GC30-3003-0

Systems

IBM 3705 Communications Controller Assembler Language

Program Numbers OS 360H-TX-035
DOS 360H-TX-036

Release Number 1

IBM

Preface

This publication is a reference manual for the systems programmer, the systems engineer and the applications programmer coding in the IBM Communications Controller Assembler Language.

The publication is similar to the *IBM System/360 Operating System Assembler Language* publication, GC28-6514. The Communications Controller Assembler is similar to the OS assembler (level F) and the DOS assembler (level D). The similarity is intended to aid the Communications Controller programmer who already has OS or DOS assembler knowledge

Chapter 1 introduces the assembler language and describes the major differences between the language and the OS and DOS assembler language. Chapter 2 presents basic assembler language concepts. Chapter 3 describes instruction alignment, machine instruction mnemonics, machine formats and briefly describes the extended mnemonics. Chapter 4 discusses the instructions to the assemblers, including symbol definition, data definitions, program sectioning and linkages, symbolic linkages, base register instructions, listing control and program control instructions. Chapter 5 describes the macro language and the procedures for its use.

Appendixes A through E contain a summary of assembler language features and usage. Appendix F describes the job control language and the storage requirements necessary to produce an assembly, and Appendix G contains messages and codes helpful in debugging a program.

Before using this publication, the reader should be familiar with basic programming concepts and techniques. The prerequisite publication is *Introduction to the IBM 3705 Communications Controller,* GA27-3051. Corequisite to this publication is the *IBM 3705 Communications Controller Principles of Operation,* GC30-3004.

The contents of this publication apply to both OS and DOS users unless noted differently in the text.

Abbreviations

attrib	attribute
(B)	base register
D	displacement
E	external register
Gbl	Global
I	immediate
K'	count attribute
Lcl	Local
L'	length attribute
m	a bit position
n	name
N'	number attribute
O	operand
P	a bit position in a register
R	register
RA	register to immediate address
RE	register to external register
RI	register to immediate
RR	register to register
RS	register to storage
RSA	register to storage with additional operation
RT	register or branch or both
S.P.	symbolic parameter
S.S.	sequence symbol
T	transfer address
T'	type attribute

First Edition (June 1972)

Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

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

This manual has been prepared by the IBM Systems Development Division, Publications Center, Department E01, P.O.Box 12275, Research Triangle Park, North Carolina 27709. A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be sent to the above address. Comments become the property of IBM.

Contents

Illustrations

IBM Communications Controller programs are written in a symbolic language. Source program statements coded in this language must be translated into Communications Controller machine language before program execution. The 3705 OS and DOS assemblers are available to assemble programs written in Communications Controller assembler language. In their external structure, the Communications Controller assemblers are very similar to the IBM System/360 OS and DOS assemblers. Some of the major differences between the Communications Controller and the System/360 assemblers are:

- no literals
- no floating point arithmetic
- new operation codes

(See Appendix A for a detailed comparison of IBM assembler features, and Appendix B for a listing of the Communications Controller mnemonics.)

THE ASSEMBLER PROGRAM

The assemblers translate source statements into machine language, assign storage locations to instructions and other elements of the program, and perform auxiliary assembler functions that you can designate. These functions parallel the types of functions performed by the OS and DOS assemblers. The output of the assembler program is the object module. The object module is in the input format required by the linkage editor component of the IBM System/360 Operating System and System/360 Disk Operating System.

THE ASSEMBLER LANGUAGE

The assembler language is based on a collection of mnemonic symbols that represent:

- IBM Communications Controller machine-language operation codes.
- Auxiliary functions to be performed by the assemblers.

This language is augmented by other symbols which you can use to represent storage addresses or data. The assembler language also enables you to define and use macro instructions.

Machine Operation Codes

The assembler language consists of 51 machine instructions. These are represented to the assembler by mnemonic operation codes, usually followed by one or more operands. It also provides extended mnemonic codes for certain Branch and certain Store instructions.

The majority of the machine instructions are register-oriented. That is, they represent operations involving two registers, a register and immediate data, or a register and a storage area. The assembler converts the machine instructions into two or four bytes of object code, depending on the length assigned to the particular operation code. (See Chapter 3 and Appendix B for a detailed description.)

Auxiliary Functions and Programmer Aids

The assembler language contains mnemonic assembler instruction operation codes; you may also instruct the assembler program to perform auxiliary functions; these functions will have no effect on the machine language object program produced.

Instructions to the assembler are written as assembler pseudo operation codes, with or without operands. These instructions perform such functions as delimiting the beginning and end of sections of code, defining data areas, and specifying base registers. (See Chapter 4 and Appendix D for a detailed description.)

In addition to the above, the instructions to the assembler provides the following auxiliary functions to aid you in writing your programs:

- Variety in data representation: In writing source statements, you may use decimal, binary, hexadecimal or character representation of machine language binary values. (See Chapter 4 and Appendix C for more detail.)
- Relocatability: The assemblers allow symbols to be defined in one assembly and referred to in another, thus linking separately assembled programs. This permits both reference to data and transfer of control between programs. (See *Program Sectioning and Linking Instructions*, in Chapter 4.)
- Program listings: The assemblers produce a listing of the source program statements and the resulting object program statements it assembles. You can partially control the form and the content of each listing. (See *Listing Control Instructions*, in Chapter 4.)

- Error indications: The assembler analyzes each source program for actual and potential errors in the use of the language. Detected errors are indicated in the program listings. (See Appendix G for messages produced as a result of error.)

Macro Instructions

The macro language provides a convenient way to generate a desired sequence of assembler language statements that may be needed at more than one point in a program.

The macro language simplifies the coding of programs, reduces the chance of programming errors, and ensures that standard instruction sequences are used to accomplish desired functions.

Another facility of the macro language is called *conditional assembly*. This allows you to include in your source program some statements that may or may not be assembled, depending upon conditions evaluated at the time the program is assembled. These conditions are usually values that may be defined, set, changed, and tested during the assembly process. You may code conditional-assembly statements both within source program statements and within macro definitions. (See Chapter 5 for a more detailed description and Appendix E for a summary of the macro language.)

Uses of the Assembler

The uses of the Communications Controller assembler includes: (1) preassembling user-written block handling routines, and (2) assembling the control program generation macros and application-dependent modules during the control program generation procedure.

The assembler enables you to add to the IBM-supplied Network Control Program (NCP) modules, block handling routines (BHRs) that are unique to your applications. Using the controller assembler language, you code BHRs to process the data in message blocks going to or coming from a station. Then you use the assembler to create object modules that are stored in the same library with the IBM-supplied NCP object modules. At NCP generation time, if you have coded the appropriate macros, the BHRs you have written are link-edited together with the IBM modules to form the NCP load module.

The assembler is also used to assemble Emulation program modules during the generation procedure. While the Emulation Program does not require alteration to perform its function; you could assemble and link-edit your code into the Emulation program using this assembler.

ASSEMBLER LANGUAGE CODING CONVENTIONS

The coding conventions for the Communications Controller assembler language are the same as for the OS and DOS assembler languages. For a review of these conventions, see *IBM System/360 Operating System Assembler Language,* GC28-6514-6.

ASSEMBLER LANGUAGE STRUCTURE

The basic structure of the language is as follows:

A source statement comprises:

- A name entry (usually optional). Must begin in column one and end before column nine. The name entry must begin with an alphabetic character
- An operation entry (required). Must be preceded and followed by a blank
- An operand entry (usually required). Must be preceded and followed by a blank.
- Comments entry (optional)

A name entry is:

- A symbol

A *operation entry* is:

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

An *operand entry* is:

- One or more operands comprising one or more expression which, in turn, contain a term or an arithmetic combination of terms

TERMS

This chapter explains how you can use terms and arithmetic combinations of terms in instruction operands.

Every term represents a value. The assembler may assign this value (symbols, symbol length attribute, location counter reference) or the value may be inherent in the term itself (self-defining term). The Communications Controller assemblers do not permit the use of literals.

The assemblers reduce an arithmetic combination of terms to a single value.

The types of terms and the rules for their use are described in the following text.

Symbols

A symbol is a character or a combination of characters used to represent locations or arbitrary values. Symbols, through their use in name fields and in operands, provide you with an efficient way to name, and to refer to a program element.

The three types of symbols are: *ordinary, variable,* and *sequence.*

- *Ordinary symbols* are used as name entries or operands, they must conform to these rules:

  The symbol must not consist of more than eight characters. The first position must be an alphabetic character; the other positions may be any combination of alphameric representation.

  A symbol can have no special character or blanks.

In the following text, the unqualified word *symbol* refers to an ordinary symbol.

- *Variable symbols* are used within the source program or marco definition to assign different values to one symbol. Begin *Variable symbols* with an ampersand (&), followed by one to seven alphameric characters, the first of which must be alphabetic. A complete description of variable symbols appear in *Chapter 5: The IBM Communications Controller Macro Language.*
- *Sequence symbols* consist of a period (.), followed by one to seven letters and/or numbers, the first of which must be alphabetic. Use sequence symbols to indicate the position of statements within the source program or macro definition. Through their use you can vary the sequence in which the assembler processes statements. A complete discussion of sequence symbols appears in *Chapter 5: The IBM Communications Controller Macro Language.*

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

A symbol used as a name entry in the Equate Symbol (EQU) assembler instruction is assigned the value designated in the operand entry of the instruction. Since the operand

entry may represent a relocatable value or an absolute (that is, unchanging) value, the symbol is considered a relocatable term or an absolute term, depending upon the value it is equated to.

