	4	$@M_1(R_1),@R_2$
IORD	B7	4	$I_1(R_1),@R_2$
IORI	37	2	$I_1,@R_2$
IORM	67	6	$@M_1(R_1),@M_2(R_2)$
IORR	27	2	$@R_1,@R_2$
LLDI	5C	2	I_1,R_2
LLDR	3C	2	$@R_1,R_2$
LLSI	4C	2	I_1,R_2
LLSR	2C	2	$@R_1,R_2$
LOD	A0	4	$@M_1(R_1),@R_2$
LODB	F7	4	$@M_1(R_1),@R_2$
LODD	B0	4	$I_1(R_1),@R_2$
LODF	84	4	$@M_1(R_1),R_2$
LODI	30	2	$I_1,@R_2$
LODT	70	4	$@M_1(R_1),@R_2$
LRDI	5D	2	I_1,R_2
LRDR	3D	2	$@R_1,R_2$
LRSI	4D	2	I_1,R_2
LRSR	2D	2	$@R_1,R_2$
MOVB	6A	6	$@M_1(R_1),@M_2(R_2)$
MOVL	5A	8	$M_1(L_1,R_1),M_2(R_2)$
MOVM	60	6	$@M_1(R_1),@M_2(R_2)$
MOVR	20	2	$@R_1,@R_2$

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
MOVX	54	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
MPY	A8	4	$@M_1(R_1),@R_2$
MPYD	B8	4	$I_1(R_1),@R_2$
MPYF	88	4	$@M_1(R_1),R_2$
MPYI	38	2	$I_1,@R_2$
MPYK	5B	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
MPYM	68	6	$@M_1(R_1),@M_2(R_2)$
MPYR	28	2	$@R_1,@R_2$
NEGF	80	2	blank
NOP	EE	4	blank or $@M_1(R_1),@R_2$
OUT	F6	2	$I_1,@R_2$
PAKX	58	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
PSTR	3A	2	$@R_1$
RAR	FE	4	$M_1(R_1),@R_2$
RBA	10	2	$@R_1,I_2$ or I_1,I_2
RBIT	BD	4	$@M_1(R_1),I_2$
RCN	14	2	$@R_1,I_2$ or I_1,I_2
RDC	F3	2	
RDX	F0	2	E_1,R_2
RLDI	5E	2	I_1,R_2
RLDR	3E	2	$@R_1,R_2$
RLSI	4E	2	I_1,R_2
RLSR	2E	2	$@R_1,R_2$
ROFR	6F	2	$@R_1,@R_2$
RONR	6D	2	$@R_1,@R_2$
RPM	15	2	$@R_1,I_2$ or I_1,I_2
RRO	FD	4	$M_1(R_1),@R_2$
RSAR	FF	4	$M_1(R_1),@R_2$
S	*		M_1 or I_1
SAR	FF	4	$M_1(R_1),@R_2$
SB	BB	2	I_1
SBA	10	2	$@R_1,I_2$ or I_1,I_2
SBIT	BC	4	$@M_1(R_1),I_2$
SBR	*		M_1 or I_1
SBS	*		M_1 or I_1
SCF	*		M_1,I_2 or I_1,I_2
SCFB	4B	2	I_1,I_2
SCFF	49	2	I_1,I_2
SCN	14	2	$@R_1,I_2$ or I_1,I_2
SCT	*		M_1,I_2 or I_1,I_2
SCTB	4A	2	I_1,I_2

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
SCTF	48	2	I_1,I_2
SCY	*		M_1 or I_1
SEQ	*		M_1 or I_1
SF	BA	2	I_1
SGE	*		M_1 or I_1
SGT	*		M_1 or I_1
SHFK	3B	6	$M_1(L_1,R_1),I_2(R_2)$
SID	*		M_1 or I_1
SIO	F1	2	$@R_1,R_2$
SKM	*		M_1 or I_1
SKP	*		M_1 or I_1
SKZ	*		M_1 or I_1
SLE	*		M_1 or I_1
SLEQ	*		M_1 or I_1
SLGE	*		M_1 or I_1
SLGT	*		M_1 or I_1
SLLE	*		M_1 or I_1
SLLT	*		M_1 or I_1
SLNE	*		M_1 or I_1
SLT	*		M_1 or I_1
SNC	*		M_1 or I_1
SNE	*		M_1 or I_1
SNI	*		M_1 or I_1
SNV	*		M_1 or I_1
SOV	*		M_1 or I_1
SPM	15	2	$@R_1,I_2$ or I_1,I_2
SR	13	2	$@I_1$
SRM	*		M_1,R_2 or I_1,R_2
SRMB	47	2	I_1,R_2
SRMF	46	2	I_1,R_2
SRN	*		M_1,R_2 or I_1,I_2
SRNB	43	2	I_1,I_2
SRNF	42	2	I_1,R_2
SRP	*		M_1,R_2 or I_1,I_2
SRPB	45	2	I_1,R_2
SRPF	44	2	I_1,R_2
SRZ	*		M_1,R_2 or I_1,I_2
SRZB	41	2	I_1,R_2
SRZF	40	2	I_1,R_2
STO	FA	4	$@M_1(R_1),@R_2$
STOB	F8	4	$@M_1(R_1),@R_2$

Mnemonic Code	Operation Code	Instruction Size (Bytes)	Operand Configuration
STOF	8A	4	$@M_1(R_1)$
STOT	FB	4	$@M_1(R_1),@R_2$
SUB	A3	4	$@M_1(R_1),@R_2$
SUBD	B3	4	$I_1(R_1),@R_2$
SUBF	85	4	$@M_1(R_1),R_2$
SUBI	33	2	$I_1,@R_2$
SUBK	53	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
SUBM	63	6	$@M_1(R_1),@M_2(R_2)$
SUBR	23	2	$@R_1,@R_2$
SUBT	73	4	$@M_1(R_1),@R_2$
TBIT	BE	4	$@M_1(R_1),I_2$
TOFR	6E	2	$@R_1,@R_2$
TONR	6C	2	$@R_1,@R_2$
TRNX	56	8	$M_1(R_1),M_2(L_2,R_2)$
TST	11	2	I_1
UNPX	59	8	$M_1(L_1,R_1),M_2(L_2,R_2)$
WAR	FE	4	$M_1(R_1),@R_2$
WRC	F4	2	R_1,R_2
WRO	FD	4	$M_1(R_1),@R_2$
WRX	F0	2	E_1,R_2
ZADK	50	8	$M_1(L_1,R_1),M_2(L_2,R_2)$

D. HEX CODE TO MNEMONIC CODE

10	SBA/RBA	42	SRNF	6A	MOVB	B7	IORD
11	TST	43	SRNB	6B	CBYM	B8	MPYD
12	CTB	44	SRPF	6C	TONR	B9	DIVD
13	SR	45	SRPB	6D	RONR	BA	SF
14	SCN/RCN	46	SRMF	6E	TOFR	BB	SB
15	SPM/RPM	47	SRMB	6F	ROFR	BC	SBIT
20	MOVR	48	SCTF	70	LODT	BD	RBIT
21	CMPR	49	SCFF	71	CMPT	BE	TBIT
22	ADDR	4A	SCTB	72	ADDT	BF	IBIT
23	SUBR	4B	SCFB	73	SUBT	E0	BRZ
24	INVR	4C	LLSI	7C	DIVK	E1	BRN
25	ANDR	4D	LRSI	80	NEGF	E2	BOF
26	EORR	4E	RLSI	81	INT/INTT	E3	BON
27	IORR	4F	ARSI	82	FLT/FLTT	E4	BA1
28	MPYR	50	ZADK	84	LODF	E5	BA2
29	DIVR	51	CMPK	85	SUBF	E6	BS1
2A	CSTR	52	ADDK	86	ADDF	E7	BS2
2B	CLDR	53	SUBK	87	CMPF	E8	BCT
2C	LLSR	54	MOVX	88	MPYF	E9	BCF
2D	LRSR	55	CMPX	89	DIVF	EA	BSR
2E	RLSR	56	TRNX	8A	STOF	EB	BR
2F	ARSR	57	EDTX	A0	LOD	EC	BCH
30	LODI	58	PAKX	A1	CMP	ED	B
31	CMPI	59	UNPX	A2	ADD	EE	NOP
32	ADDI	5A	MOVL	A3	SUB	EF	BCM
33	SUBI	5B	MPYK	A4	INV	F0	RDX/WRX
34	INVI	5C	LLDI	A5	AND	F1	SIO
35	ANDI	5D	LRDI	A6	EOR	F2	DIO
36	EORI	5E	RLDI	A7	IOR	F3	RDC
37	IORI	5F	ARDI	A8	MPY	F4	WRC
38	MPYI	60	MOVM	A9	DIV	F5	INP
39	DIVI	61	CMPM	AA	CVB/CVBT	F6	OUT
3A	PSTR	62	ADDM	AB	CVD/CVDT	F7	LODB
3B	SHFK	63	SUBM	B0	LODD	F8	STOB
3C	LLDR	64	INVM	B1	CMPD	F9	CBY
3D	LRDR	65	ANDM	B2	ADDD	FA	STO
3E	RLDR	66	EORM	B3	SUBD	FB	STOT
3F	ARDR	67	IORM	B4	INVD	FD	RRO/WRO
40	SRZF	68	MPYM	B5	ANDD	FE	RAR/WAR
41	SRZB	69	DIVM	B6	EORD	FF	SAR/RSAR

E. SUMMARY OF ASSEMBLER STATEMENTS

Name	Operation	Operand
Sequence symbol, set symbol, or blank	ADO	Arithmetic set expression, sequence symbol
Sequence symbol or blank	AGO	Sequence symbol
Symbol or blank	ALIGN	Absolute, resolved arithmetic expression
Sequence symbol	ANOP	Not used — ignored by the assembler
Symbol or blank	BDD	One or more operands separated by commas
Symbol or blank	BRS	Absolute, arithmetic expression
Symbol or blank	COM	Not used — ignored by the assembler
Symbol or blank	CSECT	Not used — ignored by the assembler
Sequence symbol or blank	EJECT	Not used — ignored by the assembler
Blank	END	Ordinary symbol or blank
Sequence symbol or blank	ENTRY	One or more relocatable symbols separated by a comma
Ordinary or variable symbol	EQU	Expression
Sequence symbol or blank	EXTRN	One or more relocatable symbols separated by a comma
Ordinary symbol	FORM	One or more positive arithmetic expressions separated by commas
Symbol or blank	Form name	Exp,exp,. . .,exp
Blank	GBLA	1-35 set symbols separated by commas
Blank	GBLC	1-35 set symbols separated by commas
Blank	ICTL	Two decimal arithmetic constants separated by a comma

Name	Operation	Operand
Blank	ISEQ	Blank, or two decimal arithmetic constants separated by a comma
Symbol or blank	LTORG	Not used
Blank	MACRO	Not used — ignored by the assembler
Sequence symbol or blank	MEND	Not used — ignored by the assembler
Sequence symbol or blank	MEXIT	Not used — ignored by the assembler
Sequence symbol or blank	MNOTE	Severity code, message or message only
Sequence symbol or blank	ORG	Relocatable expression or blank
Sequence symbol or blank	PRINT	One-to-four operands separated by commas
Sequence symbol or blank	PUNCH	Not used
Set symbol	SETA	Arithmetic set expression
Set Symbol	SETC	Character term or arithmetic set expression
Sequence symbol or blank	SPACE	Absolute arithmetic expression
Symbol or blank	TITLE	Character string constant
Symbol or blank	WDD	One or more operands separated by commas
Symbol or blank	WRS	Absolute arithmetic expression

F. MACRO EXAMPLE

The example in Figures F-1 through F-3 demonstrates the use of the FORM instruction, but may serve as a model for many macro language and conditional features. It shows the comprehensive definition for a system macro, and two MACRO instructions for that macro. Two are included because the definition generates some code unique to the first call.

The formal parameter list of the definition identifies expected parameters from the call: IDENT, LABDEF, REWIND, USAGE, CONTROL, and LIST. Three of these (REWIND, USAGE, and CONTROL) have default values provided in the definition. The other three remain null in value, if the call does not provide an explicit value for them.

By using relational expressions in SETA statements, the macro definition provides for the setting of many counters according to the parameter values provided. Hence, the first call in the example, whose parameter string is: IDENT=FILE1,USAGE=O, sets the counter &UO (line 00039) because USAGE=O was coded.

Because of the default value YES for REWIND, counter &RY (line 00042) is also set. The default ANS for CONTROL sets counter &CCA (line 00045). Conversely, counters &LY (line 00021), &LN (line 00025), and &LB (line 00037) are zero, because the LABDEF and LIST parameters were not given values by the macro call.

Note how the definition explicitly checks for failure to provide the IDENT parameter (lines 00048-00053). If the user does not provide it in the call, its count is zero (K'&IDENT), &IDNO is set to 1, and the ADO statement (line 00050) generates the fatal MNOTE message. In the call examples, IDENT=FILE1, the count (K'&IDENT) is 5 and &IDNO is not set. By similar means, the definition checks to ensure that specifications for the other parameters are correct.

When all the incoming parameter values are verified and the appropriate conditional counters are set or reset, the macro generates the I/O packet with a FORM statement to designate the principal options.

In line 00003, the macro declares the count field (&DMOCCNT) as global, and tests it for a zero (default) value. If it is zero, line 00005 sets it to 1. In this way, the macro determines whether the call is the first call within the assembly. This allows the once-only definition of the FORM instruction: $DMFRM, consisting of eight one-bit fields (line 00007). (Also note the use of the same code to generate an external for $DMOCC (line 00006), an external subroutine.)

The FORM reference is generated through the $F1 SETC (line 00069), which creates the character string $DMFRM to be used at line 00072. In the operand fields of the FORM reference, SETA references pick up the counters established by earlier conditional instructions.

In the first call shown, USAGE=O sets &UO=1 and REWIND=YES (default) sets &RY=1. Since the operand string of the &F1 statement was &CCN,0,0,0,&VO,&VV,&RY,&LB, the $DMFRM generates 0,0,0,0,1,0,1,0 as shown in the call expansion. In the second call, USAGE=U sets &UV=1, REWIND=YES (default) sets &RY=1, and the $DMFRM generates 0,0,0,0,0,1,1,0.

```
PRINT FUNCTION:  DATE=72304   TIME=074604.
   OPENL            MAC LISTING
00001               MACRO
00002    &TAG       OPENL     &IDENT=,&LABDEF=,&REWIND=YES,&USAGE=1,&CONTROL=ANS,&LIST=
00003               GBLA      &DMOCCNT
00004               ADO       &DMOCCNT<EQ>0, .DMOPIO          SET GLOBAL FORM
00005    &DMOCCNT   SETA      1                               AND EXTERNAL
00006               EXTRN     &DMOCC                          FIRST TIME
00007    &DMFRM     FORM      1,1,1,1,1,1,1,1
00008    .DMOPIO    ANOP
00009               ADO       K'&TAG<EQ>0, .DMOP15            SET 6 BYTE
00010    &TAG1      SETC      C'DM'                             TAG FOR LABEL
00011    &TG        SETC      &SYSNDX
00012               AGO       .DMOP25                         PREFIX ON FIELDS OF
00013    .DMOP15    ANOP                                      THE LIST
00014    &TG        SETC      C' '
00015               ADO       K'&TAG<LE>6, .DMOP20
00016    &TAG1      SETC      &TAG
00017               AGO       .DMOP25
00018    .DMOP20    ANOP
00019    &TAG1      SETC      &TAG(1,6)
00020    .DMOP25    ANOP
00021    &LY        SETA      &LIST<EQ>C'YES'                 LIST=YES
00022               ADO       &LY,.DMOP30
00023               AGO       .DMOP50
00024    .DMOP30    ANOP
00025    &LN        SETA      &LIST<EQ>C'NO'                  LIST=NO OR OMITTED
00026    &TAG       LODD      OPED&SYSNDX,@7                  SET RETURN INTO SAVE AREA
00027               ADO       &LN, .DMOP40
00028               BCH       @$DMOCC                         REG 6 ALREADY POINTS TO PACKET
00029               AGO       .MEXIT
00030    .DMOP40    ANOP
00031               BSR       @$DMOCC,6                       SET REG 6 AT PACKET
00032               AGO       .DMOP55
00033    .DMOP50    ANOP
00034    &TAG       ALIGN     2
00035    .DMOP55    ANOP
```