A symbol used as a name entry in the Equate Symbol to Register Expression (EQUR) assembler instruction is assigned the value of the grouping in the operand field. A register expression defines a particular byte of a register. The symbol is considered to be neither absolute nor relocatable. Its occurrence in an expression is governed by the special rules described under *EQUR - Equate Symbol to Register Expression,* in Chapter 4.

The value of a symbol may not be negative and may not exceed 2^{18}-1, or 262,143.

> *Note:* The assembly program always checks to see that the value of a symbol is not negative and not larger than 2^{18}-1. However, depending upon the model of the Communications Controller being programmed, this may cause a problem. The extended addressing feature on the larger models (for example, the IBM 3705 Communications Controller Model 3) with storage capacity greater than 65,535 allows you to address more storage, but for a model 2 the maximum you can safely address is 65,535. The difference between the limit of storage and the maximum address allowable in the register is an area which will cause an addressing exception. See *Introduction to the IBM 3705 Communications Controller,* GA27-3051-0 for a discussion of models and storage capacities by model. For a discussion of extended addressing, storage addressing, and address exception, see *IBM 3705 Communications Controller Principles of Operation,* GC30-3004.

A symbol is said to be defined when it appears as the name of a source statement. (A special case of symbol definition is described under *Program Sectioning and Linking Instructions,* in Chapter 4.)

Symbol definition also involves the assignment of a length attribute to the symbol. (The assembler maintains an internal table--the symbol table--in which the values and attributes of symbols are kept. When the assembler encounters a symbol in an operand, it refers to the assembler tables for the value associated with the symbol.) The length attribute of a symbol is the length, in bytes, of the storage field whose address is represented by the symbol. There are exceptions to this rule: for example, in the case where a symbol has been defined by an equate to location counter value (EQU *) or to a self-defining term, the length attributes of the symbol is 1. These and other exceptions are noted under the applicable instructions. Regardless of the number of times the constant is generated, the length attribute is never affected.

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

Self-Defining Terms

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

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

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

Using Self-Defining Terms: Self-defining terms are the means of specifying machine values or bit configurations without equating the values to symbols and using the symbols.

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

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

Decimal Self-Defining Term: A decimal self-defining term is an unsigned decimal number written as a sequence of decimal digits. High-order zeros may be used (for example, 007). A decimal self-defining term is assembled as its binary equivalent. A decimal self-defining term may not consist of more than six digits or exceed 262,143 (2^{18}-1). Some examples of decimal self-defining terms are: 8, 147, 4092, and 00021.

Note: For models without extended addressing, a decimal self-defining term may not consist of more than four digits or exceed 65,535 (2^{16}-1). See also *Extended Addressing, Storage Addressing,* and *Address Exception* in the publication, *IBM 3705 Communications Controller Principles of Operation,* GC30-3004.

Hexadecimal Self-Defining Term: A hexadecimal self-defining term consists of one to five hexadecimal digits enclosed by apostrophes and preceded by the letter X: X'C49' A hexadecimal term may not exceed X'3FFFF' (2^{18}-1).

Note: For models without extended addressing, a hexadecimal term may not exceed X'FFFF' (2^{16}-1).

Binary Self-Defining Term: A binary self-defining term is written as an unsigned sequence of 1s and 0s enclosed in apostrophes and preceded by the letter B, as follows: B'10001101'. This term would appear in storage as shown, occupying one byte. A binary term may have up to 18 bits represented, or as noted above, 16 bits without extended addressing.

Character Self-Defining Term: A character self-defining term consists of one or two characters enclosed by apostrophes. It must be preceded by the letter C. All letters, decimal digits, and special characters may be used in a character term. In addition, any of the remainder of the 256 EBCDIC characters may be designated in a character self-defining term. Examples of character self-defining terms are as follows:

```
C'/'    C' ' (blank)    (apostrophes are a 5-8 punch)
C'AB'   C'13'
```

Because of the use of both *apostrophes* in the assembler language and *ampersands* in the macro language as syntactic characters, observe the following rule when using these characters in a character term.

For each apostrophe or ampersand desired in a character self-defining term, you must write two apostrophes or ampersands. For example, you code the character value A' as 'A'''; for an apostrophe followed by a blank, you code ''' '. Code an ampersand && in order for one & to be a self-defining term.

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

Location Counter Reference

The Location Counter: A location counter is used to assign storage addresses to program statements. As each machine instruction or data area is assembled, the location 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 storage location. If the statement is named by a symbol, the value attribute of the symbol is the value of the location counter after boundary adjustment, but before addition of the length.

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

The location counter setting can be controlled by using the START and ORG assembler instructions. The counter affected by either of these assembler instructions is the counter for the control section in which they appear. The maximum value for the location counter is 2^{18}-1.

You may refer to the current value of the location counter at any place in a program, by using an asterisk as a term in an operand. The asterick represents the location of the first byte of currently available storage (that is, after any required boundary adjustment). Using an asterisk as the operand in a machine-instruction statement is the same as placing a symbol in the name field of the statement and then using that symbol as an operand of the statement. Because a location counter is maintained for each control section, a location counter reference designates the location counter for the section in which the reference appears. A location counter reference may not be used in a statement which requires the use of a predefined symbol, with the exception of the EQU and ORG assembler instructions.

Symbol Length Attribute Reference

The length attribute of a symbol (the length in bytes) may be used as a term. Reference to the attribute is made by coding L', followed by the symbol, as in:

L'BETA

The length attribute of BETA will be substituted for the term.

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

Terms in Parentheses

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

Arithmetically combined terms, enclosed in parentheses,

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

14+BETA–(GAMMA–LAMBDA)

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

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

A+E–(C+D–(E+F) +10)

The innermost set of terms in parentheses is evaluated first. Five levels of parentheses are allowed; a level of parentheses is a left parenthesis and its corresponding right parenthesis. Parentheses that occur as part of an operand format do not count in this limit.

EXPRESSIONS

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

An expression is composed of a single term or an arithmetic combination of terms. The following are examples of valid expressions:

*	BETA*10
AREA 1+X'2D'	B'101'
*+32	C'ABC'
N–25	29
FIELD+332	L'FIELD
FIELD	LAMBDA+GAMMA
(EXIT–ENTRY +1) +GO	TEN/TWO

ALPHA–BETA/(10+AREA*L'FIELD)–100

The rules for coding expressions are:

- An expression cannot start with an arithmetic operator, (+–/*); Therefore, the expression –A+BETA is invalid, but the expression 0–A+BETA is valid.
- An expression cannot contain two terms or two operators in succession.
- An expression cannot consist of more than 16 terms.
- An expression cannot have more than five levels of parentheses.

Evaluation of Expression

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

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

- Each term is evaluated.
- Every expression is computed to 32 bits and then truncated to the rightmost 18 bits with extended addressing, or to 16 bits without extended addressing.
- Arithmetic operations are performed from left to right except that multiplication and division are done before addition and subtraction (for example, A+B*C is evaluated as A+(B*C), not (A+B)*C). The computed result is the value of the expression.
- Division always yields an integer result; any fractional portion of the result is dropped. For example, 1/(2*10) yields a zero result, whereas (10*1)/2 yields 5.
- Division by zero is permitted and yields a zero results.

The innermost level of parenthesized expressions is processed before the rest of the terms in the expression. For example, in the expression A+BETA*(CON–10), the term CON–10 is evaluated first, and the resulting value is used in computing the final value of the expression. Final values of expressions must be in the range of 0 through 2^{18}-1, or 2^{16}-1 without extended addressing although intermediate results have a range of -2^{31} through 2^{31}-1.

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

Absolute and Relocatable Expressions

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

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

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

Absolute Expressions: An absolute expression can be an absolute term or any arithmetic combination of absolute terms. An absolute term can be a non-relocatable symbol or any of the self-defining terms or the length attribute reference. All arithmetic operations are permitted between absolute terms.

An expression is absolute, even though it contains relocatable terms (RT), under the following conditions:

- The relocatable terms must be paired. Each pair of terms must have the same relocatability; each pair must consist of terms with opposite signs. The paired terms do not have to be contiguous (for example: relocatable term + absolute term - relocatable term).
- No relocatable term can enter into a multiply or divide operation; thus, relocatable term - relocatable term *10 is invalid, but (relocatable term - relocatable term) *10 is valid.

The pairing of relocatable terms (with opposite signs and the same relocatability) cancels the effect of relocation,

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

An absolute expression reduces to a single absolute value. The following examples illustrate absolute expressions. A is an absolute term: X and Y are relocatable terms with the same relocatability.

A–Y+X
A
A*A
X–Y+A
*–Y (A reference to the location counter must be paired with another relocatable term from the same control section; that is, with the same relocatability.)

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

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

- All relocatable terms but one must be paired. Pairing is described in the preceding text under *Absolute Expressions.*
- The unpaired term must not be directly preceded by a minus sign; -Y+X–Z is invalid.
- No relocatable term can enter into a multiply or divide operation.