Figure F-1. Macro Definition

```
00036    * CHANGE MADE PER PTR 333 06/16/72  DJS
00037    &LB        SETA     K'&LABDEF<GE>1                    SET COUNTERS
00038    &UI        SETA     &USAGE<EQ>C'1'                    DEPENDING
00039    &UO        SETA     &USAGE<EQ>C'0'                    ON
00040    &UU        SETA     &USAGE<EQ>C'U'                    KEYWORD
00041    &USG       SETA     &UI+&UO+&UU  NE  1                SPECIFICATIONS
00042    &RY        SETA     &REWIND<EQ>C'YES'
00043    &RN        SETA     &REWIND<EQ>C'NO'
00044    &REW       SETA     &RY+&RN<NE>1
00045    &CCA       SETA     &CONTROL<EQ>C'ANS'
00046    &CCN       SETA     &CONTROL<EQ>C'NATIVE'
00047    &CNTRL     SETA     &CCA+&CCN<NE>1
00048    &IDNO      SETA     K'&IDENT<LT>1
00049    &ERR       SETA     0
00050               ADO      &IDNO, .DMOP60                    FATAL IF IDENT MISSING
00051               MNOTE    F,***IDENT KEYWORD MISSING***
00052    &ERR       SETA     1
00053    .DMOP60    ANOP
00054               ADO      &USG, .DMOP65
00055               MNOTE    F,***USAGE=&USAGE INCORRECT SPECIFICATION***
00056    &ERR       SETA     1
00057    .DMOP65    ANOP
00058               ADO      &REW, .DMOP70
00059               MNOTE    F,***REWIND=&REWIND INCORRECT SPECIFICATION***
00060    &ERR       SETA     1
00061    .DMOP70    ANOP
00062               ADO      &CNTRL, .DMOP73
00063               MNOTE    F,***CONTROL=&CONTROL INCORRECT SPECIFICATION***
00064    &ERR       SETA     1
00065    .DMOP73    ANOP
00066               ADO      &ERR, .DMOP75
00067               AGO      .MEXIT
00068    .DMOP75    ANOP
00069    &F1        SFTC     C'&DMERM'
00070               WDD      "7                                LENGTH OF PACKET (WORDS)
00071               BDD      S'04'                             OPEN
00072    &TAG1.BT&TG &F1   &CCN,0,0,0,&UO,&UU,&RY,&LB OPTION BITS
00073    &TAG1.ER&TG  WDD  "0                                   ERROR RETURNED HERE
00074               BDD      X'00'                             N/A
00075               BDD      &SYSEG                            SEGMENT TAG
00076    &TAG1.BD&TG  WDD  &IDENT                               PIR TO BDT
00077               BDD      X'00'                             N/A
00078               BDD      &SYSEG                            SEGMENT TAG
00079               ADO      &LB, .DMOP80                      LABEL PARAMETER
00080    **** CHANGE 06/21/72 PIR 4351  DJS
00081    &TAG1.LB&TG  WDD  &LABDEF
00082               AGO      .MEXIT
00083    .DMOP80    ANOP
00084    **** CHANGE 06/21/72 PTR 4351  DJS
00085    &TAG1.LB&TG  WDD  "0
00086    .MEXIT     ANOP                                       CODED
00087    OPED&SYSNDX  EQU *                                    RETURN HERE
00088               MEND
LBIN0010 LIBRARY FUNCTION COMPLETE
```

Figure F-1. Macro Definition (Continued)

```
                    0224                OPENL    IDENT=FILE1 USAGE=0
                    0227A               EXTRN    &DMOCC                  FIRST TIME
                    0228A   $DMFRM      FORM     1,1 1 1 1 1 1 1
052E    B00F0544    0239A               LODD     OPED0002 @7             SET RETURN INTO SAVE AREA
0532    EA860000    0242A               BSR      @$DMOCC 6               SET REG 6 AT PACKET
                    0245A   * CHANGE MADE PER PTR 333 06/16/72  DJS
0536    0007        0270A               WDD      "7                      LENGTH OF PACKET (WORDS)
0538    04          0271A               BDD      X'04'                   OPEN
0539    0A          0272A   DMBT0002    &DMFRM   0,0,0,0,1,0,1,0         OPTION BITS
053A    0000        0273A   DMER0002    WDD      "0                      ERROR RETURNED HERE
053C    00          0274A               BDD      X'00'                   N/A
053D    00          0275A               BDD      $SYSEG                  SEGMENT TAG
053E    0082        0276A   DMBD0002    WDD      FILE1                   PTR to BDT
0540    00          0277A               BDD      X'00'                   N/A
0541    00          0278A               BDD      $SYSEG                  SEGMENT TAG
                    0281A   **** CHANGE 06/21/72 PTR  351  DJS
0542    0000        0282A   DMLB0002    WDD      "0
        0544        0284A   OPED0002    EQU      *                       RETURN HERE
```

Figure F-2. Macro Instruction and Expansion (Call 1)

```
                    0408                OPENL    IDENT=FILE1,USAGE=U
05A4    B00F05BA    0420A               LODD     OPED0005,*7             SET RETURN INTO SAVE AREA
05A8    EA860000    0423A               BSR      @$DMOCC,6               SET REG 6 AT PACKET
                    0426A   * CHANGE MADE PER PTR 333 06/16/72  DJS
05AC    0007        0451A               WDD      "7                      LENGTH OF PACKET (WORDS)
05AE    04          0452A               BDD      X'04'                   OPEN
05AF    06          0453A   DMBT0005    &DMFRM   0,0,0,0,0,1,1,0         OPTION BITS
05B0    0000        0454A   DMER0005    WDD      "0                      ERROR RETURNED HERE
05B2    00          0455A               BDD      X'00'                   N/A
05B3    00          0456A               BDD      $SYSEG                  SEGMENT TAG
05B4    0082        0457A   DMBD0005    WDD      FILE1                   PTR TO BDT
05B6    00          0458A               BDD      X'00'                   N/A
05B7    00          0459A               BDD      $SYSEG                  SEGMENT TAG
                    0462A   **** CHANGE 06/21/72 PTR  351  DJS
C5B8    0000        0463A   DMLB0005    WDD      "0
        05BA        0465A   OPED0005    EQU      *                       RETURN HERE
```

Figure F-3. Macro Instruction and Expansion (Call 2)

G. ASSEMBLER ERROR MESSAGES

The Assembler issues two types of errors. They are source error diagnostic messages and source error abort messages. Also listed in this section are the system messages that cause the Assembler to abort.

ASSEMBLER SOURCE ERROR DIAGNOSTIC MESSAGES

The assembler source errors are printed at the end of the listing.

The messages have the following format:

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	t	aappnnn	text * insert * text

where:

nnnn	is a 4-digit decimal number that refers to the line in the source listing where the error occurred.
t	is either W designating the error as warning or F designating the error as fatal.
aappnnn	is a 7-character error code where aa is always AS specifying the Assembler as the source of the error, pp is a 2-digit decimal number indicating the pass of the Assembler during which the error occurred, and nnn is a 3-digit decimal number specifying the error within the pass.
text * insert * text	is the text of the message. If the text contains an insert, an asterisk precedes and follows the insert. An insert contains the erroneous character and all characters back to the beginning of the invalid term. For example, if the listing contains an invalid variable symbol, such as &8PAM, the insert will contain the characters &8.