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

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

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

Y–32*A	W–X+*
W–X+Y	A*A+W–W+Y
* (reference to location counter)	W–X+W
	Y

Machine instructions request the Communications Controller to perform a sequence of operations during program execution time. Machine instructions may be represented symbolically as assembler language statements. The symbolic format of each varies according to the actual machine-instruction format. Within each basic format, further variations are possible. See *Machine Instruction Examples* following, and Chapter 4 of *IBM 3705 Communications Controller Principles of Operation,* GC30-3004.

A mnemonic operation code is written in the operation field, and one or more operands are written in the operand field.

Any machine-instruction statement may be named by a symbol, which assembler statements can use as an operand. The value attribute of the symbol is the address of the leftmost byte assigned to the assembled instruction. The length attribute of an instruction having the RA format is 4. All other instructions have length attributes of 2.

INSTRUCTION ALIGNMENT AND CHECKING

The assembler aligns all machine instructions automatically, on halfword boundaries. The byte skipped due to alignment is filled with hexadecimal zeros. Expressions specifying storage addresses are checked to ensure that they refer to appropriate boundaries for instructions in which they are used. Register numbers are also checked for correctness (for example; odd-numbered registers in byte instructions). Displacements are checked to ensure proper alignment.

OPERAND FIELDS AND SUBFIELDS

Some symbolic operands are written as a single field, and other operands are written as a field followed by one or two subfields. In instructions containing two operand fields, a comma must separate the two. Subfield(s) of an operand field must be enclosed within parentheses. When two subfields are contained within parentheses, they must be separated by commas.

Fields and subfields in a symbolic operand may be represented either by absolute or by relocatable expressions, depending on what the field requires. (As defined earlier, an expression consists of one term or a series of arithmetically combined terms.) In addition, each operand field containing a byte selection may be represented with a symbolic register expression. Symbolic register expressions allow symbolic representation of specific register bytes. See Chapter 4, *EQUR.*

Note: Blanks may not appear in an operand unless they are provided by a character self-defining term. Thus, blanks may not intervene between fields and their comma separation or between parentheses and fields.

MACHINE INSTRUCTION MNEMONIC CODES

The mnemonic operation codes are designed to be easily remembered codes that indicate the functions of the Communications Controller instructions.

The first character generally specifies the function:

A–Add
B–Branch
C–Compare
I–Insert
L–Load
N–And
O–OR
S–Subtract
T–Test
X–Exclusive OR

There are four exceptions. The store function is represented by the first two characters, ST. Three functions, input, output, and exit are represented by IN, OUT, and EXIT.

The data length--C for character (8 bits) or H for halfword (16 bits) -- appears next in some instructions. Examples are:

LH Load halfword	IC Insert character
STH Store halfword	STC Store character

The letter R represents *register* notation. For instance:

AR	Add register
CCR	Compare character register
XHR	Exclusive OR halfword register

In three instructions the letter O represents *offset:*

LOR	Load with offset register
LCOR	Load character with offset register
LHOR	Load halfword with offset register

T (in ICT and STCT) or CT (in BCT) represents *count.*

M in TRM (test register under mask) represents *mask.*

In addition to the preceding machine instructions, the assembler converts a number of extended mnemonic codes

into corresponding machine instructions. See Figure 9, Extended Mnemonics.

MACHINE-INSTRUCTION EXAMPLES

The examples that follow are grouped according to machine-instruction format. They illustrate the various symbolic operand formats. (Assume that all symbols used in the examples are defined elsewhere in the same assembly.)

Implied addressing and the function of the USING assembler instruction are discussed further under *Base Register Instructions.*

RR Format

The RR instruction format denotes a register-to-register operation. See Figure 1 for the format of the RR instructions.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RR	R1(N1), R2(N2) Q1 Q2	LCR ACR SCR CCR XCR OCR NCR LCOR
	R1, R2	LHR AHR SHR CHR OHR NHR XHR LHOR LR AR SR CR XR OR NR LOR BALR

Figure 1. Register-to-Register Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

N, N1, and N2 are absolute expressions that specify a byte. The value of the expression may be 0 or 1. Zero indicates the high-order or leftmost byte. One indicates the low-order or rightmost byte. Note that for ACR, SCR, ARI, SRI, and BCT, a value of 1 for N1 or N implies bytes 0 and 1 rather than just the rightmost byte.

Q, Q1, and Q2 are symbolic register expressions that specify a register-byte combination. (See *EQUR.*)

See Figure 2 for examples of this instruction format.

Name	*Operation*	*Operand*
ALPHA1	LHR	1, 2
ALPHA2	LHR	REG1, REG2
BETA1	CR	3, 5
BETA2	CR	THREE, FIVE
GAMMA1	ACR	3(1), 5(0)
GAMMA2	ACR	HITHREE, LOFIVE

Figure 2. Examples of RR Instruction

The operands of ALPHA 1, BETA 1, and GAMMA 1 are decimal self-defining values, which are absolute expressions. The operands of ALPHA2 and BETA2 are symbols that are equated elsewhere to absolute values. The operands of GAMMA2 are symbols that are equated elsewhere to symbolic register expressions.

RS Format

The RS instruction format denotes a register-to-storage operation. See Figure 3 for the format of the RS instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RS	R(n), D(B) Q S	IC STC
	R, D(B) S	L ST LH STH

Figure 3. Register-to-Storage Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

D is an absolute expression that specifies a displacement. A value of 0-127 may be specified. (Note: The displacement for LH and STH must be a multiple of 2, and the displacement for L and ST must be a multiple of 4.)

B is an absolute expression that specifies a base register. Register numbers are 0-7.

M is an absolute expression that specifies a bit. The value of the expression may be 0-7.

N, N1, and N2 are absolute expressions that specify a byte. The value of the expression may be 0 or 1. Zero indicates the high-order or leftmost byte. One indicates the low-order or rightmost byte. Note that for ACR, SCR, ARI, SRI, and BCT a value of 1 for N1 or N implies bytes 0 and 1, rather than just the rightmost byte.

S is either an absolute or relocatable expression that specifies an implied address (used with a USING statement). The assembler selects a proper base and displacement, based on the symbol value and the USING information.

Register 0 implies direct addressable storage when used as a base register for IO, STC, LH, STH, L, and ST. Use of D (displacement) without B (base) implies register 0.

When 0 is used for the R operand in STH and ST, a constant of zeroes is stored.

See Figure 4 for examples of the RS instruction.

Name	*Operation*	*Operand*
ALPHA1	L	1, 12(4)
ALPHA2	L	REG1, ZETA (4)
BETA1	L	2, PI
BETA2	L	REG2, PI
GAMMA1	IC	3(1), 12(4)
GAMMA2	IC	HITHREE, 12(4)

Figure 4. Examples of the RS Instruction

Both ALPHA instructions specify explicit addresses; REG1 and ZETA are absolute symbols. Both BETA instructions specify implied addresses; PI represents a relocatable value. The assembler will determine the proper register and displacement values, based upon USING information. The first operand of GAMMA2 is a symbol that is equated elsewhere to a symbolic register expression.

RSA Format

The RSA instruction format denotes a register-to-storage with additional operation. See Figure 5 for the format of the RSA instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RSA	R(N) , B Q	ICT STCT

Figure 5. Register-to-Storage with Additional Operation Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

N, N1, and N2 are absolute expressions that specify a byte. The value of the expression may be 0 or 1. Zero indicates the high-order or leftmost byte; one indicates the low-order or rightmost byte. Note that for ACR, SCR, ARI, SRI, and BCT a value of 1 for N1 or N implies bytes 0 and 1, rather than just the rightmost byte.

Q, Q1, and Q2 are symbolic register expressions that specify a register-byte combination. (See *EQUR*.)

See Figure 6 for examples of the RSA instructions.

Name	*Operation*	*Operand*
ALPHA	ICT	3(1), 6
BETA	ICT	HITHREE, SIX
GAMMA	STCT	3(0), SIX
GAMMA	STCT	LOTHREE, 6

Figure 6. Examples of the RSA Instruction

SIX has been equated to an absolute value elsewhere in the program. HITHREE has been equated to a symbolic register elsewhere in the program.

RT Format

The RT instruction format denotes a branch operation. See Figure 7 for the format of the RT instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RT	R(N, M) , T Q(M)	BB
	R(N) ,T Q	BCT
	T	B BCL BZL

Figure 7. Branch Operation Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

M is an absolute expression that specifies a bit. The value of the expression may be 0-7.

N, N1, and N2 are absolute expressions that specify a byte. The value of the expression may be 0 or 1. Zero indicates the high-order or leftmost byte; one indicates the low-order or rightmost byte. Note that for ACR, SCR, ARI, SRI, and BCT, a value of 1 for N1 or N implies bytes 0 and 1, rather than just the rightmost byte.

T is a relocatable expression that specifies a transfer address. The assembler determines the proper displacement, based upon the transfer address value and the location counter value. The relocatability of the transfer address must be the same as the relocatability of the instruction which makes reference to it as an operand; that is, they must both be associated with the same control section.

Q, Q1, and Q2 are symbolic register expressions that specify a register-byte combination. (See *EQUR.*)

See Figure 8 for examples of the RT instruction.