The error messages and their explanations follow.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02001	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM. This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or number of operands.
nnnn	F	AS02002	UNMATCHED RIGHT PARENTHESIS.
nnnn	F	AS02003	UNMATCHED LEFT PARENTHESIS.
nnnn	F	AS02004	EXPRESSION OR SUBLIST CONTAINS AN INVALID COMMA.
nnnn	F	AS02005	INVALID SYNTAX *. . .insert. . .* An invalid element or combination of elements appear.
nnnn	F	AS02006	INVALID USE OF INDIRECTION. *. . .Insert. . .* The use of the indirect operator, @, is invalid.
nnnn	F	AS02007	INVALID EXPRESSION *. . .insert. . .* The syntax does not follow the rules for coding expressions.
nnnn	F	AS02008	INVALID USE OF LITERALS *. . .insert. . .* The use of the literal operator, =, is invalid in this statement.
nnnn	F	AS02009	INVALID USE OF PARENTHESIS *. . .insert. . .*
nnnn	F	AS02010	CHARACTER STRING INVALID WITH + - * OR / OPERATOR *. . .insert. . .* Arithmetic operations are invalid with strings.
nnnn	F	AS02011	INVALID SUBLIST *. . .insert. . .* The use of the sublist is invalid in this statement.
nnnn	F	AS02012	RELOCATABLE TERM USED IN MULTIPLICATION, DIVISION OR LOGICAL OPERATION *. . .insert. . .* The location counter, *, may not enter into the above mentioned operations.
nnnn	F	AS02013	MAY NOT FOLLOW A LOGICAL, RELATIONAL OR ARITHMETIC OPERATOR *. . .insert. . .* The unary operator NOT may not follow the above mentioned operators.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
NOTE: AS02014 and AS02015 are reserved for future use.			
nnnn	F	AS02016	SYMBOL TOO LONG *. . .insert. . .* The name entry or a symbolic operand may not exceed 8 characters.
nnnn	F	AS02017	INVALID HEXADECIMAL CONSTANT *. . .insert. . .* A hexadecimal constant may only contain digits 0-9 and characters A-F.
nnnn	F	AS02018	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or the number of operands.
nnnn	F	AS02019	INVALID OPERATOR *. . .insert. . .* The operator or symbol is not in the language.
nnnn	F	AS02020	INVAL ID CHARACTER *. . .insert. . .* The character is not in the language or is contextually incorrect.
nnnn	F	AS02021	INVALID CONTINUATION *. . .insert. . .* The usage of the semicolon is contextually invalid.
nnnn	F	AS02022	INVALID SYMBOL *. . .insert. . .* The length attribute operand may only be symbolic.
nnnn	F	AS02023	INVALID STRING *. . .insert. . .* An invalid string structure appeared.
NOTE: AS02024 is reserved for future use.			
nnnn	F	AS02025	INVALID STRING *. . .insert. . .* Same as AS02023 above.
nnnn	W	AS02026	OPERAND TRUNCATED – TOO LONG *. . .insert. . .* The number of digits exceeds the maximum allowed. For a decimal integer, 5 is the maximum. For an integer string, 10 is the maximum. For a hexadecimal constant, 4 is the maximum.
nnnn	W	AS02027	VALUE TRUNCATED – EXCEEDS PERMISSIBLE MAGNITUDE *. . .insert. . .* A decimal integer may not exceed 65,535. An integer string may not exceed 268,435,455.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02028	OPERAND ENTRY MISSING. The operation code in this statement requires an operand field entry.
nnnn	W	AS02029	CONTINUATION LINE IS BLANK.
nnnn	F	AS02030	TOO MANY CONTINUATION STATEMENTS. In a normal statement, only one continued statement is allowed.
nnnn	F	AS02031	VARIABLE SYMBOL MUST START WITH & FOLLOWED BY A-Z OR $.
nnnn	F	AS02032	LEFT PARENTHESIS MAY NOT BE IMMEDIATELY PRECEDED BY A VARIABLE SYMBOL.
nnnn	F	AS02033	VARIABLE SYMBOL MUST START WITH & FOLLOWED BY A-Z OR $.
nnnn	F	AS02034	VARIABLE SYMBOL TOO LONG. The variable symbol is greater than 7 characters excluding the ampersand sign.
nnnn	W	AS02035	.* COMMENT VALID ONLY WITH A MACRO.
nnnn	F	AS02036	CHARACTER THAT FOLLOWS . OR & MUST BE A-Z OR $. Period or & is followed by a numeric or an illegal character in the name field.
nnnn	W	AS02037	NAME ENTRY TOO LONG. In an ordinary symbol, only 8 characters are allowed and in a sequence or variable symbol only 7 characters are allowed.
nnnn	W	AS02038	INVALID CHARACTER IN NAME ENTRY. Only characters allowed are A-Z, 0-9 and $. For a sequence or variable symbol, the first character must be . or & respectively.
nnnn	F	AS02039	OPERATION ENTRY MISSING. An operation entry is required in every statement.
nnnn	F	AS02040	NAME ENTRY CANNOT BE CONTINUED.
nnnn	F	AS02041	NAME ENTRY MUST BE FOLLOWED BY A SPACE. The space is the delimiter of each field in the MRX Assembler.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02042	OPERATION ENTRY MUST BE FOLLOWED BY A SPACE. The space is the delimiter of each field in the MRX Assembler.
nnnn	F	AS02043	OPERATION CODE CANNOT BE CONTINUED.
nnnn	F	AS02044	INVALID CHARACTER IN OPERATION ENTRY. Only valid characters are A-Z, 0-9, & and $.
nnnn	F	AS02045	OPERATION ENTRY TOO LONG. Only 8 characters are allowed if there is no substitution in the operation entry.
nnnn	W	AS02046	INVALID ISEQ PARAMETERS – COMMAND IGNORED. 1. The parameter value is within the begin and end limits in the statement. 2. The length of the sequence field is zero. 3. The length of the sequence field is greater than 8 characters.
nnnn	W	AS02047	MISPLACED ICTL STATEMENT. An ICTL statement must be the first statement of an assembly.
nnnn	W	AS02048	STATEMENT VAL ID ONLY WITHIN A MACRO. The following operation codes are allowed within a macro definition: GBLA GBLC MACRO MEND MNOTE MEXIT
nnnn	W	AS02049	INVALID OR MISPLACED INSTRUCTION IN MACRO DEFINITION. The following operation codes are not allowed within a macro definition: PRINT ISEQ MACRO Also GBLA, GBLC must immediately follow the macro definition prototype.
nnnn	F	AS02051	INVALID CONTINUATION. The continued statement has a continuation character as the first nonblank character.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	W	AS02052	END STATEMENT SUPPLIED BY ASSEMBLER. End of input detected but no end card received.
nnnn	W	AS02053	NAME FIELD OF MEND STATEMENT MAY ONLY CONTAIN A SEQUENCE SYMBOL.
nnnn	W	AS02054	NAME FIELD OF MACRO DEFINITION HEADER MUST BE BLANK. The name of the operation code MACRO must be blank.
nnnn	F	AS02055	REQUIRED OPERAND ENTRY MISSING. The operation code in the statement requires an operand field entry.
nnnn	F	AS02056	SYMBOLIC PARAMETER CANNOT BE USED AS A GBLA OR GBLC OPERAND.
nnnn	F	AS02057	SET SYMBOL MAY NOT BE DEFINED AS BOTH SETA AND SETC.
nnnn	F	AS02058	INVALID SYNTAX MATCHING QUOTE MISSING.
nnnn	F	AS02059	MACHINE AND ASSEMBLER OPERATION CODES MAY NOT BE USED AS MACRO INSTRUCTION.
nnnn	W	AS02060	MULTIPLE DEFINITION OF MACRO INSTRUCTION. Macro instruction has been previously defined.
nnnn	W	AS02061	NAME ENTRY OF MACRO PROTOTYPE STATEMENT MUST BE BLANK OR A VARIABLE SYMBOL.
nnnn	F	AS02062	MORE THAN 35 SYMBOLIC PARAMETERS. Only 35 symbolic parameters are allowed.
nnnn	W	AS02063	NAME ENTRY OF MACRO INSTRUCTION MAY NOT BE A VARIABLE SYMBOL.
nnnn	W	AS02064	STATEMENT OUT OF SEQUENCE.
nnnn	W	AS02065	INVALID ISEQ SYNTAX. In an ISEQ statement the name entry must be blank and the parameters must be separated by a comma and must be terminated by a space.
nnnn	W	AS02066	INVALID ICTL SYNTAX. In an ICTL statement the name entry must be blank.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	W	AS02067	NAME FIELD OF GBLA AND GBLC STATEMENTS MUST BE BLANK.
nnnn	F	AS02068	OPERAND SYNTAX ERROR. An invalid character is in the operand field or invalid termination of a sublist, etc., appears in the operand field.
nnnn	F	AS02069	OPERAND LENGTH ERROR. The maximum number of characters allowed per operand is 127.
nnnn	F	AS02070	MORE THAN 35 OPERANDS. Only 35 operands are allowed in a macro prototype or a GBLA or GBLC statement or a macro definition instruction statement.
nnnn	W	AS02071	NAME ENTRY MAY NOT BE A SEQUENCE SYMBOL.
nnnn	W	AS02072	NAME FIEL D MUST CONTAIN A SET SYMBOL.
nnnn	W	AS02073	SET SYMBOL MAY NOT BE DEFINED AS BOTH SETA AND SETC.
nnnn	W or F	AS02074	MNOTE *. . .insert. . .* An MNOTE Assembler instruction has been encountered. The MNOTE message is output to the error file and processing continues.
nnnn	W	AS02075	VALUE OF OPERAND EXCEEDS 65535. A number being converted from EBCDIC to binary has a value greater than +65535. The 5 least-significant digits are converted to binary and only the least-significant 16 bits of the result are retained.
nnnn	F	AS02076	INVALID ADO EXPRESSION. The expression in the operand field of this ADO instruction did not resolve to an integer of value between 0 and 65535. The ADO statement is not processed.
nnnn	F	AS02077	IMPROPER ADO TERMINATION. 1. The sequence symbol in the name field of this ADO statement terminates the currently active ADO loop. The currently active ADO loop is unstacked and the ADO statement is not processed.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
			2. The sequence symbol in the name field of this macro instruction terminates the currently active ADO loop. The currently active ADO loop is unstacked. 3. A MEXIT or MEND instruction has been encountered, and the currently active ADO loop is at the current macro nesting level. The currently active ADO loop is unstacked.
nnnn	F	AS02078	SECOND ADO OPERAND MISSING. The sequence symbol is missing from the operand field of this ADO statement. The ADO statement is not processed.
nnnn	F	AS02079	BACKWARD BRANCH IN ADO OR AGO. The statement which contains the sequence symbol referred to in this ADO or AGO statement precedes the current statement. The ADO or AGO statement is not processed.
nnnn	F	AS02080	INVALID CONTINUATION. 1. Substitution into a source statement cannot be completed without generating more than one continuation line. Substitution is discontinued. 2. Substitution into a form reference cannot be completed without generating more than one continuation line. The statement is not processed.
nnnn	F	AS02081	DUPLICATE DEFINITION OF VARIABLE SYMBOL. 1. A variable symbol defined in a macro instruction is already in the local symbol table at the current macro nesting level. 2. &SYSECT or &SYSNDX is already in the local symbol table at the current macro nesting level. The value already in the symbol table is retained.
nnnn	F	AS02082	DUPLICATE FORM DEFINITION. More than one definition was encountered for the current form. The form definition is dropped.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02083	INVALID SEQUENCE SYMBOL. 1. Sequence symbol does not start with a period followed by a letter or $. 2. Sequence symbol is not 1-8 characters followed by a space. The instruction is replaced with ERR.
nnnn	F	AS02084	INVALID FORM SYNTAX. The form reference contains a keyword parameter or a sublist parameter. The form reference is dropped.
nnnn	F	AS02085	UNDEFINED SET SYMBOL OR SYMBOLIC PARAMETER. 1. Set symbol or symbolic parameter is not found in symbol table. Substitute null value for missing value. 2. Global set symbol is not found in symbol table. The ADO statement is not processed. 3. Global set symbol is not found in symbol table. The SETA or SETC statement is not processed.
nnnn	F	AS02086	INVALID SUBSTITUTION OF SEMICOLON. The first character of the value assigned to a variable symbol is a semicolon, and the character which would immediately precede it in the substitution record is not an escape character ( ). Substitute null for the value.
nnnn	F	AS02087	UNDEFINED MACRO. 1. The operation code in a statement created by substitution is neither an ordinary instruction nor a form instruction. 2. The operation code in this statement is neither an ordinary instruction nor a form instruction, nor can it be found in the macro library. The instruction is replaced by ERR.
nnnn	F	AS02088	INVALID NAME ENTRY. 1. Symbol in the name field of record created by substitution is either too long or contains an invalid character. 2. Name field of form definition is either an instruction mnemonic or it is not an ordinary symbol.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
			3. Statement has nonblank name field. 4. Statement has nonblank name field which is not a sequence symbol. 5. Statement has nonblank name field which is not an ordinary symbol or a sequence symbol. 6. Name field of ADO statement contains an ordinary symbol. 7. Name field of SETA statement does not contain a SETA symbol. 8. Name field of a SETC statement does not contain a SETC symbol. For conditions 1, 3, 4, 5, and 6, the name field is ignored. For condition 2, form definition is not processed. For conditions 7 and 8, SETA and SETC statements are not processed.
nnnn	F	AS02089	ADO OR AGO OPERAND MUST BE A SEQUENCE SYMBOL. The ADO or AGO statement is not processed.
nnnn	F	AS02090	&SYSNDX NOT IN SYMBOL TABLE. Assembler logic error. The SETA and SETC statement is not processed.
nnnn	F	AS02091	MORE THAN FIVE LEVELS OF MACRO NESTING. The instruction is replaced by ERR.
nnnn	F	AS02092	OPERAND LENGTH ERROR. Length of operand or suboperand exceeds 127 bytes. The operand is replaced by null.
nnnn	F	AS02093	TOO MANY OPERANDS. 1. More operands are in a macro call than there are parameters in the prototype. 2. AGO, SETA, or SETC statement contains more than one operand. 3. ADO statement contains more than two operands. The extra operands are ignored.
nnnn	F	AS02094	MACRO DEFINITION ERROR. Macro instruction is replaced by ERR.
nnnn	F	AS02095	TOO MANY LEVELS OF ADO NESTING. The ADO statement is not processed.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02096	INNER ADO LOOP MUST BE COMPLETELY CONTAINED WITHIN THE OUTER ADO LOOP. Processing of the outer ADO loop is discontinued.
nnnn	W	AS02097	INVALID PRINT OPERAND. 1. The operand field is terminated by a comma or semicolon. 2. One of the operands is not "OFF, ON, NOGEN, GEN, NODATA, DATA, NOCOND, OR COND." The remainder of the operand is ignored and the assembly process continues, using the last valid operand processed for each option.
nnnn	F	AS02098	MACRO INSTRUCTION MAY NOT HAVE BOTH POSITIONAL AND KEYWORD PARAMETERS. The macro instruction is not processed.
nnnn	F	AS02099	MACRO INSTRUCTION USED AS FORM NAME. The name field of a form definition which was created by substitution contains a mnemonic that has been identified as a macro instruction. The form definition is not processed.
nnnn	F	AS02100	INVALID SET EXPRESSION. 1. The operand field of SETA instruction contains a nonnumeric character. 2. The value of a SETA expression exceeds 65,535. 3. The operand field of a SETA or SETC instruction contains a sublist or a sequence symbol. The SETA or SETC instruction is not processed.
nnnn	F	AS02101	REFERENCE TO UNDEFINED SET SYMBOL. This is an Assembler logic error.
nnnn	F	AS02102	INVALID SUBSTITUTION. 1. The symbol whose value is to be used in substitution is not a SETA, a SETC, a symbolic parameter, or &SYSNDX. This is an Assembler logic error. The value is replaced by null. 2. The name field of a statement created by substitution contains a sequence symbol. The name field of the statement is ignored.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02103	REFERENCE TO &SYSNDX VALID ONLY WITHIN A MACRO. The SETA or SETC statement is not processed.
nnnn	F	AS02104	REFERENCE TO UNDEFINED FORM. Reference to a form instruction whose definition has not yet appeared in the generated source. The instruction is replaced by ERR.
nnnn	W	AS02105	UNDEFINED KEYWORD OPERAND. 1. The macro instruction has more operands than the macro prototype has parameters. 2. The macro instruction has at least one operand whose name does not match any of the prototype's keyword parameters. The extra operands are ignored.
nnnn	F	AS02106	REFERENCE TO UNDEFINED SEQUENCE SYMBOL. The ADO or AGO statement is not processed.
nnnn	F	AS02107	INVALID SUBSTITUTION INTO OPERATION ENTRY. The instruction in this substitution record was one which may not be created by substitution. The instruction is not processed.
nnnn	F	AS02108	MORE THAN 35 SUB-OPERANDS. This is an Assembler logic error.
nnnn	F	AS02109	OPERATION ENTRY MISSING. 1. The statement created by substitution is all blank after the name field. 2. The statement created by substitution has a continuation character immediately after the name field. The instruction is replaced by ERR.
nnnn	F	AS02110	INVALID OPERATION ENTRY. 1. The operation code in the statement created by substitution is more than eight characters long. 2. The operation field in the statement created by substitution contains an invalid character. 3. The operation code in the statement created by substitution is continued on the second line of the statement. The instruction is replaced by ERR.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02111	REFERENCE TO DOUBLY-DEFINED SEQUENCE SYMBOL. The ADO or AGO statement is not processed.
nnnn	F	AS02112	INSTRUCTION NOT VALID AFTER SEG STATEMENT. The instruction is not processed.
nnnn	W	AS02114	SPECIFIED SUBSTRING – LENGTH TOO LARGE.
NOTE:	AS02115 reserved for future use.		
nnnn	F	AS02116	INVALID USE OF SEQUENCE SYMBOL.
NOTE:	AS02117 reserved for future use.		
nnnn	F	AS02118	INVALID USE OF CHARACTER STRING.
NOTE:	AS02119 reserved for future use.		
NOTE:	AS02120 reserved for future use.		
NOTE:	AS02121 and AS02122 reserved for future use.		
nnnn	F	AS02123	EXPRESSION CONTAINS INCOMPATIBLE OPERAND TYPES.
nnnn	F	AS02124	EVALUATOR – STACK OVERFLOW.
nnnn	F	AS02125	MULTIPLICATION OR DIVISION OVERFLOW.
nnnn	F	AS02126	UNDEFINED SEQUENCE SYMBOL OR VARIABLE SYMBOL.
nnnn	F	AS02127	INVALID SUBSTRING OR SUBLIST REFERENCE.
nnnn	F	AS03001	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM. This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or number of operands.
nnnn	F	AS03002	UNMATCHED RIGHT PARENTHESIS.
nnnn	F	AS03003	UNMATCHED LEFT PARENTHESIS.
nnnn	F	AS03004	EXPRESSION OR SUBLIST CONTAINS AN INVALID COMMA.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03005	INVALID SYNTAX *...insert...* The lexical or contextual syntax is invalid.
nnnn	F	AS03006	INVALID USE OF INDIRECTION *. . .insert. . .* The use of the indirect operator, @, is invalid.
nnnn	F	AS03007	INVALID EXPRESSION *. . .insert. . .* The syntax does not follow the rules for coding expressions.
nnnn	F	AS03008	INVAL ID USE OF LITERALS *. . .insert. . .* The use of the literal operator, =, is invalid in this statement.
nnnn	F	AS03009	INVAL ID USE OF PARENTHESIS *. . .insert. . .*
nnnn	F	AS03010	CHARACTER STRING INVALID WITH + - * OR / OPERATOR *. . .insert. . .* Arithmetic operations are invalid with strings.
nnnn	F	AS03011	INVALID SUBLIST *. . .insert. . .* The use of the sublist is invalid in this statement.
nnnn	F	AS03012	RELOCATABLE TERM USED IN MULTIPLICATION, DIVISION OR LOGICAL OPERATION *. . .insert. . .* The location counter, *, may not enter into the above mentioned operations.
nnnn	F	AS03013	MAY NOT FOLLOW A LOGICAL, RELATIONAL OR ARITHMETIC OPERATOR *. . .insert. . .* The unary operator NOT may not follow the above mentioned operators.

NOTE: AS03014 and AS03015 are reserved for future use.

nnnn	F	AS03016	SYMBOL TOO LONG *. . .insert. . .* A name entry or a symbolic operand may not exceed 8 characters.
nnnn	F	AS03017	INVALID HEXADECIMAL CONSTANT *. . .insert. . .* A hexadecimal constant may only contain digits 0-9 and characters A-F.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03018	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or the number of operands.
nnnn	F	AS03019	INVALID OPERATOR *. . .insert. . .* The operator or symbol is not in the language.
nnnn	F	AS03020	INVALID CHARACTER *. . .insert. . .* The character is not in the language or is contextually incorrect.
nnnn	F	AS03021	INVALID CONTINUATION *. . .insert. . .* The usage of the semicolon is contextually invalid.
nnnn	F	AS03022	INVALID SYMBOL *. . .insert. . .* The length attribute operand may only be symbolic.
nnnn	F	AS03023	INVALID STRING *. . .insert. . .* An invalid string structure appeared.
nnnn	F	AS03024	INVALID STRING *...insert...* Same as AS03023 above.
nnnn	W	AS03025	FLOATING-POINT CONSTANT *...insert...* OVER/UNDERFLOW The explicitly coded exponent or the value exceeds permissible magnitude. An 8-byte floating point zero is generated.
nnnn	F	AS03025	INVALID STRING *. . .insert. . .* Same as AS03023 above.
nnnn	W	AS03026	OPERAND TRUNCATED – TOO LONG *. . .insert. . .* The number of digits exceeds the maximum allowed. For a decimal integer, 5 is the maximum. For an integer string, 10 is the maximum. For a hexadecimal constant, 4 is the maximum.
nnnn	W	AS03027	VALUE TRUNCATED – EXCEEDS PERMISSIBLE MAGNITUDE *. . .insert. . .* A decimal integer may not exceed 65,535. An integer string may not exceed 4,294,967,295.

NOTE: AS03028 and AS03029 are reserved for future use.

nnnn	W	AS03030	INVALID USE OF NAME ENTRY.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03031	INVALID FORM DEFINITION. A form reference cannot be used as an operand entry.
nnnn	F	AS03032	INVALID USE OF $SYSEG. $SYSEG is invalid in an evaluated expression.
nnnn	F	AS03033	RELOCATABLE TERM INVALID WITH UNARY OPERATOR. A relocatable term may not be used in conjunction with a NOT operator.
nnnn	F	AS03034	INVALID USE OF STRING CONSTANT. A string constant may not be used as a term in conjunction with an arithmetic or logical operator.
nnnn	F	AS03035	RELOCATABLE TERM INVALID WITH */ OR LOGICAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a multiplication, division, or logical operation.
nnnn	F	AS03036	INVALID SYNTAX. The operand structure does not follow the rules of the language.
nnnn	F	AS03037	MORE THAN SIX RELOCATABLE TERMS. An expression may not contain more than six unresolved relocatable terms.

NOTE: AS03038 is reserved for future use.

nnnn	F	AS03039	INVALID STRING *. . .insert. . .* Both strings in a relational or logical operation must be the same type.
nnnn	F	AS03040	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* The expression size is too large to be evaluated.
nnnn	F	AS03041	MULTIPLICATION OR DIVISION OVERFLOW *. . . insert . . .* Either division overflow has occurred or the second term of a multiplication or division operation exceeds a 16-bit value.

NOTE: AS03042 is reserved for future use.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03043	RELOCATABLE TERM INVALID WITH RELATIONAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a relational operator.
nnnn	F	AS03044	UNDEFINED SYMBOLIC OPERAND. The reference is not used as a label within this program.

NOTE: AS03045 is reserved for future use.

NOTE: AS03046 through AS03050 are reserved for future use.

nnnn	F	AS03051	DUPLICATE ENTRY OF A SYMBOLIC DEFINITION *. . .insert. . .* The name field entry definitions must be unique. All duplicates are discarded.
nnnn	F	AS03052	ENTRY POINT DEFINITION IS NOT RELOCATABLE *. . .insert. . .* The entry point definition must resolve to a relocatable term.
nnnn	F	AS03053	CSECT NAME IS ALREADY DEFINED, BUT NOT AS CSECT *. . .insert. . .* Control section names must not appear as ordinary name field entries.
nnnn	F	AS03054	COM NAME IS ALREADY DEFINED, BUT NOT COM *. . .insert. . .* COM names must not appear as ordinary name field entries.

NOTE: AS03055 is the same as AS03052.

NOTE: AS03056 through AS03058 are reserved for future use.

nnnn	F	AS03059	DUPLICATE FORM DEFINITION. *. . .insert. . .* The form definition name entry is previously defined.
nnnn	F	AS03060	INVALID SYNTAX IN STORAGE RESERVATION. The operand of a reserve storage instruction must be preresolved, absolute, positive arithmetic expression. Only one operand is allowed.
nnnn	F	AS03061	INVALID SYNTAX IN DATA DEFINITION. The syntactical structure of the data definition operand is invalid.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03062	INVALID USE OF INDIRECTION. An indirect operator is invalid in data definition operands.
nnnn	F	AS03063	INVALID USE OF LITERAL. The literal operator is invalid in size or repetition field of a data definition operand.
nnnn	F	AS03064	INVALID SIZE SPECIFICATION. The size operand of data definitions must be preresolved absolute expression.
nnnn	F	AS03065	INVALID REPETITION FACTOR. The repetition factor of a data definition operand must be a preresolved absolute expression.
nnnn	F	AS03066	VALUE OF LOCATION COUNTER EXCEEDS 65,535.
nnnn	W	AS03067	TRUNCATION OCCURRED. The implied size of the value operand is greater than the explicit size operand in a data definition.