Name	*Operation*	*Operand*
ALPHA	BB	3(0, 6), ADDR
ALPHA1	BCT	CTR(1), ADDR1
	BZL	ADDR3
GAMMA1	BB	LOFIVE (4), ADDR

Figure 8. Examples of the RT Instruction

In ALPHA1, CRT is a symbol which has been equated to an absolute value elsewhere in the program. In GAMMA1, LOFIVE is a symbol that is equated elsewhere to a symbolic register expression.

RI Format

The RI instruction format denotes a register-to-immediate operand operation. See Figure 9 for the format of the RI instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RI	R (N) , I Q	LRI ARI SRI CRI NRI ORI TRM XRI

Figure 9. Register to Immediate Operand Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

I is an absolute expression that provides immediate data. The value of the expression may be 0-255.

N, N1, and N2 are absolute expressions that specify a byte. The value of the expression may be 0 or 1. Zero indicates the high-order or leftmost byte; one indicates the low-order or rightmost byte. Note that for ACR, SCR, ARI, SRI, and BCT a value of 1 for N1 or N implies bytes 0 and 1, rather than just the rightmost byte.

Q, Q1, and Q2 are symbolic register expressions that specify a register-byte combination. (See *EQUR.*)

See Figure 10 for examples of the RI instructions.

Name	*Operation*	*Operand*
ALPHA1	NRI	3(0), X'04'
ALPHA2	SRI	3(0), FOUR
ALPHA3	ARI	REG(0), FOUR
BETA1	CRI	3(1), C'6'
GAMMA1	ARI	LOSEVEN, 22

Figure 10. Examples of the RI Instruction

FOUR and REG have been equated to absolute values elsewhere in the program. LOSEVEN has been equated to a symbolic register expression elsewhere in the program.

RA Format

The RA instruction format denotes a register-to-immediate address operation. See Figure 11 for the format of the RA instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RA	R, A	BAL LA

Figure 11. Register to Immediate Address Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only odd-numbered registers are valid.

A may be either an absolute or a relocatable expression. The value of the expression may range from 0 to 2^{16}-1, or with extended addressing from 0 to 2^{18}-1.

See Figure 12 for examples of the RA instruction.

Name	*Operation*	*Operand*
ALPHA1	LA	3, 1000
ALPHA2	LA	3, ADDR1
BETA1	BAL	4, X'240'
BETA2	BAL	4, ADDR2

Figure 12. Examples of the RA Instruction

The ALPHA1 and BETA1 instructions specify absolute addresses. The addresses in the ALPHA2 and BETA2 instruction can be absolute or relocatable.

RE Format

The RE instruction format denotes a register-to-external register operation. See Figure 13 for the format of the RE instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RE	R, E	IN OUT

Figure 13. Register to External Register Format

R, R1, and R2 are absolute expressions that specify general registers. The general register numbers are 0 through 7. Note that for all instructions allowing byte selection, only the odd-numbered registers are valid.

E is an absolute expression that specifies an external register. The value of the expression may be 0-127.

An external register is a register in the Communications Controller that the control program must access through input and output instructions. See External Registers in *IBM 3705 Communications Controller Principles of Operation,* GC30-3004. See Figure 14 for examples of the RE instruction.

Name	*Operation*	*Operand*
ALPHA1	IN	2, 10
ALPHA2	IN	REG2, EXTREG10
BETA1	OUT	2, X'3F'
BETA2	OUT	REG2, EXTREG96

Figure 14. Examples of the RE Instruction

The operands of the ALPHA1 and BETA1 instructions are decimal self-defining values. The operands of ALPHA2 and BETA2 are symbols that are equated elsewhere to absolute values.

EXIT Format

The EXIT instruction format denotes an exit from the active program level. See Figure 15 for the format of the EXIT instruction.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
EXIT		EXIT

Figure 15. Exit Format

See *Chapter 4: Instruction Set,* in *IBM 3705 Communication Controller Principles of Operations.*

EXTENDED MNEMONIC CODES

For the convenience of the programmer, the assembler provides extended mnemonic codes. The codes are not part of the set of machine instructions, but are translated by the assembler into the corresponding operation and condition combinations.

The allowable extended mnemonic codes, their operand formats, and their machine-instruction equivalents are shown in Figure 16, Extended Mnemonics.

Extended Code	*Meaning*	*Equivalent Machine Instruction*
BR R2	Branch Register	LR 0, R2
NOP	No Operation	B *+2
BND D (B)	Branch Indirect	L 0, D(B)
BND S	Branch Indirect	L 0, S
BLG A	Branch Long	BAL 0, A
BBE R (P), T	Branch on Bit Extended	BB R (0,P), T for P<8
	or	BB R (1,P-8), T for P≥8
STZ D (B)	Store Zeros	ST 0, D(B)
STZ S	Store Zeros	ST 0, S
STHZ D (B)	Store Halfword Zeros	STH 0, D(B)
STHZ S	Store Halfword Zeros	STH 0, S
	Used After Compare instructions:	
BE T	Branch on Equal	BZL T
BL T	Branch on Low (that is, branch if the first operand is less than second operand)	BCL T
	Used after Add instructions:	
BO T	Branch on Overflow	BCL T

Figure 16. Extended Mnemonics

Note: In the BBE extended code, P represents an absolute expression that specifies a bit in byte 0 or 1 of a register. The value of the expression must be between 0 and 15. All other operand values have the same meaning, as in the standard machine instruction format.

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

SYMBOL DEFINITION INSTRUCTIONS

EQU - Equate Symbol

The EQU instruction is used to define a symbol by assigning to it the length, value, and relocatability attributes of an expression in the operand field. See Figure 17 for the format of the EQU statement.

Name	*Operation*	*Operand*
variable symbol or ordinary symbol	EQU	an absolute or relocatable expression

Figure 17. EQU Statement Format

The expression in the operand field may be absolute relocatable. Any symbols oppearing in the expression must be previously defined.

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

The EQU instruction is the means of equating symbols to register numbers, immediate data, and other arbitrary values. The following examples illustrate how this might be done:

```
REG2   EQU   2       (general register)
TEST   EQU   X'3F'   (immediate data)
```

EQUR - Equate Symbol to Register Expression

The EQUR instruction is used to assign a symbol to a register expression. A register expression defines a particular byte of a register. The symbol defined in the EQUR statement may be used in a symbolic machine instruction in place of an explicitly defined byte. See Figure 18 for the format of the EQUR statement.

Name	*Operation*	*Operand*
symbol	EQUR	an expression grouping of the form R (N) *or* Q

Figure 18. EQUR Statement Format

R is an absolute expression of value 1, 3, 5, or 7, and N is an absolute expression of value zero or one. Any symbols appearing in the expressions must be previously defined. Q is a previously defined symbolic register expression.

The symbol in the name field is given the value of the grouping in the operand field. The symbolic register expression is allowed only in the operands of machine instruetions or other EQUR instructions. Mixed expressions--that is, arithmetic combinations of symbolic register expressions with other symbolic register expressions or with absolute or relocatable expressions--are not allowed. The following examples are valid definitions and usages of symbolic register expressions:

```
CTR    EQUR   3(0)
       BCT    CTR,DONE
       BB     CTR(2),DONE
       BB     CTR(BIT2),DONE
CTR2   EQUR   CTR
```

DATA DEFINITION INSTRUCTIONS

There are three data definition instruction statements: Define Constant (DC), Define Storage (DS), and Define Control Word (CW).

These statements are used (1) to enter data constants into storage, (2) to define and reserve areas of storage, and (3) to specify the contents of control words. The statements can be named by symbols so that other program statements can refer to the generated fields.

DC - Define Constant

The DC instruction is used to enter constant data into storage. It can specify one constant or a series of constants. A variety of constants can be specified: fixed-point, hexadecimal, character, and storage addresses. (Data constants are generally called constants unless they are created from storage addresses, in which case they are called address constants.) See Figure 19 for the format of the DC statement.

Name	*Operation*	*Operand*
any symbol or blank	DC	one or more operands, separated by commas, written in the format described in the text.

Figure 19. DC Statement Format

DOS -- only one operand permitted.

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

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

1	2	3	4
Duplication Factor	Type	Length	Nominal Values

Although the constants specified within one operand must have the same characteristics, each operand can specify a different type of constant. For example, in a DC instruction with three operands, the first operand might specify four fixed-point constants; the second, a hexadecimal constant; and the third, a character constant.

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

The value attribute of the symbol naming the DC instruction is defined as the address of the leftmost byte (after alignment) of the first, or only, constant. The length attribute depends upon (1) the type of constant being defined, and (1) the presence of a length specification. Implied lengths are assumed for the various types of constants, in the absence of a length specification. If more than one constant is defined, the length attribute is the length in bytes (specified or implied) of the first constant.

Boundary alignment also varies according to the type of constant being specified and the presence of a length specification. Some types of constants are aligned only to a byte boundary, but the DS instruction can be used to force halfword or fullword boundary alignment for them. This is explained under *DS - Define Storage* below. Other constants are aligned on halfword or fullword boundaries, in the absence of a length specification. If length is specified, no boundary alignment occurs for such constants.

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

Any bytes skipped in aligning statements that do not cause information to be assembled are not zeroed, such as DS statements. However, bytes skipped to align a DC statement are zeroed.

Operand Subfield 1: Duplication Factor

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

A duplication factor of zero is permitted and achieves the same result as it wouldbe a DS instruction. A DC instruction with a zero duplication factor does not produce control dictionary entries. See *Forcing Alignment* under *DS - Define Storage*, following.

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

Operand Subfield 2: Type

The type subfield defines the type of constant being specified. From the type specification, the assembler determines how it is to interpret the constant and translate it into the appropriate machine format.

Figure 20 lists the type codes for constants.

Code	*Type of Constant*	*Machine Format*
C	Character	8-bit code for each character
X	Hexadecimal	4-bit code for each hexidecimal digit
B	Binary	Binary format
F	Fixed-point	Fixed-point binary format; normally a fullword
H	Fixed-point	Fixed-point binary format; normally a halfword
A	Address	value of address; normally a fullword
Y	Address	value of address; normally a halfword
R	Address	value of address; normally a halfword
V	Address	space reserved for external symbol address; each address is normally a fullword

Figure 20. Type Codes for Constants

Operand Subfield 3: Length

The length subfield is written as Ln, where n is an unsigned decimal self-defining term or an absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. The value of n represents the number of bytes of storage that are assembled for the constant. An implied length is used if a length modifier is not present. A length modifier may be specified for any type of constant, but no boundary alignment will be provided when a length modifier is given.

Operand Subfield 4: Constant

This subfield supplies the constant (or constants) described by the subfields that precede it. A data constant (C, X, B, F, H) is enclosed by apostrophes. An address constant (A, Y, R, V) is enclosed by parentheses. Two or more constants in the subfield must be separated by commas, and the entire sequence of constants must be enclosed by the appropriate delimiters (apostrophes or parentheses).

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

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

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

The types of constants are discussed in the following text.

Character Constant - C: Any of the valid 256 EBCDIC characters can be designated in a character constant. Only one character constant can be specified per operand.

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

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

- If the number of characters in the constant exceed the specified length, as many bytes as necessary are dropped from the right.
- If the number of characters is less than the specified length, the excess bytes are filled with blanks on the right.

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

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

If no length modifier is given, the implied length of the constant is half the number of hexadecimal digits in the constant (assuming that a hecadecimal zero is added to an odd number of digits). If a length modifier is given, the constant is handles as follows:

- If the number of hecadecimal digit pairs exceeds the specified length, the necessary bits (and/or bytes) are dropped from the left.
- If the number of hexadecimal digit pairs is less than the specified length, the necessary bits (and/or bytes) are added to the left and filled with hexadecimal zeros.

Binary Constant - B: A binary constant must be written, 1s and 0s enclosed in apostrophes. Only one binary constand can be specified in an operand. Duplication and length can be specified. The maximum length of a binary constant is 256 bytes.

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

Fixed-Point Constants - F and H: A fixed-point constant is written as an unsigned decimal integer. The assembler converts the decimal integer to a binary number. If the value of the number exceeds the length specified or implied, as many bits as necessary are dropped (truncated) from the left. Any duplication factor present is applied after the constant is assembled.

An implied length of four bytes is assumed for a fullword (F) and two bytes for a halfword (H), and the constant is aligned to the proper fullword or halfword boundary if a length is not specified. However, any length up to, and including, eight bytes may be specified for either type of constant by a length modifier, in which case no boundary alignment occurs.

Address Constants: An address constant is a storage address that is translated into a constant. An address constant, unlike data constants, is enclosed in parentheses.

There are four types of address constants: A, Y, R, and V.

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

A-Type Address Constant: This constant is specified as an absolute, relocatable, or complex relocatable expression. (An expression may be single-term or multi-term.) The value of the expression is calculated to 32 bits; the expression may range from -2^{31} to $2^{31}-1$. The implied length of an A-type constant is four bytes, and the alignment is to a fullword boundary unless a length is specified, in which case no alignment will occur. The length that may be specified depends on the type of expression used for the constant; a length of one to four bytes may be used for an absolute expression, while a length of only three or four bytes may be used for a relocatable or complex relocatable expression.

Y-Type Address Constant:

CAUTION
Relocatable Y-type constants must not be specified in programs destined to be executed at addresses above 65,535 in Communications Controller storage. Relocatable Y-type address constants cannot be handled by the linking editor.

A Y-type address constant has much in common with the A-type constant. It, too, is specified as an absolute relocatable, or complex relocatable expression. The value of the expression is also calculated to 32 bits. The range of the expression is -2^{15} to $2^{15}-1$; however, the maximum value of the expression can be only $2^{15}-1$. The value is then truncated, if necessary, to the specified or implied length of the field and assembled into the rightmost bits of the field.

The implied length of a Y-type constant is two bytes, and alignment is to a halfword boundary unless a length is specified, in which case no alignment will occur. The maximum length of a Y-type address constant is two bytes. If length specification is used, a length of two bytes may be designated for a relocatable or complex expression and one or two bytes for an absolute expression.

DOS - The linkage editor can process Y-type address constants.

R-Type Address Constant:

CAUTION
Relocatable, R-type constants must not be specified in programs destined to be executed at addresses above 65,535 in Communications Controller storage.

An R-type address constant has much in common with the Y-type constant. It is specified as an absolute, relocatable, or complex relocatable expression. The value of the expression is calculated to 32 bits. The range of the expression is -2^{15} to $2^{15}-1$. The implied length of an R-type constant is two bytes, and alignment is to a halfword boundary unless a length is specified, in which case no alignment will occur. The only length specification allowed is two bytes.

The primary function of the R-type constant is to provide a two-byte relocatable address constant processable by the OS linkage editor. The Linkage Editor record (RLD) generated for the R-type constant indicates a length of three (rather than two), and points to the byte preceding the constant. During linkage editing, the high-order byte (the byte preceding the R-type constant) is not disturbed as long as the constant is not relocated to a value above 65,535. Note that no R-type constant can be assembled in the two bytes of any CSECT.

V-Type Address Constant: This constant is used to reserve storage for the address of an external symbol that is used for branching to other programs. The constant may not be used for external data references within an overlay program. The constant is specified as one relocatable symbol, which need not be identified by an EXTRN statement. Whatever symbol is used is assumed to be an external symbol because it is supplied in a V-type address constant.

Note that specifying a symbol as the operand of a V-type constant does not constitute a definition of the symbol for this assembly. The implied length of a V-type address constant is four bytes, and boundary alignment is to a fullword. A length modifier may be used to specify a length of either three or four bytes, in which case no boundary alignment occurs. It must be emphasized that a V-type address constant length of less than four can and will be processed by the Communications Controller Assembler but cannot be handled by the linkage editor.

DS - Define Storage

The DS instruction is used to reserve areas of storage and to assign names to those areas. The use of this instruction is the preferred way to symbolically define storage for work areas, input/output areas, etc. The size of a storage area that can be reserved by using the DS instruction is limited only by the maximum value of the location counter. See Figure 21 for the format on the DS statement.

Name	*Operation*	*Operand*
any symbol or blank	DS	one or more operands, separated by commas, written in the format described in the text.

Figure 21. DS Statement Format

DOS - Only one operand is permitted.

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

- The specification of data (subfield 4), though mandatory is a DC operand, is *optional* in a DS instruction. If the constant is specified, it must be valid.
- The maximum length that may be specified for character (C) and hexadecimal (X) field types is 65,535 bytes, rather than 256 bytes.

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

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

Each field type (for example; hexadecimal, character, binary) is associated with certain characters, as shown in *Appendix C: Summary of Constants.* These will determine which field-type code should be selected for the operand of the DS instruction and whether length or duplication factor information should be included.

For example, the F-type field has an implied length of four bytes; the leftmost byte is aligned to a fullword boundary. Thus, you could specify an F-type field, without a length modifier, in order to reserve four bytes aligned to a fullword boundary. For an eight-byte field similarly aligned, you could specify an F-type field with a length modifier of eight. However, to reserve an F-type field larger than eight bytes (the largest you can specify with a length modifier alone), you would specify a duplication factor. Remember, however, that boundary alignment is not automatic if you specify a length modifier. See *Using the Duplication Factor to Force Alignment,* following.

Data constants of types C, X, and B have an implied

length of one byte unless the data characters are specified, in which case the assembler calculates the length (but does not assemble the data). If you wish to define a field of more than one byte, without specifying the data, you must include a length modifier.

Although no alignment occurs, field types C and X permit large data areas of up to 65,535 bytes to be defined, using the length modifier.

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

Using the Duplication Factor to Force Alignment

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

```
         DS    OH
AREA     DS    CL128
```

CW-Define Control Word

The CW instruction provides a convenient way to define and generate a four-byte control word. Control words in the Communications Controller, although fullwords in length, must be alighed on *halfword* boundaries. The CW automatically performs this alignment and causes any skipped bytes to be zeroed. See Figure 22 for the format of the CW statement.

Name	*Operation*	*Operand*
any symbol or blank	CW	four operands, separated by commas, specifying the contents of the control word in the format described in the text.