NOTE: AS03068 through AS03069 are reserved for future use.

nnnn	F	AS03070	INVALID USE OF $SYSEG. The data following $SYSEG definition must be two bytes long, word aligned, and relocatable.
nnnn	F	AS03071	REQUIRED OPERAND ENTRY MISSING. This instruction requires an operand and none was supplied.
nnnn	F	AS03072	INVALID SUBLIST. The syntax indicates a suboperand, but the instruction does not allow suboperands.
nnnn	F	AS03073	INVALID USE OF LITERAL. The instruction does not allow a literal as an operand, but one was coded.
nnnn	F	AS03074	INVALID USE OF INDIRECTION. The instruction does not allow indirection, but indirection was coded.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03075	INVALID EXPRESSION. The expression coded does not fall within the types allowed. 1. A string type was used where only arithmetic type expressions are allowed. 2. An unresolved expression was coded on an instruction which required expressions to be predefined. 3. A relocatable expression is coded where only absolute are allowed.
nnnn	W	AS03076	NAME FIELD OF ORG STATEMENT MAY ONLY CONTAIN A SEQUENCE SYMBOL.
nnnn	W	AS03077	INVALID RELOCATION. 1. An absolute value was coded where a relocatable value was required. 2. The relocation identifier does not match the relocation identifier for the control section in effect; e.g., trying to ORG to another CSECT or COM section.
nnnn	F	AS03078	NAME ENTRY REQUIRED ON AN EQU STATEMENT.
nnnn	W	AS03079	TOO MANY OPERANDS. More than the maximum number of operands allowable for this instruction were coded. The values of the first operands were used.
nnnn	F	AS03080	ONLY SINGLE TERM RELOCATABLE EXPRESSIONS ARE VALID.
nnnn	F	AS03081	COMBINED CSECT, COM AND EXTRN COUNT EXCEEDS 252. The binary generated will probably not be useless. Reduce the number of EXTERNs and CSECTs and COMs and reassemble.
nnnn	W	AS03082	INVALID NAME ENTRY. The name entry was coded where none was allowed. The name entry has not been entered into the symbol table. Any reference to it will result in an undefined reference.
nnnn	F	AS03083	REFERENCE TO INVALID FORM DEFINITION. 1. The operation entry matches something other than a FORM definition. 2. There was no such entry. 3. No operand was coded on the reference.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03084	OPERAND SIZE EXCEEDS 255 BITS. A FORM definition was coded with an operand which did not resolve to an absolute value less than 255.
nnnn	F	AS03085	DUPLICATE ENTRY OF A SYMBOLIC DEFINITION.
nnnn	F	AS03086	NAME ENTRY REQUIRED ON A FORM DEFINITION.
nnnn	F	AS03087	CSECT OR COM NAMES MAY NOT DUPLICATE ANOTHER NAME. The name entries of CSECT and COM statements cannot duplicate each other or other symbols in the same assembly.
nnnn	W	AS03088	ONLY ONE TITLE STATEMENT IN A PROGRAM MAY HAVE A NONBLANK NAME FIEL D.
nnnn	F	AS03089	INVALID PRIME ENTRY POINT. 1. The operand of an END statement does not resolve to an even boundary. 2. The operand does not reference a relocatable value.
nnnn	F	AS04001	INVALID FORM DEFINITION. A form reference cannot be used as an operand entry.
nnnn	F	AS04002	INVALID USE OF $SYSEG. $SYSEG is invalid in an evaluated expression.
nnnn	F	AS04003	RELOCATABLE TERM INVALID WITH UNARY OPERATOR. A relocatable term may not be used in conjunction with a NOT operator.
nnnn	F	AS04004	INVALID USE OF STRING CONSTANT. A string constant may not be used as a term in conjunction with an arithmetic or logical operator.
nnnn	F	AS04005	RELOCATABLE TERM INVALID WITH */ OR LOGICAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a multiplication, division, or logical operation.
nnnn	F	AS04006	INVALID SYNTAX. The operand structure does not follow the rules of the language.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS04007	MORE THAN SIX RELOCATABLE TERMS. An expression may not contain more than six unresolved relocatable terms.
NOTE:	AS04008 is reserved for future use.		
nnnn	F	AS04009	INVALID STRING *. . .insert. . .* Both strings in a relational or logical operation must be the same type.
nnnn	F	AS04010	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* The expression size is too large to be evaluated.
nnnn	F	AS04011	MULTIPLICATION OR DIVISION OVERFLOW *. . .insert. . .* Either division overflow has occurred or the second term of a multiplication or division operation exceeds a 16-bit value.
NOTE:	AS04012 is reserved for future use.		
nnnn	F	AS04013	RELOCATABLE TERM INVALID WITH RELATIONAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a relational operator.
nnnn	F	AS04014	UNDEFINED SYMBOLIC OPERAND. The reference is not used as a label within this program.
NOTE:	AS04015 reserved for future use.		
nnnn	W	AS04016	TRUNCATION OCCURRED. The value coded exceeded the match field on a FORM definition. Normal rules of truncation are followed.
nnnn	W	AS04017	TOO MANY OPERANDS.
nnnn	W	AS04018	REQUIRED OPERAND ENTRY MISSING. Fewer operands were coded in a form reference than were coded in the FORM definition.
nnnn	F	AS04019	INVALID RELOCATION-VALUE TREATED AS ABSOLUTE. 1. The receive field was not 16 bits or was not on an even byte boundary. The number of relocation identifiers was greater than one. 2. For $SYSEG, the receive field was not 8 bits long, or was not on a byte boundary.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS04020	INVALID SYNTAX. The syntax of a FORM reference operand may have had one of the following errors: 1. Indirection coded 2. A sublist coded 3. No operand was coded.
NOTE:	AS04021 is reserved for future use.		
nnnn	W	AS04022	TITLE OPERAND MUST BE A CHARACTER STRING.
nnnn	W	AS04023	LENGTH OF TITLE OPERAND EXCEEDS 90 CHARACTERS. The operand is truncated on the right to 90 bytes and then used.
nnnn	W	AS04024	SPACE OPERAND MUST BE A RESOLVED ARITHMETIC EXPRESSION.
NOTE:	AS04025 is reserved for future use.		
nnnn	W	AS04026	FORWARD REFERENCE TO STRING EQUATE – IMPLICIT STRING – LENGTH IS 2 BYTES. A data definition operand referencing a forward string equate is implicitly resolved to two bytes. Truncation or padding may have occurred.
nnnn	W	AS04027	TRUNCATION OCCURRED. The implicit size of the value operand is greater than the explicit size operand in data definitions.
nnnn	F	AS04028	INVALID RELOCATION – VALUE TREATED AS ABSOLUTE. A relocatable value must be on word boundary and two bytes in length. The number of relocation factors in an expression must be resolved to one.
NOTE:	AS04029 is reserved for future use.		
nnnn	W	AS04030	NOP SUBSTITUTED FOR INVALID OPERATION CODE.
nnnn	F	AS04031	TOO MANY OPERANDS OR SUBOPERANDS. Refer to the MRX/OS Assembler Reference manual to determine the maximum number of operands.
nnnn	F	AS04032	INVALID INDIRECTION OR LITERAL USAGE.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	W	AS04033	LITERAL POOL SIZE EXCEEDS MEMORY LIMITS. The current literal pool is located at an address greater than 65,535. Binary generation is suppressed.
nnnn	W	AS04034	INVALID USE OF STRING CONSTANT. A string constant may have been used as an address or as a register designator.
nnnn	F	AS04035	LOSS OF SIGNIFICANCE. An immediate operand cannot contain the amount coded.
nnnn	F	AS04036	INVALID RELOCATION. More than one relocatable value remains after evaluation.
nnnn	F	AS04037	REQUIRED SUBOPERAND MISSING. An instruction which must have a certain minimum of operands or suboperands has been coded without one or more of them.
nnnn	F	AS04038	VALUE NOT WITHIN RANGE OF DESIGNATED FIELD. The resolved value exceeds the maximum value permitted.
nnnn	W	AS04039	EXPLICIT USE OF REGISTER ZERO AS AN INDEX. Register zero has no effect as an index register, but it was coded as an index.
nnnn	W	AS04040	TRUNCATION OCCURRED. A string constant used in a direct instruction exceeds 2 bytes. Normal rules of truncation are used.
nnnn	W	AS04041	WORD BOUNDING REQUIRED. The instruction functions only on even bounded addresses, but an odd address was coded.
nnnn	W	AS04042	RELOCATABLE TERM USED – ABSOLUTE VAL UE REQUIRED.
nnnn	W	AS04043	EDIT LENGTH L2 MUST BE GREATER THAN L1 UNLESS L2 IS ZERO.
nnnn	W	AS04044	INVAL ID USE OF $SYSEG. $SYSEG is not allowed as an operand or sub-operand of any machine instruction.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS04045	INVALID ENTRY POINT. The entry point did not resolve to a relocatable value. Note that the line number is associated with the END statement because the fact cannot be discovered when the ENTRY instruction was encountered.
nnnn	F	AS04046	MORE THAN 175 SEG DIRECTIVES ENCOUNTERED – FIRST 175 USED.

ASSEMBLER ABORT MESSAGES

The MRX/OS Assembler abort messages are printed as the final line(s) of the listing. The assembly is aborted when any of these messages appear. The messages and their explanations are given below. The format of the messages is exactly as presented.

MESSAGES

ERROR IN PHASE CALL.
Assembler in error.

ERROR IN I/O HANDLING.

INPUT BLOCK SIZE EXCEEDS MEMORY AVAILABLE.
The partition size is too small to handle the larger input block size. Increase the partition size for this input file. In 8K, the maximum input block size allowed is 86 bytes.

PARTITION-SIZE LESS THAN 8K BYTES.
The Assembler requires a minimum of 8K bytes to run.

SYMBOL TABLE ERROR.
Assembler in error.

SYSTEM MACRO BLOCK SIZE TOO LARGE (GREATER THAN 86 BYTES).

END OF INPUT OCCURRED WHILE PROCESSING A MACRO – MEND STATEMENT MISSING.
Either an end of input or an END statement was detected while a macro definition was being processed.

FATAL ERROR IN ICTL STATEMENT PARAMETER.
In an ICTL statement, the first parameter is less than 1 or greater than 40, or the second parameter is less than 40 or greater than 120.

PARAMETER DELIMITER ERROR IN COLUMN xx IN STATEMENT *. . .insert. . .*

PARAMETER CARD ERROR – JOB ABORTED.
On a //PAR statement, either a parameter or a delimiter is in error. The contents of the //PAR statement, but not the characters //PAR, are printed as an insert between the asterisks. The column number, xx, indicates the erroneous character, column 1 being the first character between the asterisks. Several parameter or delimiter errors may be listed in succession before the PARAMETER CARD ERROR message.

FATAL ERROR IN ICTL STATEMENT SYNTAX.
The ICTL statement parameters must be separated by a comma and terminated by a space.

SPECIFIED IMEM NOT FOUND IN LIBRARY.
Either the member name does not exist or the wrong library name was used.

MEND STATEMENT MISSING IN SYSTEM MACRO.
While processing a system macro definition an end of input was detected, but not a MEND statement. Check if the system macro library has been destroyed or if just one macro was incorrectly written on it.

BINARY REQUESTED – BUT OMEM1 NOT SUPPLIED.
A member name must be provided so that the binary may be entered in the object library under that name.

OPERAND MISSING IN ICTL STATEMENT.
An ICTL statement requires an operand entry.

SYSTEM MESSAGES

The following system messages cause the assembler to abort.

ERROR CODE	MESSAGE TEXT
5021	ILLEGAL BLOCK NUMBER. If the assembly process has not reached the print phase, the most probable cause is too small an allocation of MAXSIZ. Another possible cause is too small a file allocated for OMEM2. If the assembly process has reached the print stage, the most probable cause is that the binary file (OUTPUT1) has been filled.
21nn	These errors are probably caused by an error in the Control Language statements.
24nn	These errors are probably caused by an error in the Control Language statements. Error 2109 indicates too large a MAXSIZ as a probable cause. Error 240B indicates that one or more of the user-supplied files is noncontiguous. The file with multiple extents must be recreated as a contiguous file.

INDEX

Abort messages G-25
Absolute
  expressions, definition of 2-17
  terms, definition of 2-4
Addressing in machine instructions 3-2,3-3
ADO statement 10-20
AGO statement 10-23
ALIGN statement 6-6
Alignment
  of data 8-2
  of machine instructions 3-1
Alphabetical list of machine instructions C-1
ANOP statement 10-23
Arithmetic
  constants 2-7
  operators 2-16
  set expressions 10-16
Assembler instructions
  definition 1-1
  overview 4-1
  summary E-1
Assembly options 11-1

BDD statement 8-2
Begin-end columns
  alteration of 6-4
  description 2-18
BRS statement 8-5
Byte reserve storage 8-5
Byte defined data 8-2

Calling assembler 11-1
Card codes A-1
Character
  codes A-1
  set 2-1
  set expression 10-18
  string constants 2-5
Coding form
  format of 2-18
  statement continuation 2-21
COM statement 5-4
Comments 2-20
Common control sections (see COM statement)
Conditional assembly statements
  ADO 10-20
  AGO 10-23
  ANOP 10-23
  GBLA 10-20
  GBLC 10-20
  SETA 10-16
  SETC 10-18
Constants
  arithmetic 2-7
  string 2-5
Continuation of statements 2-21
Control language for assembler 11-1
Control sections
  and location counter 2-11
  assembler statements for 5-1
Count attribute of macro instruction operand 10-24
Cross reference list, suppressing 11-2
CSECT statement 5-1

Data definition statements 8-1
Diagnostic messages G-1

EBCDIC table A-1
EJECT statement 9-2
END statement 6-3
ENTRY statement 5-2
EQU statement 8-1
Error messages
  abort messages G-25
  diagnostic messages G-1
  system messages G-26

Expressions
  absolute 2-17
  definition 2-14
  evaluation of 2-15
  relocatable 2-18
Extended mnemonics 3-10
EXTRN statement 5-3

Floating point string constants 2-6a
FORM definition statement 8-7
FORM definition statement 8-7
FORM instruction statement 8-7

GBLA statement 10-20
GBLC statement 10-20
General purpose machine instructions 3-3,3-5
Global arithmetic and character set symbols 10-20
Hex codes of machine instructions D-1
Hexadecimal string constants 2-6

ICTL statement 6-4
Identification-sequence field 2-20
Index registers 3-2,3-3
Input format control 6-4
Integer string constants 2-6
ISEQ statement 6-5

Job control language (see Control language)

Linkage editor
  and control sections 5-1
  and symbol linkage 5-2
  map directive (SEG) 7-1
Linking statements 5-1,5-2
Listing control statements
  EJECT 9-2
  PRINT 9-3
  SPACE 9-2
  TITLE 9-1
Literal pools
  and LTORG statement 6-4
  description 2-13
Literals
  description 2-13
  in WDD and BDD statements 8-3

Location counter
  and ALIGN statement 6-6
  and ORG statement 6-1
  and WDD and BDD statements 8-4
  description 2-11
  reference (asterisk) 2-11
Logical operators 2-16
LTORG statement 6-4

Machine instructions
  alphabetical list C-1
  definition 1-1
  hex code to mnemonic D-1
  object formats B-1
  summary 3-1
Macro language
  concatenation of variable symbols 10-11
  count attribute 10-24
  example F-1
  file definition 11-1
  general description 10-1
  macro definition 10-1
  macro instruction 10-5
  MEXIT statement 10-13
  MNOTE statement 10-12
  nesting of macros 10-13
  number attribute 10-25
  sublists in macro instructions 10-9
  sublists in model statements 10-9
  substring notation 10-10
  system variable symbols (&SYSNDX, &SYSECT) 10-13
Messages, error G-1
Mnemonic definition (FORM) 8-7

Name field, description 2-3
Notation used to describe machine instructions 3-2
Number attribute of macro instruction operand 10-25

Object formats of machine instructions B-1
Object program
  definition 1-1
  file definition 11-1

Operand field, description 2-3,2-20
Operating system, relationship to assembler 1-2
Operation codes (hex) for machine instructions D-1
Operation field, description 2-3,2-20
Operators 2-16
Ordinary symbols 2-9
ORG statement 6-1

Packed decimal string constants 2-6
PRINT statement 9-3
Program
  control statements 6-1
  listing 1-2,9-1
  sectioning 5-1
  termination 6-3
PUNCH statement 6-3

Registers 3-2,3-3
Relational operators 2-16
Relocatable
  expressions, definition of 2-18
  symbols, identification of 5-1,5-2
  terms, definition of 2-4
Reserving storage 8-5

SEG statement 7-1
Segment names (see SYSEG)
Sequence checking statements 6-5
Sequence symbols 2-11
Set symbols 10-16
SETA statement 10-16
SETC statement 10-18
Source program
  definition 1-1
  file definition 11-1
  listing control 9-1,11-2
Source statements
  basic format 2-3
  character set 2-1
  coding form 2-18
  expressions 2-3,2-14
Source statements (continued)
  terms 2-3,2-4
SPACE statement 9-2
String constants
  character 2-5
  floating point 2-6a
  hexadecimal 2-6
  integer 2-6
  packed decimal 2-6
  zoned decimal 2-6
Symbol definition statements 8-1
Symbol length attribute 2-12
Symbolic linkage statements 5-2
Symbols
  definition 2-8
  ordinary 2-9
  rules for using 2-9
  sequence 2-11
  variable 2-10
SYSEG reserved name 5-6,8-3
System
  machine instructions 3-3,3-9
  messages G-26
  requirements for assembler 1-2

TITLE statement 9-1
Termination of assembly 6-3
Terms
  constants 2-5
  definition 2-3,2-4
  literals 2-13
  location counter reference 2-11
  symbols 2-8
  symbol length attribute 2-12

Variable symbols 2-10

WDD statement 8-2
Word defined data 8-2
Word reserve storage 8-5
Writing to disk file 6-3
WRS statement 8-5

Zoned decimal string constants 2-6

COMMENTS FORM

MRX/OS Assembler Reference Manual (2202.001-01)

Please send us your comments, to help us produce better publications. Use the space below to qualify your responses to the following questions, if you wish, or to comment on other aspects of the publication. Please use specific page and paragraph/line references where appropriate. All comments become the property of the Memorex Corporation.