Figure 22. CW Statement Format

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

1. An absblute expression that specifies the command code. The value of this expression is placed in bits 0-1 of the control word.
2. An absolute expression that specifies the flags set in bits 2-3.
3. An absolute expression that specifies the count. The value of this expression is right-justified in bits 4-13.
4. An expression specifying the data address. This value is treated as a three byte, A-type constant. The value of this expression is in bits 14-31. The data address must be halfword-aligned.

The following is an example of a CW statement:

```
ANYNAME     CW     2,B'01',50,READAREA
```

If you code a symbol in the name field of the CW instructions, it is assigned the address value of the leftmost byte of the control word. The length attribute of the symbol is 4.

PROGRAM SECTIONING AND LINKING INSTRUCTIONS

It is often convenient, or necessary, to write a large program in sections. The sections may be assembled separately, then combined into one object program. The assembler provides facilities for creating multisectioned programs and for symbolically linking separately assembled programs or program sections. The total number of control sections, dummy sections, and external symbols must not exceed 255.

Control Section

The concept of program sectioning is a consideration at coding time, assembly time, and load time. To the programmer, a program is a logical unit. You may want to divide it into sections called control sections; if so, you write it in such a way that control passes properly from one section to another, regardless of the relative physical position of the sections in storage.

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

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

The output of the assembler consists of the assembled control sections and a control dictionary. The control dictionary contains information the linkage editor needs to complete cross-referencing between control sections as it combines them into an object program. The linkage editor can take control sections from various assemblies and combine them properly, with the help of the corresponding control dictionaries. Successful combination of

separately assembled control sections depends upon the techniques used to provide symbolic linkages between the control sections.

Whether you write an unsectioned program, a multisection program, or part of a multisection program, you know what will be entered into storage eventually because you have described storage symbolically.

Though you may not know where each section appears in storage, you will know what storage contains. There is no constant relationship between control sections. Thus, knowing the location of one control section does not make another control section addressable by relative addressing techniques.

Control Section Location Assignment

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

START--Start Assembly

The START instruction can be used to give a name to the first (or only) control section of a program. It can also be used to specify an initial location counter value for the program. This location counter value is ignored by the linkage editor. See Figure 23 for the format of the START statement.

Name	*Operation*	*Operand*
any symbol or blank	START	a self-defining term or blank

Figure 23. START Statement Format

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

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

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

```
PROG2        START    2040
PROG2        START    X'7F8'
```

Note: The START instruction may not be preceded by any code that will cause an unnamed control section to be assembled (See *Unnamed First Control Section*, following.)

CSECT--Identify Control Section

The CSECT instruction identifies the beginning or the continuation of a control section. The format is described in Figure 24.

Name	*Operation*	*Operand*
any symbol or blank	CSECT	not used; should be blank

Figure 24. CSECT Statement Format

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

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

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

Unnamed First Control Section

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

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

Common Control Sections (COM)
Dummy Control Sections (DSECT)
Marco Definitions
Conditional Assembly Instructions
Comments
COPY (depends upon the copied code)
EJECT
ENTRY
EXTRN
ICTL
ISEQ
PRINT
PUNCH
REPRO
SPACE
TITLE

No other assembler or machine instructions can precede a START instruction.

Resumption of an unnamed control section at later points can be accomplished through unnamed CSECT statements. A program can contain only one unnamed control section. It is possible to write a program that does not contain CSECT or START statements, in which case the program will be assembled as one unnamed control section.

DSECT -- Identify Dummy Section

A dummy section represents a control section that is assembled but is not part of the object program. A dummy section is a convenient means of describing the layout of an area of storage without actually reserving the storage. (It is assumed that the storage is reserved, either by some other part of the same assembly or by another assembly.) See Figure 25 for the format of the DSECT statement.

Name	*Operation*	*Operand*
variable symbol or ordinary symbol	DSECT	not used; should be blank

Figure 25. DSECT Statement Format

The DSECT instruction identifies the beginning or resumption of a dummy section. More than one dummy section may be defined in this assembly, but each must be named.

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

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

All assembler language instructions may occur within dummy sections.

Symbols that name statements in a dummy section may be used in USING instructions. Therefore, thay may be used in program elements (for example: machine-instructions and data definitions) that specify storage addresses.

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

Dummy Section Location Assignment: A location counter is used to determine the relative locations of named program elements in a dummy section. The location counter is always set to zero at the beginning of the dummy section, and the location values assigned to symbols that name statements in the dummy section are relative to the initial statement in the section.

Addressing Dummy Sections: You may wish to describe the format of an area whose storage location will not be determined until the program is executed. You can describe the format of the area in a dummy section and use symbols defined in the dummy section as the operands of machine instructions. References to the storage area may be made as follows:

1. Provide a USING statement specifying both a general register that the assembler can assign to the machine instructions as a base register and a value from the dummy section that the assembler may assume the register contains.
2. Ensure that the same register is loaded with the actual address of the storage area.

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

COM -- Define Blank Common Control Section

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

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

When several assemblies are loaded, each designating a common control section, the amount of storage reserved is equal to the longest common control section. See Figure 26 for the format of the COM statement.

Name	*Operation*	*Operand*
sequence symbol or blank	COM	blank

Figure 26. COM Statement Format

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

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

If the assignment of common storage is done in the same manner by each independent assembly, reference to a location in common by any assembly results in the same location being referred to. When assembled, blank common location assignment starts at zero.

SYMBOLIC LINKAGES

Symbols may be defined in one program and referred to in another, thus effecting symbolic linkages between independently assembled programs. The linkages can be completed only if the assembler is able to provide information about the linkage symbols to the linkage editor, which resolves these linkage references at load time. The assembler places the necessary information in the control dictionary on the basis of the linkage symbols identified by the ENTRY and EXTRN instructions.

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

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

Another way to obtain symbolic linkage is by using the V-type address constant. Information on writing V-type constants appears earlier in this chapter under *Data Definition Instructions.* It is sufficient here to note that this constant may be considered an indirect linkage point. It is created from an externally defined symbol, but that symbol does not have to be identified by an EXTRN statement.

The BAL and BALR instructions may be used with ENTRY and EXTRN statements, to branch between separately assembled control sections. The BAL instruction operand is coded in an EXTRN statement in the assembly in which the BAL appears. The BALR instruction is used by loading the branch register with a V-constant or an A-constant whose operand is identified with an EXTRN. In both cases, the branch label must be identified by an ENTRY statement in the assembly where it appears.

ENTRY -- Identify Entry-Point Symbol

The ENTRY instruction identifies linkage symbols that are defined in this program but may be used by some other program. See Figure 27 for the format of the ENTRY statement.

Name	*Operation*	*Operand*
sequence symbol or blank	ENTRY	one or more relocatable symbols separated by commas, that also appears as a statement name

Figure 27. ENTRY Statement Format

An assembly may contain a maximum of 100 ENTRY symbols. ENTRY symbols that are not defined (not appearing as statement names), although invalid, will also count towards this maximum of 100 ENTRY symbols.

The symbols in the ENTRY operand field may be used as operands by other programs. An ENTRY statement operand may not contain a symbol defined in a dummy section or blank common control section.

Note: The name of a control section does not have to be identified by an ENTRY instruction when another program uses it as an entry point. The assembler automatically places information on control section names in the control dictionary.

EXTRN -- Identify External Symbol

The EXTRN instruction identifies linkage symbols that are used by this program but which are defined in some other program. Each external symbol must be identified: this includes symbols that name control sections. See Figure 28 for the format of the EXTRN statement.

Name	*Operation*	*Operand*
sequence symbol or blank	EXTRN	one or more relocatable symbols, separated by commas

Figure 28. EXTRN Statement Format

The symbols in the operand field may not appear as names of statements in this program.

A V-type address constant does not have to be defined by an EXTRN statement.

When external symbols are used in an expression, they may not be paired. Each external symbol must be considered as having a unique relocatability attribute.

The total number of control sections, dummy sections, and external symbols in an assembly must not exceed 255.

BASE REGISTER INSTRUCTIONS

The addressing technique in certain Communications Controller instructions requires the use of a base register that contains the base address and a displacement which is added to the contents of the base register.

You may specify a symbolic or implicit address and request the assembler to determine its storage address, composed of a base register and a displacement. You can rely on the assembler to perform this service by indicating which general registers are available for assignment and what values the assembler can assume each contains. The USING and DROP instructions convey this information to the assembler.

USING -- Use Base Address Register

The USING instruction indicates that one or more general registers are available for use as base registers. This instruction also states the base address value that the assembler can assume will be in the registers at object time. A USING instruction does *not* load the registers specified. It is your responsibility to see that the specified base address values are placed into the registers. A reference to any name in a control section cannot occur in a based machine instruction before the USING statement that makes that name addressable. See Figure 29 for the format of the USING statement.

Name	*Operation*	*Operand*
sequence symbol or blank	USING	from two to eight expressions of the form v, r1, r2, r3, ... r7

Figure 29. USING Statement Format

Operand v must be an absolute or relocatable expression. Operand v specifies a value that the assembler can use as a base address. The other operands must be absolute expressions, with values between 1 and 7. The operand r1 specifies the general register that can be assumed to contain the base address represented by operand v. Operands r2 through r7 specify registers that can be assumed to contain v+128, v+256, v+384,---, respectively.

If you change the value in a base register currently being used and wish the assembler to compute displacement from this value, you must tell the assembler the new value by another USING statement. In the following example, the assembler first assumes that the value of ALPHA is in register 7. The second statement then causes the assembler to act as though ALPHA+1000 is the value in register 7.

```
USING     ALPHA,7
USING     ALPHA+1000,7
```

DROP -- Drop Base Register

The DROP instruction specifies a previously available register that may no longer be used as a base register. See Figure 30 for the format of the DROP statement.

Name	*Operation*	*Operand*
sequence symbol or blank	DROP	up to seven absolute expressions of the form r1, r2, . . . , r7

Figure 30. DROP Statement Format

The expressions indicate general registers previously named in a USING statement that are now unavailable for base addressing. The register values may range from 1 through 7. The following statement, for example, prevents the assembler from using registers 5 and 7:

```
DROP      5, 7
```

It is not necessary to use a DROP statement when the base address being used is changed by a USING statement; nor are DROP statements needed at the end of the source program.

A register made unavailable by a DROP instruction can be made available again by a subsequent USING instruction.

LISTING CONTROL INSTRUCTIONS

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

Note: TITLE, SPACE, and EJECT statements will not appear in the source listing unless the statement is continued onto another card. Then the first card of the statement is printed. However, none of these three types of statements, if generated as macro instruction expansion, will ever be listed, regardless of continuation.

TITLE - Identify Assembly Output

The TITLE instruction enables the programmer to identify the assembly listing and assembly output cards. See Figure 31 for the format of the TITLE statement.

Name	*Operation*	*Operand*
special sequence or variable symbol or blank	TITLE	a sequence of characters enclosed in apostrophes

Figure 31. TITLE Statement Format

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

The operand field can contain up to 100 characters enclosed in apostrophes. The contents of this operand field is printed at the top of each page of the assembly listing.

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

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

EJECT - Start New Page

The EJECT instruction causes the next line of the listing to appear at the top of a new page. This instruction provides a convenient way to separate routines in the program listing. See Figure 32 for the format of the EJECT statement.

Name	*Operation*	*Operand*
sequence symbol or blank	EJECT	not used; should be blank

Figure 32. EJECT Statement Format

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

SPACE - Space Listing

The SPACE instruction is used to insert one or more blank lines in the listing, see Figure 33.

Name	*Operation*	*Operand*
sequence symbol or blank	SPACE	a decimal value or blank

Figure 33. SPACE Statement Format

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

PRINT - Print Optional Data

The PRINT instruction is used to control printing of the assembly listing, see Figure 34.

Name	*Operation*	*Operand*
sequence symbol or blank	PRINT	one to three operands

Figure 34. PRINT Statement Format

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

- ON - A listing is printed.
  OFF - No listing is printed.
- GEN - All statements generated by macro instructions are printed
- NOGEN - Statements generated by macro instructions are not printed; however, the macro instruction itself will appear in the listing, with the exception of MNOTE which will print regardless of NOGEN.
- DATA - Constants are printed out in full in the listing.
- NODATA - Only the leftmost eight bytes are printed on the listing.

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

Until the first PRINT statement (if any) is encountered, PRINT, ON, NODATA, GEN is assumed.

The hierarchy of print control statements is:

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

Thus, with the following statement nothing would be printed:

```
PRINT   OFF, DATA, GEN
```

PROGRAM CONTROL INSTRUCTIONS

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

ICTL - Input Format Control

The ICTL instruction permits altering the normal format of source program statements see Figure 35. The ICTL statement must precede all other statements in the source program and can be used only once.

Name	*Operation*	*Operand*
blank	ICTL	one to three decimal self-defining values of the form *b, e, c*

Figure 35. ICTL Statement Format

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

Operand e specifies the end column of the source statement. The end column, when specified, must be within 41-80, inclusive; when not specified, it is assumed to be 71. The end column must not be less than the begin column +5. The column after the end column is used to indicate whether or not the next card is a continuation card.

DOS - The end column must not be less than the begin column +4.

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

The operand forms b,,c (no end column), and b, (no comma allowed) are invalid.

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

Example: ICTL 25 designates the begin column as 25; since the end column is not specified, it is assumed to be 71. No continuation codes will be recognized because no continue column is specified.

ISEQ -- Input Sequence Checking

The ISEQ instruction is used to check the sequence of input cards. (A sequence error is considered serious, but the assembly is not terminated.) See Figure 36 for the format of the ISEQ statement.

Name	*Operation*	*Operand*
blank	ISEQ	two decimal self-defining values of the form 1, r; or blank

Figure 36. ISEQ Statement Format

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

Sequence checking begins with the first card following the ISEQ statement. Comparison of adjacent cards makes use of the eight-bit internal collating sequence. Each card checked must have a sequence number higher than that of the preceding card.

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

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

PUNCH -- Punch A Card

The PUNCH assembler instruction causes the data in the operand to be punched into a card. As many PUNCH statements as are necessary may be used. See Figure 37 for the PUNCH statement format.

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

Figure 37. PUNCH Statement Format

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

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

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

REPRO - Reproduce Following Card

The REPRO assembler instruction causes data on the following statement line to be punched into a card. The data is not processed; it is punched in a card, and no substitution is performed for variable symbols. No sequence number or identification is punched on the card. One REPRO instruction produces one punched card. The REPRO instruction may not appear before a macro definition. REPRO statements that occur before all statements composing the first or only control section will punch cards which precede all other cards of the object deck. See Figure 38 for the REPRO statement format.

Name	*Operation*	*Operand*
sequence symbol or blank	REPRO	blank

Figure 38. REPRO Statement Format

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

ORG - Set Location Counter

The ORG instruction is used to alter the setting of the location counter for the current control section. See Figure 39 for the ORG statement format.

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

Figure 39. ORG Statement Format

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

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

An ORG statement must not be used to specify a location below the beginning of the control section in which it appears. The following is invalid if it appears less than 500 bytes from the beginning of the current control section:

```
ORG     *-500
```

To reset the location counter to the next available byte in the current control section, the following statement is used:

```
ORG
```

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

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

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

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

CNOP - Conditional No Operation

The CNOP instruction lets you align an instruction at a specific halfword boundary. If any bytes must be skipped in order to align the instruction properly, the assembler ensures an unbroken instruction flow by generating no-operation instructions. (If the CNOP is coded on as odd boundary, one byte of zero padding is generated to force the CNOP to an even boundary.)

The CNOP instruction ensures the alignment of the location counter, to a halfword, fullword, or doubleword boundary. If the location counter is already properly aligned, the CNOP instruction has no effect. If the specified alignment requires the location counter to be incremented, one to three no-operation instructions are generated, each of which uses two bytes. See Figure 40 for the CNOP statement format.

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

Figure 40. CNOP Statement Format

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

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

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

COPY - Copy Predefined Source Coding

The COPY instruction obtains source-language coding from a library and includes it in the program currently being assembled. See Figure 41 for the COPY statement format.

Name	Operation	Operand
blank	COPY	one symbol

Figure 41. COPY Statement Format

The operand is a symbol that identifies a partitioned data set member to be copied from either the system macro library or a user library concatenated to it.

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

If identical COPY statements are encountered, the coding they request is brought into the program each time. All statements include in the program via the copy function are processed using the standard format, regardless of any ICTL instructions in the program.

END - End Assembly

The END instruction terminates the assembly of a program. It may also designate a point in the program or in a separately assembled program to which control may be transferred after the program is loaded. The END instruction must always be the last statement in the source program. If an external symbol is used in the expression, the value of the expression must be 0. See Figure 42 for the END statement format.

The format of the END instruction statement is as follows:

Name	*Operation*	*Operand*
a sequence symbol or blank	END	a relocatable expression or blank

Figure 42. END Statement Format

The operand specifies the point to which control may be transferred when loading is complete. This point is usually the first machine instruction in the program.

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

INTRODUCTION

IBM Communications Controller macro language is an extension of the Communications Controller assembler language. The language provides a convenient way to generate a desired sequence of assembler language statements many times, in one or more programs. The macro definition is written only once, and a single statement, a macro instruction statement, is written each time you want to generate the desired sequence of statements.

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

An additional facility, called conditional assembly, allows you to code statements which may or may not be assembled, depending upon conditions evaluated at assembly time. These conditions are usually tests of values, which may be defined, set, changed, and tested during assembly. The conditional assembly facility may be used without using macro instruction statements.

The Macro Instruction Statement

A macro instruction statement (hereafter called a "macro instruction") is a source program statement. The assembler generates a sequence of assembler language statements for each occurrence of the same macro instruction. The generated statements are then processed like any other assembler language statement.

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

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

The Macro Definition

A macro definition is a set of statements that provides the assembler with: (1) the mnemonic operation code and the format of the macro instruction, and (2) the sequence of statements the assembler generates when the macro instruction appears in the source program.

Every macro definition consists of (1) a macro definition header statement, (2) a macro instruction prototype statement, (3) zero or more model statements. Within the definition you can code COPY statements, MEXIT, MNOTE, or conditional assembly instructions and (4) a macro definition trailer statement.