	Yes	No
• Is the material:		
Easy to understand?	☐	☐
Conveniently organized?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Accurate?	☐	☐
Suitable for its intended audience?	☐	☐
Adequately indexed?	☐	☐

• For what purpose did you use this publication? (reference, general interest, etc.)

_____

• Please state your department's function: _____

_____

• Please check specific criticism(s), give page number(s), and explain below:

☐ Clarification on page(s) _____

☐ Addition on page(s) _____

☐ Deletion on page(s) _____

☐ Error on page(s) _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

MEMOREX

First Class

Permit No. 14831
Minneapolis,
Minnesota 55427

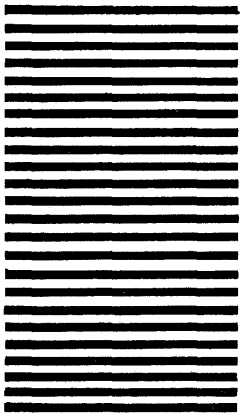
Business Reply Mail

No Postage Necessary if Mailed in the United States

Postage Will Be Paid By

Memorex Corporation

Midwest Operations – Publications
8941 Tenth Avenue North
Minneapolis, Minnesota 55427



. . . . . .

. . . . . .

Thank you for your information. . . . . .

Our goal is to provide better, more useful manuals, and your comments will help us to do so.

. . . . . .Memorex Publications

LITERALS

A literal term is used to introduce data into a program. The formats of a literal term are as follows.

=a	a = Data value to be generated (required); any legal expression except another literal term.
=a(b,c)	b = Length specification (in bytes): a positive absolute expression. If omitted, the length specification is the implied size of the expression.
=a(b)	
=a(,c)	c = Repetition factor; a positive absolute expression. If omitted, a repetition of 1 is assumed. If the size or length is specified symbolically, the symbol must have been previously defined.

Where:

Examples of literals are shown in Figure 2-9.

==C'ABD'	Invalid: literal cannot define another literal.
=C'ABD'(4,3)	Valid: same as =C'ABD ABD ABD'
=A+B/2+4	Valid: implied length is the length of symbol A; implied repetition factor is 1.
=P'-446'(6)	Valid: specified length is 6; implied repetition factor is 1.
=X'FF00'(,3)	Valid: implied length is 2; specified repetition factor is 3.

Figure 2-9. Examples of Literals

The assembler generates the literal data, stores this data in a literal pool, and places the address of the stored data in the operand field of the statement using the literal. The position of the literal pool may be controlled by the programmer with the LTORG assembler statement. If LTORG is not specified at the end of a control section, the literal pool for that segment is placed at the end of the first control section.

A literal can be defined at any point in a program by specifying the literal in the operand of the statement in which it is used. In contrast, data definition statements define and label data, and then the label is used to specify the data.

A literal may not be combined with any other term, nor may a literal be used as a receiving field of a statement that modifies storage.

Literals are relocatable, because the address of the literal, not the literal itself, is assembled into the statement using the literal.

Publications Bulletin

2202.001-0101
3/19/73

Release 2 Update Package for:

MRX/OS Assembler
Reference Manual
2202.001

This bulletin advises of changes which have occurred to the **Assembler Reference Manual** since the **December 1972** edition was issued. New and replacement pages are provided where required.

Pages	Action
v and vi	Replace
2-1 thru 2-6	Replace
2-6a	Add
2-7 and 2-8	Replace
2-13 and 2-14	Replace
8-3 and 8-4	Replace
8-9 and 8-10	Replace
10-9 and 10-10	Replace
10-19 thru 10-22	Replace
11-1 thru 11-4	Replace
11-4a	Add
G-5 and G-6	Replace
G-13 thru G-20	Replace
Index-1 thru Index-3	Replace

Technical changes to text, tables, and figures are marked with a vertical bar in the outer margin.

Pages containing *non-technical* changes (page layout, spelling corrections) are indicated by a bar opposite the page number.

Please file this bulletin with the publication to retain a record of changes.

Sequence Number: M102

MEMOREX

Computer System Products

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
	Function of the Assembler	1-1
	Relationship to the Operating System	1-2
	System Requirements	1-2
2	WRITING SOURCE STATEMENTS	2-1
	Character Set	2-1
	Basic Format of Source Statements	2-3
	Terms and Expressions	2-3
	Terms	2-4
	Constants	2-5
	String Constants	2-5
	Character String Constant	2-5
	Hexadecimal String Constant	2-6
	Packed Decimal String Constant	2-6
	Zoned Decimal String Constant	2-6
	Integer String Constant	2-6
	Arithmetic Constants	2-7
	Decimal Arithmetic Constant	2-7
	Hexadecimal Arithmetic Constant	2-8
	Symbols	2-8
	Ordinary Symbols	2-9
	Variable Symbols	2-10
	Concatenation of Variable Symbols	2-10
	Sequence Symbols	2-11
	Location Counter Reference	2-11
	Symbol Length Attribute	2-12
	Literals	2-13
	Expressions	2-14
	Evaluation of Expressions	2-15
	Absolute and Relocatable Expressions	2-17
	Absolute Expressions	2-17
	Relocatable Expressions	2-18
	Examples of Absolute and Relocatable Expressions	2-18
	Coding Form	2-18
	Name Field	2-10
	Operation Field	2-20
	Operand Field	2-20
	Comment Field	2-20
	Identification-Sequence Field	2-20
	Statement Continuation	2-21

TABLE OF CONTENTS (Continued)

Section		Page
3	MACHINE INSTRUCTIONS	3-1
	Source Statements	3-1
	Instruction Alignment and Checking	3-1
	Operands and Suboperands	3-1
	Name and Length Attributes	3-2
	Notation Used to Describe Machine Instructions	3-2
	Summary of Machine Instructions	3-3
	General-Purpose Instructions	3-5
	System Instructions	3-9
	Summary of Extended Mnemonics	3-10
	Extended Mnemonic Codes	3-11
4	ASSEMBLER INSTRUCTION SOURCE STATEMENTS – OVERVIEW	4-1
5	PROGRAM SECTIONING AND LINKING STATEMENTS	5-1
	CSECT – Identify Control Section	5-1
	Symbolic Linkage Statements – ENTRY and EXTRN	5-2
	ENTRY – Identify ENTRY Point SYMBOL	5-2
	EXTRN – Identify External Symbol	5-3
	COM – Define Common Control Section	5-4
	Reserved Symbolic Segment Name – $SYSEG	5-6
6	PROGRAM CONTROL STATEMENTS	6-1
	ORG – Set Location Counter	6-1
	END – End Assembly	6-3
	PUNCH – Write to File	6-3
	LTORG – Begin Literal Pool	6-4
	ICTL – Input Format Control	6-4
	ISEQ – Input Sequence Checking	6-5
	ALIGN – Align Location Counter	6-6
7	LINKAGE-EDITOR MAP DIRECTIVE – SEG	7-1
8	SYMBOL AND DATA DEFINITION STATEMENTS	8-1
	EQU – Equate	8-1
	WDD and BDD – Word and Byte Defined Data	8-2
	WRS and BRS – Word and Byte Reserve Storage	8-5
	FORM – Define Data Format	8-7
	FORM – Instruction Statement	8-7
	Padding and Truncation Rules for Form Statements	8-8

A,BHCD,52,;HERE CON	HERE is a comment. The four operands are: [A] [BHCD] [52] and [CON].

Blank

A blank signifies the end of the operand field. Characters following a blank are considered comments.

SUBLISTS IN MACRO INSTRUCTIONS

To group a number of suboperands as a single symbolic parameter value, the macro instruction operand is written as a sublist. A sublist consists of one or more suboperands separated by commas and enclosed in parentheses. Each suboperand has form identical to an operand. The entire sublist including the parentheses is considered one macro instruction operand.

Omitted suboperands have a null value. However, the operand ( ) is considered a character string, not a sublist with all the suboperands omitted. The operand limit of 127 characters applies to the entire sublist. Examples of valid sublists are:

(A,2,4,16)	Four suboperands
(A+40,B(2,6),(N,M),240/C+X)	Four suboperands

SUBLISTS IN MODEL STATEMENTS

In a macro instruction operand, a sublist can assign a set of values to a single symbolic parameter of the macro definition. Any model statement may reference the entire symbolic parameter, but only a SETA, SETC, or ADO statement can reference the suboperands of the symbolic parameter.

The format used to reference a suboperand is: symbolic parameter (n).

The subscript n is an arithmetic set expression that refers to the position of the suboperand being referenced. For example, &LIST(2) references the second suboperand of the symbolic parameter, &LIST.

If the sublist (A,,C20,'192',(N,M)) is a macro instruction operand corresponding to the symbolic parameter &FIEL D, the values assigned to the suboperands are:

FIEL D(0) = null

FIELD(1) = A

FIELD(2) = null

FIELD(3) = C20

FIEL D(4) = '192'

FIELD(5) = (N,M)

FIELD(6) through FIELD(n) = null

A subscripted reference to an omitted sublist element, such as FIELD(2) above, is assigned a null value. If the macro instruction operand is a simple operand, rather than a sublist, the macro instruction can refer to the operand value by the symbolic parameter without a subscript, or by the symbolic parameter with a subscript of 1. All other subscripted references have a null value.

Suboperands of sublists within a sublist (for example, N in FIELD(5) above) cannot be referenced in a model statement.

SUBSTRING NOTATION

Substring notation allows the programmer to reference part of a macro instruction operand or a character string constant. Substring notation consists of a symbolic parameter or a character string constant, immediately followed by two arithmetic set expressions separated by a comma and enclosed in parentheses, as in the examples:

&NAME(2,4)

C'AB24CFG'(&P1,2)

The first expression indicates the position of the first character to be included in the substring. The second expression indicates the number of consecutive characters (beginning with the character indicated by the first expression) to be included in the substring.

If the substring specifies more characters than are in the macro instruction operand or in the character string constant, only the number of available characters are supplied. If the starting character is outside the range of the string, a null character is supplied.

Substring notation has the following limitations:

1. Substrings can only be specified for character string constants and macro instruction operands.

2. A substring of a sublist cannot be specified.

3. A substring reference can only be used as a term in a conditional assembly statement.

Consider the example in Figure 10-4.

The operand field may contain an arithmetic set expression or one of the following character terms:

Term	Example
Symbolic parameter, including sublist and substring references	&PARM2(3)
SETC symbol, including substring references	&SET1
Substring of a character constant	C'ABCDE'(2,3)
&SYSECT reference	&SYSECT
Character constant	C'ABCD'
&SYSNDX reference	&SYSNDX

Only one operand is allowed. The maximum size of an assigned character value is 16 bytes. If a larger value appears, only the leftmost 16 bytes are assigned by the assembler.

When &SYSNDX is used as a single character term, the value is a string of four characters, including leading zeros. If the operand specified is an arithmetic set expression, the arithmetic value is converted to a 16-bit constant.

Examples of *valid* SETC operands are:

```
C'24BK'
&P1
&P1(2,4)
&P1&P2(2,1)
&SETC
&SYSECT
&SYSNDX
(&SYSNDX+10-&P1(2,4)/2)*&P2(2)<EQ>C'FID'
```

Examples of *invalid* SETC operands are:

X'124'	String constant other than a character constant
C'AB2'+4	Character constant used as arithmetic operand
ALPHA	Ordinary symbol
L'ALPHA	Length attribute

GBLA AND GBLC – GLOBAL ARITHMETIC AND CHARACTER SET SYMBOLS

Local set symbols are made global (available outside the macro) by a GBLA or GBLC statement. The format of the GBLA or GBLC statement is:

Name	Operation	Operand
Blank	GBLA and GBLC	1-35 set symbols, separated by commas

Global statements must appear immediately after a prototype statement or after another global statement. Any number of continuation lines may be used.

When a set symbol is declared global, its assigned value is available to statements in the main program, but not to other macro definitions. To be available to other macro definitions, the set symbols must also be declared global in those macro definitions.

If set symbols have not been assigned values outside this macro definition or by a previous call to this definition, the global statement assigns an initial value of 0 to SETA symbols and a null character value to SETC symbols. If the set symbols have an assigned value, the global statements do not affect the value of the set symbols.

ADO – ITERATIVE RETURN

The ADO statement sets up a loop between the ADO statement and a subsequent statement identified in the ADO statement. The format of the ADO statement is:

Name	Operation	Operand
Sequence symbol, set symbol, or blank	ADO	Positive arithmetic set expression, sequence symbol

The sequence symbol in the name field can only be referenced by an AGO statement.

The operand field must have two operands, separated by a comma. The first operand is a positive arithmetic set expression that indicates the number of iterations to be executed. The second operand is a sequence symbol that specifies the last statement of the loop. This sequence symbol must appear in the name field of a statement that follows, not precedes, the ADO statement. The sequence symbol cannot be in the name field of another ADO statement, a macro instruction, or an AGO statement.

If the set expression is equal to zero, control is transferred to the statement named in the second operand, and no intervening code is included. However, if the first operand is a positive number, the assembler subtracts 1 from the first operand and includes in the assembly all the code from the ADO statement to the statement named in the second operand. If the first operand is zero after the subtraction, the assembler does not return to the ADO statement but processes the next statement. If the first operand is not zero after the subtraction, the assembler again subtracts 1 from the first operand and repeats the loop.

Notice that when the first operand initially is zero, the jump is made to the statement named in the second operand. When the first operand is not initially zero, the indicated number of loops is performed and a jump is made to the first statement after the statement named in the second operand. If the first operand is invalid, the value is set to 1.

The name field may contain a sequence symbol, a set symbol, or a blank. If the name field contains a set symbol, the symbol is initially assigned a value of 1. For each iteration, this value increases by one while the first operand of ADO decreases by one. Thus, in the following example, after 10 iterations, the first operand is 0 and &COUNT is 10.

```
NAME      OPERATION      OPERAND
&COUNT    ADO            10,.END
```

The preceding statement performs a function equivalent to the following three statements. During the first iteration, &COUNT equals 1 and the first operand (10) is decreased by one. After 10 iterations, &COUNT equals 10 and the first operand equals 0. Control is then transferred to the first statement following the .END statement.

```
NAME      OPERATION      OPERAND
&COUNT    SETA           0
          ADO            10,.END
&COUNT    SETA           &COUNT+1
```

The ADO statement can be used to include or exclude code from the assembly, depending upon the value of the first operand, as in the following example. If the variable symbol &LABEL is equal to YES, the first operand of the ADO statement is 1, and all intervening code is included in the assembly. If, however, &LABEL is not equal to YES, the first operand is zero and all intervening code is excluded.

```
NAME      OPERATION      OPERAND
&SETI     ADO            &LABEL<EQ>C'YES',.A1
.A1       ANOP
```

NESTING OF ADO STATEMENTS

As many as six ADO statements can occur within a primary ADO loop. Such nesting of ADO statements increases the total number of iterations in geometric progression. If five iterations are performed by an outer ADO loop, an inner ADO with five iterations increases the number of iterations to 25. An inner ADO statement cannot reference a statement outside one referenced by the outer ADO.

The following statements produce a 5 x 5 matrix containing all products of the ordinary numbers 1 through 5.

```
NAME      OPERATION   OPERAND
&COL      ADO         5,.END
&ROW      ADO         5,.END
.END      WDD         &COL*&ROW
```

During the first iteration of the outer ADO (named &COL), &COL equals 1 and &ROW increases by 1 with each iteration of the &ROW ADO statement until &ROW equals 5. During this first iteration, the statement .END WDD &COL *&ROW produces the values 1, 2, 3, 4, and 5, as shown in the first column of values in this matrix.

First Pass of the Outer ADO

		&COL				
		1	2	3	4	5
	1	1				
	2	2				
&ROW	3	3				
	4	4				
	5	5				

During the second iteration of the outer ADO, &COL equals 2 and &ROW again equals 1 through 5. The second column of values is produced.

Second Pass of the Outer ADO

		&COL				
		1	2	3	4	5
	1	1	2			
	2	2	4			
&ROW	3	3	6			
	4	4	8			
	5	5	10			

11. CONTROL LANGUAGE STATEMENTS

The Control Language for the assembler must provide the following four basic services:

1. Call the assembler into execution
2. Specify the assembly options
3. Define source input, object output, source output, and macro library input files
4. Obtain the source program from the card reader, library member, or spooled input

The Control Language statements for the assembler are explained below. For the exact format of the statements, consult the **MRX/OS Control Language Services, Extended Reference** manual.

Control Language Statement	Parameter	Description
//EXEC PGM=	ASM	Calls the assembler into execution.
//PAR	Keyword options	Specifies the assembly options in free form.
//PAR IMEM=	Input-member-name	Specifies the name of the input source module on the library. If omitted, the source is assumed to be a non-partitioned sequential data file (such as a spooled input file).

Control Language Statement	Parameter	Description
//PAR OMEM1=	Output-member-name	Specifies the name of the relocatable object module on the library. If omitted and OBJECT=YES or COND, the assembly is aborted.
//PAR OMEM2=	Punch-indicator	1-10 alphabetic characters to indicate that PUNCH output is expected. If omitted, no PUNCH output is produced.
//PAR MAXSIZ=	1-5 decimal digits	Specifies the approximate number of source lines generated in the program. If omitted, the default value is the SYSGEN parameter, usually 1000.
//PAR LIST=	{YES / NO}	Specifies whether the source program is to be listed: YES List source program NO Omit listing If omitted, the default parameter is YES.
//PAR XREF=	{YES / NO}	Specifies whether a cross-reference list is to be generated: YES Generate cross-reference list NO Omit cross reference list
//PAR ERROR=	{YES / NO}	Specifies whether warning errors are to be listed: YES List warning errors NO Omit listing If omitted, the default parameter is YES. Fatal errors are always listed.
//PAR OBJECT=	{YES / NO / COND}	Specifies under what conditions a relocatable object module is to be generated: YES Module is generated unconditionally NO Module is not generated COND Module is generated if no fatal error occurred If omitted, the default parameter is YES. If the option, YES or COND, is selected, the OMEM1 option must be specified.

Control Language Statement	Parameter	Description
//DEF ID=	File identifier	Defines the source input, object output, source output, list output, and macro library input files.
//DEF ID=	INPUT	Source input file
//DEF ID=	OUTPUT1	Relocatable output file: must be 252 bytes, blocked 1. The file must be a partitioned data file. The device must be disc. CSD=YES
//DEF ID=	OUTPUT2	Source punch file identifier: must be 80 bytes, blocked 1. The file must be a non-partitioned sequential data set. CSD=YES
//DEF ID=	LIST	List output file: must be 132 bytes, blocked 1. The file must be a sequential data set. CSD=NO. The file is written with the first character being a "native" mode control character for a printer.
//DEF ID=	MACLIB	Macro library input: the file must be a partitioned data set, CSD format, 80 byte records, blocked 1.

If MACLIB is not specified, the default file name is $SYSMACLIB. The INPUT and LIST files must always be defined; the OUTPUT1 and OUTPUT2 files are optional. DEFINE statements may also include keyword parameters to identify the file name or the device with which the file is associated.

If the source program is to be read from the card reader, the source program card deck must be preceded by a //DATA FIL=SYSCRD statement and terminated by a /* card.

The cards in Figure 11-1 illustrate the Control Language statements to specify:

- Source input from cards
- Relocatable output to a specified library member
- List output to printer
- Cross-reference output

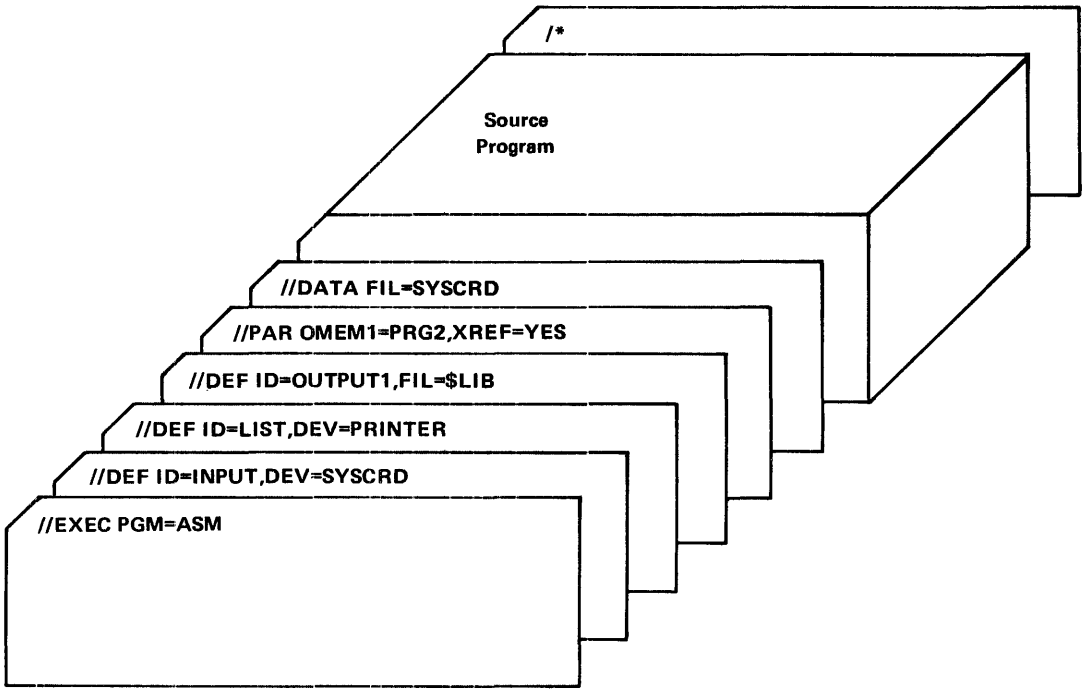


Figure 11-1. Example of Co[illegible] Language Statements

2. WRITING SOURCE STATEMENTS

To write source statements, the programmer should be familiar with the following topics:

- Character set
- Basic format of source statements
- Types of terms and expressions
- Coding form

CHARACTER SET

Source statements may contain the following characters:

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

The EBCDIC formats and card punch codes for these characters are listed in Appendix A. Any of the 256 punch combinations may appear inside a character constant, in comments, or in macro instruction operands. The meanings of these characters, and combinations of these characters, are explained in Figure 2-1.

Character	Explanation	Example
A through Z, and $	Used in symbols and character string constants	C'ACCOUNT NO.'
0 through 9	Used in numeric constants and symbols	TAG3,5825
,	Operand or suboperand separator	HERE,THERE
=	Indicates a literal term or a keyword parameter value	=A+2
C'	Defines a character constant (all characters to the next apostrophe)	C'ABC'
X'	Defines a hexadecimal string constant (all hexadecimal characters to the next apostrophe)	X'1AFEE'
P'	Defines a packed decimal string constant (all characters to the next apostrophe)	P'425'
Z'	Defines a zoned decimal string constant (all characters to the next apostrophe)	Z'-44'
I'	Defines an integer string constant (all characters to the next apostrophe)	I'4286'
"	Defines a hexadecimal arithmetic constant	"FF1A
< >	Define relational (EQ, GT, LT, NE, LE, GE) and logical (NOT, AND, OR, EOR) operations	A<EQ>B A<AND>B
L'	Defines a reference to a symbol length attribute	L'SYMX
*	Location counter reference or multiplication indicator	*+4 12*20
/	Division indicator (Note that 1/2=0 because division always results in an integer, not a fraction.)	10/0 TAG/B
+	Addition Indicator	TAG+12
-	Subtraction Indicator	TAB-4
&	Defines a variable symbol	&TAC
( )	Separates an address-modifying index from the rest of the address, delimits sublisted operands, or encloses operands or suboperands	PAG(R2)
.	Used for sequence symbols and concatenation	.LAST
.*	Used for macro definition comments	.*COMMENT
#	The character following this symbol is to be evaluated for its literal value, not for its special function. In the example, the symbol following the # sign is a semicolon, not a continuation indicator.	C'24#;4'
;	Continuation indicator	THE STATEMENT IS;
@	Indirect Addressing	@REG1,@TAG1
blank	field separator	ADDR 3,4

Figure 2-1. Character Usage

BASIC FORMAT OF SOURCE STATEMENTS

Source program statements have the fields outlined in Figure 2-2.

Name	Operation	Operand	Comment
Any symbol or blank	Machine instruction, assembler instruction, macro instruction, or FORM instruction	Single expression, several expressions, or blank	Informational material or blank

Figure 2-2. Source Statement Format

The name field entry is a symbol used to identify a statement. The name field is necessary for certain statements, or when the statement is referred to in another statement, such as in a Branch instruction.

The operation field entry is a predefined mnemonic code (or mnemonic) which identifies the function of a machine, macro, assembler, or FORM instruction. Mnemonics are designed to be easily learned and remembered; for example, ADDR for Add Register-Register, or EQU for an Equate assembler instruction.

The operand field entry defines or identifies the data involved in the operation. Most statements have one or more operands, although some statements have no operands at all. Each operand has one or more terms, which may be used in a combination to form one or more expressions. (Refer to immediately following text for a discussion of terms and expressions.) An operand field may not have more than 35 terms. Operands of machine statements generally represent storage locations, general registers, immediate data, or constant values. Operands of assembler statements provide the information necessary for the assembler to perform the designated operation.

The optional comment field contains any informational material the programmer wishes to add.

TERMS AND EXPRESSIONS

A term is a symbol, character, or number that represents a value; an expression is a single term or a combination of terms. An expression is used in the operand field of a source statement. The following text fully defines terms and expressions.

TERMS

Every term represents a value; the value may be assigned by the assembler program (symbol, symbol length attribute, location counter reference) or may be inherent in the term itself (constant, literal).

An arithmetic combination of terms is reduced to a single arithmetic value by the assembler. An arithmetic value is represented as a 16-bit binary value in two's complement form. A logical value has a range of 0 through 65,535; and an arithmetic value has a range of -32,768 through 32,767. Limitations on the value of an expression depend on its use. For example, a term designating a general register must have a value between 0-7 inclusively; a term representing an address must not exceed the size of storage.

A term is absolute if its value does not change upon program relocation. It is relocatable if its value changes upon program relocation.

The terms used in assembler statements are outlined in Figure 2-3. An explanation of each type of term and the rules for its use are provided in the following text.

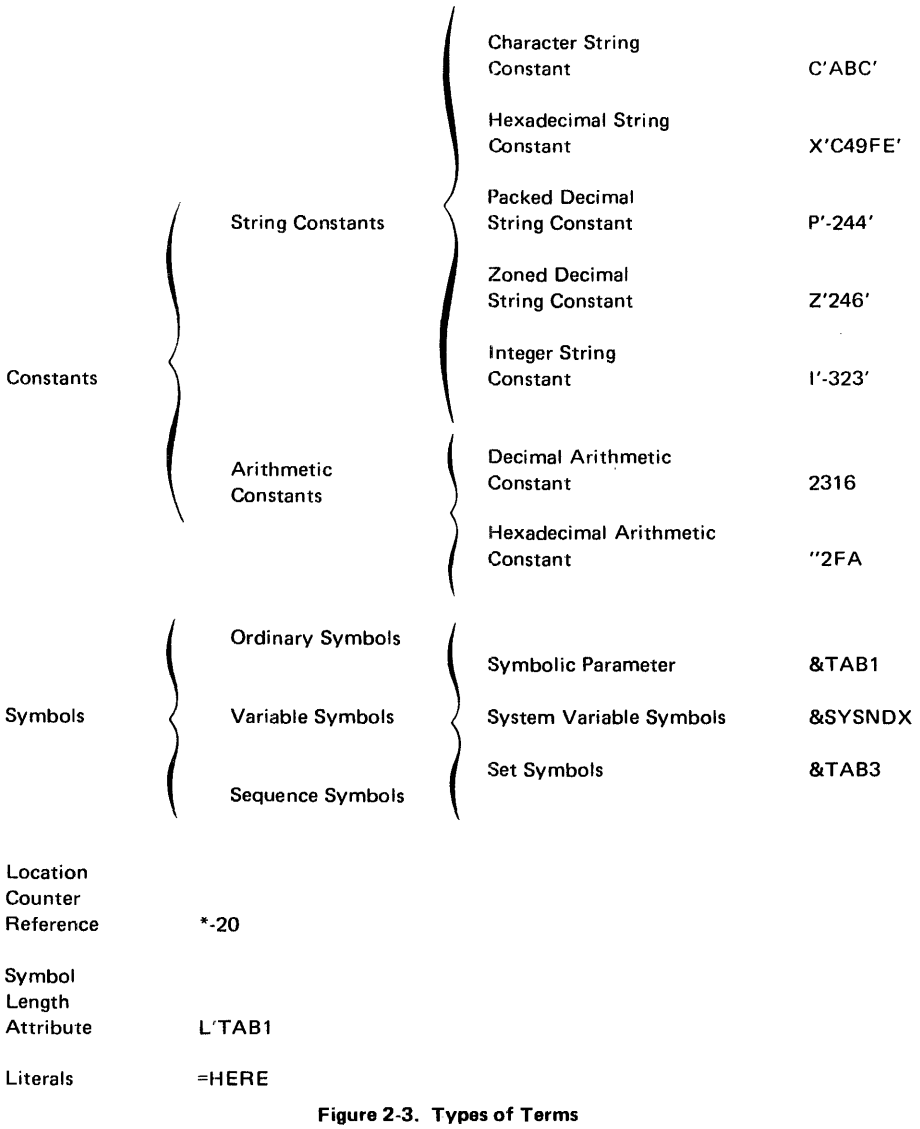


Figure 2-3. Types of Terms

CONSTANTS

Constants are terms whose values are inherent in the terms themselves. They specify machine values or bit configurations directly, rather than by equating the values to symbols and then using symbolic references. Constants represent such program elements as immediate data, masks, registers, addresses, and address increments.

Constants are string or arithmetic. String constants are of variable size; arithmetic constants are 16 bits long. Examples of all types of constants are presented in Figure 2-6.

String Constants

A string constant can only be used as a single term expression, or in a relational expression. In a relational expression, both terms must be of the same type (character, hexadecimal, etc.), for example: C'ABCE'< LT >C'&P1'.

Character String Constant

A character string constant is written as the letter C followed by a character string enclosed in apostrophes, for example: C'STRING'. To represent the literal value of an apostrophe, an ampersand, a semicolon, or a pound sign as part of the character constant, the character must be immediately preceded by an escape character, which is the pound sign. The length of a character constant is equal to the number of characters in the constant, excluding the escape characters, which do not appear in storage.

Examples of character constants are shown in Figure 2-4. In the last example of Figure 2-4, the generated code is: THIS CHARACTER STRING HAS MANY SPECIAL CHARACTERS IN IT: #;'&.

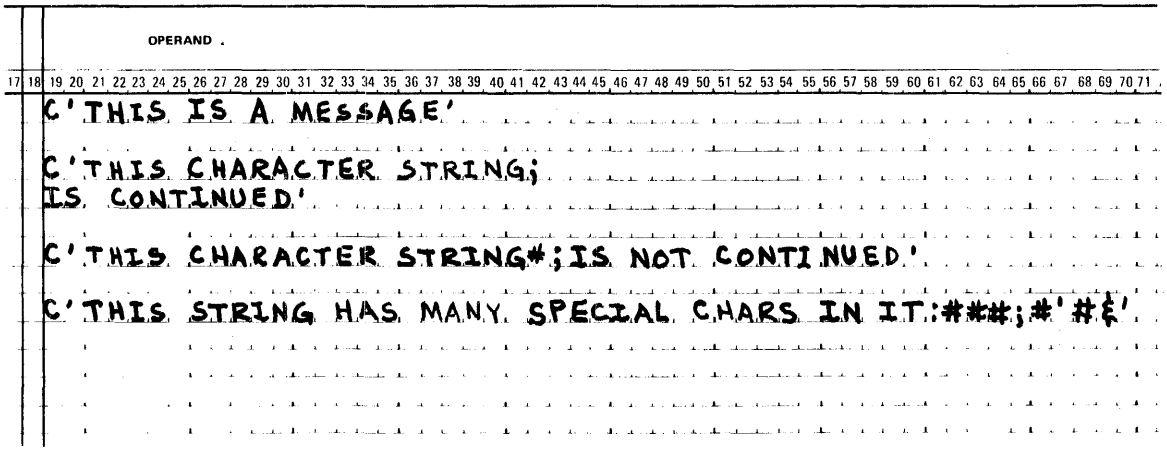


Figure 2-4. Character Constants

If the following characters are not preceded by an escape character in a character constant, they have the meaning given below:

Character	Meaning
;	A character constant is continued on the next line
'...'	Encloses the characters of a character constant
&	Variable symbol
#	Next character retains its literal value

Hexadecimal String Constant

A hexadecimal string constant is written as the letter X followed by a string of hexadecimal digits enclosed in apostrophes, such as: X'C49FE'. Each hexadecimal digit is translated into its four-bit equivalent. The maximum size of a hexadecimal string constant is limited to the maximum number of digits that can be contained on two coding lines. If an odd number of digits is specified, the leftmost four bits in the leftmost byte are set to zero. The implied length of the constant is half the number of hexadecimal digits in the constant, rounded to the next higher integer.

Packed Decimal String Constant

A packed decimal string constant is written as the letter P followed by a signed integer number enclosed in apostrophes, such as: P'-244'. If the sign is omitted, the number is assumed to be positive. Each pair of decimal digits is translated into one byte. The rightmost byte of a packed field contains the rightmost digit and the sign. Signs generated are "C_{16}" for plus, and "D_{16}" for minus. The maximum length of a packed decimal string constant is limited to the number of digits that can be contained on two coding lines.

Zoned Decimal String Constant

A zoned decimal string constant is written as the letter Z followed by a signed integer number enclosed in apostrophes, for example: Z'246'. If the sign is omitted, the number is assumed to be positive. Each decimal digit is translated into one byte. The rightmost byte contains the sign and the rightmost digit. Signs generated are "C_{16}" for plus, and "D_{16}" for minus. The maximum length of a zoned decimal string constant is limited to the number of digits that can be contained on two coding lines.

Integer String Constant

An integer string constant is written as the letter I followed by a signed integer number enclosed in apostrophes, such as: I'-246'. If the sign is omitted, the number is assumed to be positive. An integer string constant is translated into its four-byte binary equivalent. Integer constants consist of 1-10 digits with a value ranging from -2^{31} to $2^{31}-1$. The constant is word aligned when used in a WDD statement or a literal.

When string constants define data in storage, truncation and padding of their values is performed according to the rules presented in Figure 2-5.

Constant	Explicit Length= Implicit Length	Explicit Length> Implicit Length	Explicit Length < Implicit Length
Character C'ABC'	C'ABC'(3)=ABC	Left justify. Blank fill on right. C'ABC'(4)=ABC	Left justify. Truncate on right. Warning message is given. C'ABC'(2)=AB
Hexadecimal X'10A'	Right justify. Zero fill on left if the constant contains an odd number of digits. X'10A'(2)=010A	Right justify. Zero fill on left. X'10A'(3)=00010A	Right justify. Truncate on left. Warning message is given. X'10A'(1)=0A
Packed Decimal P'-24'	Right justify. Zero fill on left if the constant contains even number of digits. P'-24'(2)=024D	Right justify. Zero fill on left. P'-24'(3)=00024D	Right justify. Truncate on left. Warning message is given. P'-24'(1)=4D
Zoned Decimal Z'123'	Z'123'(3)=F1F2C3	Right justify. Zero fill on left. Z'123'(4)=F0F1F2C3	Right justify. Truncate on left. Warning message is given. F'123'(2)=F2C3
Integer I'-758'	Right justify. Propagate sign on left. I'-758'(4)=FFFFFD0A	Right justify. Propagate sign on left. I'-758'(6)= FFFFFFFFFD0A	Right justify. Truncate on left. Sign is lost. Warning message is given. I'-758'(1)=0A

Figure 2-5. Truncation and Padding of String Constant Values

Arithmetic Constants

Arithmetic constants can be used in multi-term expressions. An arithmetic constant is assembled as its two-byte binary equivalent. The maximum size of an arithmetic constant is $2^{16}-1$. If arithmetic constants are used in statements where an explicit size is specified, truncation and padding follow the same rules as those for an integer string constant.

Decimal Arithmetic Constant

A decimal arithmetic constant is written as an unsigned integer number of 1-5 digits, for example: 20.

Hexadecimal Arithmetic Constant

A hexadecimal arithmetic constant is written as quotation marks followed by a string of 1-4 hexadecimal digits, for example: "2FA. Each hexadecimal digit is assembled as its four-bit binary equivalent.

Type	Example	Generated Hexadecimal Code
Character String	C'F12AY9*' C'$Z#'## 5' C'B'	C6F1F2C1E8F95C 5BE97D7BF5 C2
Hexadecimal String	X'C49FE' X'F2' X'C'	0C49FE F2 0C
Packed Decimal String	P'14' P'925860' P'-2' P'-2596'	014C 0925860C 2D 02596D
Zoned Decimal String	Z'14' Z'925860' Z'-2' Z'-2596'	F1C4 F9F2F5F8F6C0 D2 F2F5F9D6
Integer String	I'14' I'925860' I'-2' I'-2596'	00 00 00 0E 00 0E 20 A4 FF FF FF FF FF FF F5 DC
Decimal Arithmetic	14 302 57399	000E 012E E037
Hexadecimal Arithmetic	"14 "F2A "E09F	0014 0F2A E09F

Figure 2-6. Examples of Assembled Constants

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 operands, provide the programmer with an efficient way to name and reference a program element. A symbol is defined when it appears in the name field of a source statement.

If duplicate literals are specified within one literal pool, only one literal is stored. Literals are duplicate if their final specifications, size, and repetition factors are identical on a character-by-character basis. A literal may be a duplicate even when it appears to be different (see examples in Figure 2-10). A literal is a duplicate if it contains no forward references and the expressions evaluate to the same value as the corresponding expressions of an existing literal.

A literal which contains a reference to the location counter is stored even if it duplicates another literal (see examples). If an expression used in a literal term contains a forward reference to a symbol, the symbol is assumed to represent a two-byte value.

Examples of duplicate literals are shown in Figure 2-10.

=C'ABC'(4,3) =C'ABC'(4,3)	Only one literal is stored.
=C'ABC' =X'C1C2C3'	Both literals are stored.
=A+B =B+A	Only one literal is stored if A and B are predefined symbols.
=C<EQ>D =1	Only one literal is stored if C is defined to be equal to D, so that the expression is equal to 1.
=*+10 =*+10	Both literals are stored

Figure 2-10. Examples of Duplicate Literals

EXPRESSIONS

An expression is defined as one or more terms linked by arithmetic, relational, or logical operators. Expressions may be single term or multi-term (see examples below).

Single Term Expressions	Multi-Term Expressions
29	SYMX+40
"F0	A+B/2+10
SYMX	(X<OR>"F0F0)<EQ>(SP2<OR>"F0F0)
*	(((A+4)/2+1)*2<AND>"00FF)<EQ>24)
L'SYMX	*+L'BETA
P'-240'	A+B<LE>SUM

If the length specification or the repetition factor is zero, no data is generated, but the location counter is aligned on the specified word or byte boundary.

Examples of WDD and BDD statements are shown in Figure 8-2. In the last example, if M = N, the term 35 is generated. If M = N, no data is generated, but the location counter is set on a word address, which is given the name NAM4.

```
NAME      OPERATION     OPERAND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
NAM1      WDD           C'VALUE IS'(8)
NAM2      BDD           =X'-24B'
NAM3      BDD           7(,B+1)
NAM4      WDD           35(2,M<EQ>N)
```

Figure 8-2. Examples of WDD and BDD Statements

The value attribute of the symbol in the name field is the address of the leftmost byte after alignment. The length attribute is the length in bytes (specified or implied) of the first (or only) data field in the operand.

Omitted operands, signified by a comma without a data value, indicate a zero byte or word. The last data value in a string of multiple operands must be a specified data value, not an omitted operand.

Consider the following example. Notice that an arithmetic constant, such as 45, uses two bytes.

Statement	Generated Data	
WDD ,12,,,45	00 00	Word 1
	00 0C	Word 2
	00 00	Word 3
	00 00	Word 4
	00 2D	Word 5

If the data value is a relocatable expression other than $SYSEG, the length specification and the repetition factor have the following restrictions:

1. The length attribute must be resolved to two bytes. If the length is not specified, two bytes are assumed.

2. Alignment must be on a word boundary.

For $SYSEG, the length attribute is 1 and alignment must be on an odd boundary.

Literals (which are always relocatable) in a WDD or BDD statement require special consideration. If a literal term is used in the WDD or BDD statement, the implied length and repetition attributes are (2,1). If other specifications are included, they refer to the literal term itself, but not to the symbol defined in the name field. Consider this statement:

```
NAME      OPERATION      OPERAND
B1        WDD            =C'ABCD'(5,2)
```

The length attribute of the literal C'ABCD'(5,2) is 10; but the length attribute of the symbol B1 is 2, because the operand is a relocatable term. In all other cases, the symbol in the name field receives the length attribute of the first data field in the operand.

If the location counter (*) is referenced in the operand field, the value attribute of the symbol in the name field replaces the operand. For example, TAG WDD *,*,* generates three words of data, each assigned the value attribute of TAG. If TAG is equal to to 1004, then 1004 1004 1004 is generated.

For example, the following statement specifies that two 8-byte fields of all zeros are to be generated. TAG represents the address of the first byte of generated data.

```
NAME      OPERATION      OPERAND
TAG       WDD            Ø(4,2),Ø(4,2)
```

If the location counter is pointing to 00FE, TAG is equal to 00FE and the storage locations are as follows. (The last byte is 010D or TAG+15.)

Locations	Contents	
00FE	00 00	First operand
0100	00 00	
0102	00 00	
0104	00 00	
0106	00 00	Second operand
0108	00 00	
010A	00 00	
010C	00 00	

If the location counter is pointing to an odd-byte address when a WDD statement is encountered, the assembler automatically updates the counter to the next word boundary and does not affect the contents of the odd-byte address.

In the preceding example, if the location counter is pointing to 00FD, the assembler updates the counter to 00FE and the contents of 00FD are unchanged. The storage locations are as follows.

2. Alphanumeric character constants are left-justified with blank fill on the right. If the actual data is larger than the defined field, the data is truncated on the right.

3. Packed decimal values are right-justified with zero fill on the left. If the actual data is larger than the defined field, the data is truncated on the left.

4. Zoned decimal values are right-justified with zero fill on the left. If the actual data is larger than the defined field, the data is truncated on the left.

5. Integer string values and arithmetic values are right- justified with the sign propagated on the left. If the actual data is larger than the defined field, the data is truncated on the left and the sign is lost.

The examples in Figure 8-4 illustrate certain padding and truncation rules. The first operand of TAB1, a hexadecimal 5 (0101), is truncated on the left and the two rightmost bits (01) are inserted in the 2-bit field defined by 2 in the FORM definition statement. If the value attribute of B in the second operand is less than C, the single bit position established in the corresponding definition statement is set to a binary 1. If B is greater than C, the bit is set to a binary 0.

The first A character constant (1100 0001 in EBCDIC) in the second TABLE statement is truncated on the right, and the remaining two leftmost bits (11) are assigned to location TAB2. Note, however, that all eight bits (1100 0001) of the second A character constant in the fourth operand position are retained at location TAB2+1.

NAME	OPERATION	OPERAND
TABLE	FORM	2,1,5,8,16,8
TAB1	TABLE	"5,B<LT>C,N+1,"FØ,ADDR,BYTE
TAB2	TABLE	C'A',,,C'A'

Figure 8-4. Examples of Padding and Truncation for Form Statements

The following examples show a possible use of the FORM statement — redefining instructions to create a new language closer to English. In the first example, the "MOVE" instruction generates a MOVM machine instruction, using a FORM statement and a series of Equates. This corresponds to: MOVM BUFFERB(R2),@BUFFERA(R3). Assuming that BUFFERA is at address 63FA, the code generated is:

602B

63FA

63FC

```
NAME      OPERATION   OPERAND
DIRECT    EQU         Ø
INDIRECT  EQU         1
R2        EQU         2
R3        EQU         3
MEMORY    EQU         X'6Ø'
MOVE      FORM        8,1,3,1,3,16,16
BUFFERA   WDD         C' '(2)
BUFFERB   WDD         C'5Ø'
          MOVE        MEMORY,DIRECT,R2,INDIRECT,R3,;
                      BUFFERB,BUFFERA
```

The next example shows a BRANCH FORM statement used alone to generate a BR machine instruction, or together with ROUTINE to generate a BSR machine instruction. Assuming that SUBROUT1 is at 2F3A, the code generated is:

EA07

2F3A

The last statement of this example, BRANCH REGISTER,R7, generates the code: EB07.

```
NAME      OPERATION   OPERAND
SAVERTN   EQU         X'EA'
R7        EQU         7
REGISTER  EQU         X'EB'
BRANCH    FORM        8,8
ROUTINE   FORM        16
          BRANCH      SAVERTN,R7
          ROUTINE     SUBROUT1
SUBROUT1  EQU         *
          BRANCH      REGISTER,R7
```

A detailed example of the FORM instruction can be found in Appendix F.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02042	OPERATION ENTRY MUST BE FOLLOWED BY A SPACE. The space is the delimiter of each field in the MRX Assembler.
nnnn	F	AS02043	OPERATION CODE CANNOT BE CONTINUED.
nnnn	F	AS02044	INVALID CHARACTER IN OPERATION ENTRY. Only valid characters are A-Z, 0-9, & and $.
nnnn	F	AS02045	OPERATION ENTRY TOO LONG. Only 8 characters are allowed if there is no substitution in the operation entry.
nnnn	W	AS02046	INVALID ISEQ PARAMETERS – COMMAND IGNORED. 1. The parameter value is within the begin and end limits in the statement. 2. The length of the sequence field is zero. 3. The length of the sequence field is greater than 8 characters.
nnnn	W	AS02047	MISPLACED ICTL STATEMENT. An ICTL statement must be the first statement of an assembly.
nnnn	W	AS02048	STATEMENT VAL ID ONLY WITHIN A MACRO. The following operation codes are allowed within a macro definition: GBLA GBLC MACRO MEND MNOTE MEXIT
nnnn	W	AS02049	INVALID OR MISPLACED INSTRUCTION IN MACRO DEFINITION. The following operation codes are not allowed within a macro definition: PRINT ISEQ MACRO Also GBLA, GBLC must immediately follow the macro definition prototype.
nnnn	F	AS02051	INVALID CONTINUATION. The continued statement has a continuation character as the first nonblank character.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	W	AS02052	END STATEMENT SUPPLIED BY ASSEMBLER. End of input detected but no end card received.
nnnn	W	AS02053	NAME FIELD OF MEND STATEMENT MAY ONLY CONTAIN A SEQUENCE SYMBOL.
nnnn	W	AS02054	NAME FIELD OF MACRO DEFINITION HEADER MUST BE BLANK. The name of the operation code MACRO must be blank.
nnnn	F	AS02055	REQUIRED OPERAND ENTRY MISSING. The operation code in the statement requires an operand field entry.
nnnn	F	AS02056	SYMBOLIC PARAMETER CANNOT BE USED AS A GBLA OR GBLC OPERAND.
nnnn	F	AS02057	SET SYMBOL MAY NOT BE DEFINED AS BOTH SETA AND SETC.
nnnn	F	AS02058	END QUOTE MISSING.
nnnn	F	AS02059	MACHINE AND ASSEMBLER OPERATION CODES MAY NOT BE USED AS MACRO INSTRUCTION.
nnnn	W	AS02060	MULTIPLE DEFINITION OF MACRO INSTRUCTION. Macro instruction has been previously defined.
nnnn	W	AS02061	NAME ENTRY OF MACRO PROTOTYPE STATEMENT MUST BE BLANK OR A VARIABLE SYMBOL.
nnnn	F	AS02062	MORE THAN 35 SYMBOLIC PARAMETERS. Only 35 symbolic parameters are allowed.
nnnn	W	AS02063	NAME ENTRY OF MACRO INSTRUCTION MAY NOT BE A VARIABLE SYMBOL.
nnnn	W	AS02064	STATEMENT OUT OF SEQUENCE.
nnnn	W	AS02065	INVALID ISEQ SYNTAX. In an ISEQ statement the name entry must be blank and the parameters must be separated by a comma and must be terminated by a space.
nnnn	W	AS02066	INVALID ICTL SYNTAX. In an ICTL statement the name entry must be blank.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS02111	REFERENCE TO DOUBLY-DEFINED SEQUENCE SYMBOL. The ADO or AGO statement is not processed.
nnnn	F	AS02112	INSTRUCTION NOT VALID AFTER SEG STATEMENT. The instruction is not processed.
nnnn	W	AS02114	SPECIFIED SUBSTRING – LENGTH TOO LARGE.
NOTE:	AS02115 reserved for future use.		
nnnn	F	AS02116	INVALID USE OF SEQUENCE SYMBOL.
NOTE:	AS02117 reserved for future use.		
nnnn	F	AS02118	INVALID USE OF CHARACTER STRING.
NOTE:	AS02119 reserved for future use.		
NOTE:	AS02120 reserved for future use.		
NOTE:	AS02121 and AS02122 reserved for future use.		
nnnn	F	AS02123	EXPRESSION CONTAINS INCOMPATIBLE OPERAND TYPES.
nnnn	F	AS02124	EVALUATOR – STACK OVERFLOW.
nnnn	F	AS02125	MULTIPLICATION OR DIVISION OVERFLOW.
nnnn	F	AS02126	UNDEFINED SEQUENCE SYMBOL OR VARIABLE SYMBOL.
nnnn	F	AS02127	INVALID SUBSTRING OR SUBLIST REFERENCE.
nnnn	F	AS03001	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM. This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or number of operands.
nnnn	F	AS03002	UNMATCHED RIGHT PARENTHESIS.
nnnn	F	AS03003	UNMATCHED LEFT PARENTHESIS.
nnnn	F	AS03004	EXPRESSION OR SUBLIST CONTAINS AN INVALID COMMA.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
NOTE: AS03005 reserved for future use.			
nnnn	F	AS03006	INVALID USE OF INDIRECTION *. . .insert. . .* The use of the indirect operator, @, is invalid.
nnnn	F	AS03007	INVALID EXPRESSION *. . .insert. . .* The syntax does not follow the rules for coding expressions.
nnnn	F	AS03008	INVAL ID USE OF LITERALS *. . .insert. . .* The use of the literal operator, =, is invalid in this statement.
nnnn	F	AS03009	INVAL ID USE OF PARENTHESIS *. . .insert. . .*
nnnn	F	AS03010	CHARACTER STRING INVALID WITH + - * OR / OPERATOR *. . .insert. . .* Arithmetic operations are invalid with strings.
nnnn	F	AS03011	INVALID SUBLIST *. . .insert. . .* The use of the sublist is invalid in this statement.
nnnn	F	AS03012	RELOCATABLE TERM USED IN MULTIPLICATION, DIVISION OR LOGICAL OPERATION *. . .insert. . .* The location counter, *, may not enter into the above mentioned operations.
nnnn	F	AS03013	MAY NOT FOLLOW A LOGICAL, RELATIONAL OR ARITHMETIC OPERATOR *. . .insert. . .* The unary operator NOT may not follow the above mentioned operators.
NOTE: AS03014 and AS03015 are reserved for future use.			
nnnn	F	AS03016	SYMBOL TOO LONG *. . .insert. . .* A name entry or a symbolic operand may not exceed 8 characters.
nnnn	F	AS03017	INVALID HEXADECIMAL CONSTANT *. . .insert. . .* A hexadecimal constant may only contain digits 0-9 and characters A-F.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03018	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* This message indicates internal stack overflow. To correct the condition, reduce the number of terms in the expression or the number of operands.
nnnn	F	AS03019	INVAL ID OPERATOR *. . .insert. . .* The operator or symbol is not in the language.
nnnn	F	AS03020	INVALID CHARACTER *. . .insert. . .* The character is not in the language or is contextually incorrect.
nnnn	F	AS03021	INVALID CONTINUATION *. . .insert. . .* The usage of the semicolon is contextually invalid.
nnnn	F	AS03022	INVAL ID SYMBOL *. . .insert. . .* The length attribute operand may only be symbolic.
nnnn	F	AS03023	INVALID STRING *. . .insert. . .* An invalid string structure appeared.
NOTE:	AS030024 is reserved for future use.		
nnnn	F	AS03025	INVALID STRING *. . .insert. . .* Same as AS03023 above.
nnnn	W	AS03026	OPERAND TRUNCATED – TOO LONG *. . .insert. . .* The number of digits exceeds the maximum allowed. For a decimal integer, 5 is the maximum. For an integer string, 10 is the maximum. For a hexadecimal constant, 4 is the maximum.
nnnn	W	AS03027	VALUE TRUNCATED – EXCEEDS PERMISSIBLE MAGNITUDE *. . .insert. . .* A decimal integer may not exceed 65,535. An integer string may not exceed 4,294,967,295.
NOTE:	AS03028 and AS03029 are reserved for future use.		
nnnn	W	AS03030	INVALID USE OF NAME ENTRY.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03031	INVALID FORM DEFINITION. A form reference cannot be used as an operand entry.
nnnn	F	AS03032	INVALID USE OF $SYSEG. $SYSEG is invalid in an evaluated expression.
nnnn	F	AS03033	RELOCATABLE TERM INVALID WITH UNARY OPERATOR. A relocatable term may not be used in conjunction with a NOT operator.
nnnn	F	AS03034	INVALID USE OF STRING CONSTANT. A string constant may not be used as a term in conjunction with an arithmetic or logical operator.
nnnn	F	AS03035	RELOCATABLE TERM INVALID WITH */ OR LOGICAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a multiplication, division, or logical operation.
nnnn	F	AS03036	INVALID SYNTAX. The operand structure does not follow the rules of the language.
nnnn	F	AS03037	MORE THAN SIX RELOCATABLE TERMS. An expression may not contain more than six unresolved relocatable terms.

NOTE: AS03038 is reserved for future use.

nnnn	F	AS03039	INVALID STRING *. . .insert. . .* Both strings in a relational or logical operation must be the same type.
nnnn	F	AS03040	OPERAND SIZE OR NUMBER OF TERMS EXCEEDS MAXIMUM *. . .insert. . .* The expression size is too large to be evaluated.
nnnn	F	AS03041	MULTIPLICATION OR DIVISION OVERFLOW *. . . insert . . .* Either division overflow has occurred or the second term of a multiplication or division operation exceeds a 16-bit value.

NOTE: AS03042 is reserved for future use.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03043	RELOCATABLE TERM INVAL ID WITH RELATIONAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a relational operator.
nnnn	F	AS03044	UNDEFINED SYMBOLIC OPERAND. The reference is not used as a label within this program.

NOTE: AS03045 is reserved for future use.

NOTE: AS03046 through AS03050 are reserved for future use.

nnnn	F	AS03051	DUPLICATE DEFINITION OF NAME ENTRY. *. . .insert. . .* The name field entry definitions must be unique. All duplicates are discarded.
nnnn	F	AS03052	ENTRY POINT DEFINITION IS NOT RELOCATABLE *. . .insert. . .* The entry point definition must resolve to a relocatable term.
nnnn	F	AS03053	CSECT NAME IS ALREADY DEFINED, BUT NOT AS CSECT *. . .insert. . .* Control section names must not appear as ordinary name field entries.
nnnn	F	AS03054	COM NAME IS ALREADY DEFINED, BUT NOT COM *. . .insert. . .* COM names must not appear as ordinary name field entries.

NOTE: AS03055 is the same as AS03052.

NOTE: AS03056 through AS03058 are reserved for future use.

nnnn	F	AS03059	DUPLICATE FORM DEFINITION. *. . .insert. . .* The form definition name entry is previously defined.
nnnn	F	AS03060	INVALID SYNTAX IN STORAGE RESERVATION. The operand of a reserve storage instruction must be preresolved, absolute, positive arithmetic expression. Only one operand is allowed.
nnnn	F	AS03061	INVALID SYNTAX IN DATA DEFINITION. The syntactical structure of the data definition operand is invalid.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03062	INVALID USE OF INDIRECTION. An indirect operator is invalid in data definition operands.
nnnn	F	AS03063	INVALID USE OF LITERAL. The literal operator is invalid in size or repetition field of a data definition operand.
nnnn	F	AS03064	INVALID SIZE SPECIFICATION. The size operand of data definitions must be preresolved absolute expression.
nnnn	F	AS03065	INVALID REPETITION FACTOR. The repetition factor of a data definition operand must be a preresolved absolute expression.
nnnn	F	AS03066	VALUE OF LOCATION COUNTER EXCEEDS 65,535.
nnnn	W	AS03067	TRUNCATION OCCURRED. The implied size of the value operand is greater than the explicit size operand in a data definition.

NOTE: AS03068 through AS03069 are reserved for future use.

nnnn	F	AS03070	INVALID USE OF $SYSEG. The data following $SYSEG definition must be two bytes long, word aligned, and relocatable.
nnnn	F	AS03071	REQUIRED OPERAND ENTRY MISSING. This instruction requires an operand and none was supplied.
nnnn	F	AS03072	INVALID SUBLIST. The syntax indicates a suboperand, but the instruction does not allow suboperands.
nnnn	F	AS03073	INVALID USE OF LITERAL. The instruction does not allow a literal as an operand, but one was coded.
nnnn	F	AS03074	INVALID USE OF INDIRECTION. The instruction does not allow indirection, but indirection was coded.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03075	INVALID EXPRESSION. The expression coded does not fall within the types allowed. 1. A string type was used where only arithmetic type expressions are allowed. 2. An unresolved expression was coded on an instruction which required expressions to be predefined. 3. A relocatable expression is coded where only absolute are allowed.
nnnn	W	AS03076	NAME FIELD OF ORG STATEMENT MAY ONLY CONTAIN A SEQUENCE SYMBOL.
nnnn	W	AS03077	INVALID RELOCATION. 1. An absolute value was coded where a relocatable value was required. 2. The relocation identifier does not match the relocation identifier for the control section in effect; e.g., trying to ORG to another CSECT or COM section.
nnnn	F	AS03078	NAME ENTRY REQUIRED ON AN EQU STATEMENT.
nnnn	W	AS03079	TOO MANY OPERANDS. More than the maximum number of operands allowable for this instruction were coded. The values of the first operands were used.
nnnn	F	AS03080	ONLY SINGLE TERM RELOCATABLE EXPRESSIONS ARE VALID.
nnnn	F	AS03081	COMBINED CSECT, COM AND EXTRN COUNT EXCEEDS 252. The binary generated will probably not be useless. Reduce the number of EXTERNs and CSECTs and COMs and reassemble.
nnnn	W	AS03082	INVALID NAME ENTRY. The name entry was coded where none was allowed. The name entry has not been entered into the symbol table. Any reference to it will result in an undefined reference.
nnnn	F	AS03083	REFERENCE TO INVALID FORM DEFINITION. 1. The operation entry matches something other than a FORM definition. 2. There was no such entry. 3. No operand was coded on the reference.

LINE NUMBER	ERROR TYPE	ERROR CODE	MESSAGE TEXT
nnnn	F	AS03084	OPERAND SIZE EXCEEDS 255 BITS. A FORM definition was coded with an operand which did not resolve to an absolute value less than 255.
nnnn	F	AS03085	DUPLICATE DEFINITION OF NAME ENTRY.
nnnn	F	AS03086	NAME ENTRY REQUIRED ON A FORM DEFINITION.
nnnn	F	AS03087	CSECT NAME MAY NOT DUPLICATE COM NAME. The name entries of CSECT and COM statements cannot duplicate each other in the same assembly.
nnnn	W	AS03088	ONLY ONE TITLE STATEMENT IN A PROGRAM MAY HAVE A NONBLANK NAME FIEL D.
nnnn	F	AS03089	INVALID PRIME ENTRY POINT. 1. The operand of an END statement does not resolve to an even boundary. 2. The operand does not reference a relocatable value.
nnnn	F	AS04001	INVALID FORM DEFINITION. A form reference cannot be used as an operand entry.
nnnn	F	AS04002	INVALID USE OF $SYSEG. $SYSEG is invalid in an evaluated expression.
nnnn	F	AS04003	RELOCATABLE TERM INVALID WITH UNARY OPERATOR. A relocatable term may not be used in conjunction with a NOT operator.
nnnn	F	AS04004	INVALID USE OF STRING CONSTANT. A string constant may not be used as a term in conjunction with an arithmetic or logical operator.
nnnn	F	AS04005	RELOCATABLE TERM INVALID WITH */ OR LOGICAL OPERATOR *. . .insert. . .* A relocatable term may not be used in conjunction with a multiplication, division, or logical operation.
nnnn	F	AS04006	INVALID SYNTAX. The operand structure does not follow the rules of the language.

INDEX

Abort messages G-25
Absolute
  expressions, definition of 2-17
  terms, definition of 2-4
Addressing in machine instructions 3-2,3-3
ADO statement 10-20
AGO statement 10-23
ALIGN statement 6-6
Alignment
  of data 8-2
  of machine instructions 3-1
Alphabetical list of machine instructions C-1
ANOP statement 10-23
Arithmetic
  constants 2-7
  operators 2-16
  set expressions 10-16
Assembler instructions
  definition 1-1
  overview 4-1
  summary E-1
Assembly options 11-1

BDD statement 8-2
Begin-end columns
  alteration of 6-4
  description 2-18
BRS statement 8-5
Byte reserve storage 8-5
Byte defined data 8-2

Calling assembler 11-1
Card codes A-1
Character
  codes A-1
  set 2-1
  set expression 10-18
  string constants 2-5
Coding form
  format of 2-18
  statement continuation 2-21
COM statement 5-4
Comments 2-20
Common control sections (see COM statement)
Conditional assembly statements
  ADO 10-20
  AGO 10-23
  ANOP 10-23
  GBLA 10-20
  GBLC 10-20
  SETA 10-16
  SETC 10-18
Constants
  arithmetic 2-7
  string 2-5
Continuation of statements 2-21
Control language for assembler 11-1
Control sections
  and location counter 2-11
  assembler statements for 5-1
Count attribute of macro instruction operand 10-24
Cross reference list, suppressing 11-2
CSECT statement 5-1

Data definition statements 8-1
Diagnostic messages G-1

EBCDIC table A-1
EJECT statement 9-2
END statement 6-3
ENTRY statement 5-2
EQU statement 8-1
Error messages
  abort messages G-25
  diagnostic messages G-1
  system messages G-26

Expressions
  absolute 2-17
  definition 2-14
  evaluation of 2-15
  relocatable 2-18
Extended mnemonics 3-10
EXTRN statement 5-3

FORM definition statement 8-7
FORM instruction statement 8-7
GBLA statement 10-20
GBLC statement 10-20
General purpose machine instructions 3-3,3-5
Global arithmetic and character set symbols 10-20
Hex codes of machine instructions D-1
Hexadecimal string constants 2-6

ICTL statement 6-4
Identification-sequence field 2-20
Index registers 3-2,3-3
Input format control 6-4
Integer string constants 2-6
ISEQ statement 6-5

Job control language (see Control language)

Linkage editor
  and control sections 5-1
  and symbol linkage 5-2
  map directive (SEG) 7-1
Linking statements 5-1,5-2
Listing control statements
  EJECT 9-2
  PRINT 9-3
  SPACE 9-2
  TITLE 9-1
Literal pools
  and LTORG statement 6-4
  description 2-13
Literals
  description 2-13
  in WDD and BDD statements 8-3

Location counter
  and ALIGN statement 6-6
  and ORG statement 6-1
  and WDD and BDD statements 8-4
  description 2-11
  reference (asterisk) 2-11
Logical operators 2-16
LTORG statement 6-4

Machine instructions
  alphabetical list C-1
  definition 1-1
  hex code to mnemonic D-1
  object formats B-1
  summary 3-1
Macro language
  concatenation of variable symbols 10-11
  count attribute 10-24
  example F-1
  file definition 11-1
  general description 10-1
  macro definition 10-1
  macro instruction 10-5
  MEXIT statement 10-13
  MNOTE statement 10-12
  nesting of macros 10-13
  number attribute 10-25
  sublists in macro instructions 10-9
  sublists in model statements 10-9
  substring notation 10-10
  system variable symbols (&SYSNDX, &SYSECT) 10-13
Messages, error G-1
Mnemonic definition (FORM) 8-7

Name field, description 2-3
Notation used to describe machine instructions 3-2
Number attribute of macro instruction operand 10-25

Object formats of machine instructions B-1
Object program
  definition 1-1
  file definition 11-1

Operand field, description 2-3,2-20
Operating system, relationship to assembler 1-2
Operation codes (hex) for machine instructions D-1
Operation field, description 2-3,2-20
Operators 2-16
Ordinary symbols 2-9
ORG statement 6-1

Packed decimal string constants 2-6
PRINT statement 9-3
Program
  control statements 6-1
  listing 1-2,9-1
  sectioning 5-1
  termination 6-3
PUNCH statement 6-3

Registers 3-2,3-3
Relational operators 2-16
Relocatable
  expressions, definition of 2-18
  symbols, identification of 5-1,5-2
  terms, definition of 2-4
Reserving storage 8-5

SEG statement 7-1
Segment names (see SYSEG)
Sequence checking statements 6-5
Sequence symbols 2-11
Set symbols 10-16
SETA statement 10-16
SETC statement 10-18
Source program
  definition 1-1
  file definition 11-1
  listing control 9-1,11-2
Source statements
  basic format 2-3
  character set 2-1
  coding form 2-18
  expressions 2-3,2-14
Source statements (continued)
  terms 2-3,2-4
SPACE statement 9-2
String constants
  character 2-5
  hexadecimal 2-6
  integer 2-6
  packed decimal 2-6
  zoned decimal 2-6
Symbol definition statements 8-1
Symbol length attribute 2-12
Symbolic linkage statements 5-2
Symbols
  definition 2-8
  ordinary 2-9
  rules for using 2-9
  sequence 2-11
  variable 2-10
SYSEG reserved name 5-6,8-3
System
  machine instructions 3-3,3-9
  messages G-26
  requirements for assembler 1-2

TITLE statement 9-1
Termination of assembly 6-3
Terms
  constants 2-5
  definition 2-3,2-4
  literals 2-13
  location counter reference 2-11
  symbols 2-8
  symbol length attribute 2-12

Variable symbols 2-10

WDD statement 8-2
Word defined data 8-2
Word reserve storage 8-5
Writing to disk file 6-3
WRS statement 8-5

Zoned decimal string constants 2-6