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

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

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

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

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

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

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

The Macro Library

The same macro definition may be made available to more than one source program by placing the macro definition in the macro library. The macro library is a collection of macro definitions that can be used by all assembler language programs in an installation. Once a macro definition has been placed in the macro library, it may be used by writing its corresponding macro instruction in a source program. Macro definitions must be in the system macro library under the same name as the prototype. The procedure for placing macro definitions in the macro library is described in *IBM System/360 Operating System Utilities,* GC28-6586.

DOS - The procedure for entering Disk Operating System macros in the System Source Statement Library is described in *IBM System/360 Disk Operating System: System Control and System Services Program,* GC24-5036. DOS macros are placed in the A sublibrary of the System Source Statement Library.

Varying The Generated Statements

Each time a macro instruction appears in the source program, it is replaced by the same sequence of assembler language statements. Conditional assembly instructions, however, may be used to vary the number and format of the generated statements.

Variable Symbols

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

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

Types of Variable Symbols

There are three types of variable symbols: symbolic parameters, system variable symbols, and SET symbols. The SET symbols are further broken down into SETA symbols, SETB symbols, and SETC symbols.

Assigning Values to Variable Symbols

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

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

Set symbols are assigned values by the programmer by means of conditional assembly instructions.

Global SET Symbols

The values assigned to SET symbols in one macro definition may be used to vary the statements that appear in other macro definitions. All SET symbols used for this purpose must be defined by the programmer as global SET symbols. A symbol is global when it has the same meaning throughout the entire program and all its segments. All other SET symbols (that is, those which may be used to vary statements that appear in the same macro definition) must be defined by the programmer as local SET symbols. Local SET symbols and the other variable symbols (that is, symbolic parameters and system variable symbols) are local variable symbols. Global SET symbols are global variable symbols.

MACRO DEFINITIONS

A macro definition consists of:

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

Except for MEXIT, MNOTE, and conditional assembly instructions described in the preceding text, this portion of this manual describes all of the statements that may be used to prepare macro definitions.

Macro definitions appearing in a source program must appear before all PUNCH and REPRO statements and all statements that pertain to the first control section. Specifically, only the listing control instructions (EJECT, PRINT, SPACE, and TITLE), - ICTL, and ISEQ instructions, and comment statements can occur before the macro definitions. All but the ICTL instruction can appear between macro definitions if there is more than one definition in the source program.

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

DOS - Only one continuation card for a macro definition is allowed.

MACRO -- Macro Definition Header

The macro definition header statement indicates the beginning of a macro definition. It must be the first statement in every macro definition. The format of this statement is shown in Figure 43.

Name	*Operation*	*Operand*
Blank	MACRO	Blank

Figure 43. Macro Definition Header

Macro Instruction Prototype

The macro instruction prototype statement (hereafter called the prototype statement) specifies the mnemonic operation code and the format of all macro instructions that refer to the macro definition. It must be the second statement of every macro definition. The format of this statement is shown in Figure 44.

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

Figure 44. Macro Instruction Prototype

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

The name field of the prototype statement may be blank, or it may contain a symbolic parameter.

The symbol in the operation field is the mnemonic operation code, which must appear in all macro instructions that refer to the macro definition. The mnemonic operation code must not be the same as the mnemonic operation code of another macro definition in the source program or of a machine or assembler instruction.

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

DOS - The operand field may contain 0 to 100 symbolic parameters, separated by commas.

The following is an example of a prototype statement:

```
&NAME     MOVE          &TO, &FROM
```

Statement Format

The prototype statement may be written in a format different from that used for assembler language statements. The alternate format described here allows you to write an operand on each line and allows the interspersing of operands and comments in the statement.

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

The rules for using the alternate statement format are:

1. If an operand is followed by a comma and a blank and the column after the end column contains a nonblank character, the operand field may be continued on the next line, starting in the continue column. More than one operand may appear on the same line.
2. Comments may appear after the blank that indicates the end of an operand, up to and including the end column.
3. If the next line starts after the continue column, the information entered on the next line is considered comments and the operand field is considered terminated. Any subsequent continuation lines are considered comments.

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

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

```
Name    Operation  Operand                      Comments

NAME1   OP1        OPERAND1,OPERAND2,               X
                   OPERAND3
                   THIS IS THE NORMAL               X
                   STATEMENT FORMAT
NAME2   OP2        OPERAND1, THIS IS                X
                   THE AL
                   OPERAND2,OPERAND3,               X
                   TERNATE STATEMENT
                   FORMAT
NAME3   OP3        OPERAND1, THIS IS A              X
                   COMB
                   OPERAND2,OPERAND3,               X
                   OPERAND4, OPERAND 5
                   INATION OF BOTH                  X
                   STATEMENT FORMATS
```

Model Statements

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

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

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

Name Field

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

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

Operation Field

The operation field may contain a machine instruction, any assembler instruction listed in Chapter 4 (except END, ICTL, ISEQ, or PRINT), a macro instruction, or a variable symbol. It may also contain an ordinary symbol concatenated with a variable symbol or a variable symbol concatenated with one or more other variable symbols.

Variable symbols may not be used to generate:

- Macro instructions
- Assembler instructions appearing elsewhere than in Chapter 4
- END, ICTL, ISEQ, PRINT, or REPRO instructions

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

The use of COPY instructions is described under *COPY STATEMENTS* following.

Variable symbols in the line following a REPRO instruction will not be placed by their values.

Operand Field

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

Comments Field

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

Symbolic Parameters

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

A symbolic parameter consists of an ampersand followed by from one through seven letters and/or digits, the first of which must be a letter. Elsewhere, two ampersands must be used to represent an ampersand.

You should not use &SYS as the first four characters of a symbolic parameter.

The following are valid symbolic parameters:

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

The following are invalid symbolic parameters:

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

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

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

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

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

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

```
HERE        MOVE       FIELDA,FIELDB
```

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

```
HERE        ST         2,SAVE
            L          2,FIELDB
            ST         2,FIELDA
            L          2,SAVE
```

The following example illustrates another use of the MOVE macro instruction, using operands different from those in the preceding example:

Macro	LABEL	MOVE	IN,OUT
Generated	LABEL	ST	2,SAVE
Generated		L	2,OUT
Generated		ST	2,IN
Generated		L	2,SAVE

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

Concatenating Symbolic Parameters

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

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

Header		MACRO	
Prototype	&NAME	MOVE	&TY,&P,&TO,&FROM
Model	&NAME	ST&TY	2,SAVEAREA
Model		L&TY	2,&P&FROM
Model		ST&TY	2,&P&TO
Model		L&TY	2,SAVEAREA
Trailer		MEND	
Macro	HERE	MOVE	H,FIELD,A,B
Generated	HERE	STH	2,SAVEAREA
Generated		LH	2,FIELDB
Generated		STH	2,FIELDA
Generated		LH	2,SAVEAREA

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

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

If you wish to concatenate a symbolic parameter with a letter, digit, left parenthesis, or period following the symbolic parameter, you must immediately follow the symbolic parameter with a period. A period is optional if the symbolic parameter is to be concatenated with (1) another symbolic parameter or (2) with a special character other than a left parenthesis or another period that follows it, (3) If a symbolic parameter is immediately followed by a period, then the symbolic parameter and the period are replaced by the characters that correspond to the symbolic parameter. A period that immediately follows a symbolic parameter does not appear in the generated statement.

The following macro definitions, macro instruction, and generated statements illustrate these rules:

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

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

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

Comments Statements

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

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

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

```
* THIS STATEMENT WILL BE GENERATED
.* THIS ONE WILL NOT BE GENERATED
```

To get a truly representative sampling of the various language components used effectively in writing macro instructions, you may list all or selected macro instructions from the SYS1.GENLIB or the SYS1.MACLIB by using the IEBPTPCH system utility covered in the *S/360 OS, Utilities Manual,* GC28-6586. This utility program lists

macros with the System/360 mnemonics; however, the concepts are the same for the Communications Controller.

DOS - You may list all or selected macro instructions with the SSERV program. This program is described in *IBM System/360 Disk Operating System: System Control and Service Programs,* GC24-5036. This utility program lists macros with the System/360 mnemonics; however, the concepts are the same for the Communications Controller.

Copy Statements

COPY statements may be used to copy model statements and MEXIT, MNOTE, and conditional assembly instructions into a macro definition, just as they may be used outside macro definitions to copy source statements into an assembler language program. The format of the COPY statement is shown in Figure 45.

The format of this statement is:

Name	*Operation*	*Operand*
Blank	COPY	A symbol

Figure 45. COPY Statement Format

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

DOS - The operand is a symbol that identifies a book to be copied from the private source statement library.

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

Mend -- Macro Definition Trailer

The macro definition trailer statement indicates the end of a macro definition. It can appear only once within a macro definition and must be the last statement in every macro definition. The format of this statement is shown in Figure 46.

Name	*Operation*	*Operand*
A sequence symbol or blank	MEND	Blank

Figure 46. MEND Statement Format

MACRO INSTRUCTIONS

The format of a macro instruction is shown in Figure 47.

Name	*Operation*	*Operand*
Any symbol or blank	Mnemonic operation code	0-100 operands, separated by commas.

Figure 47. Macro Instruction Format

DOS - 0-100 operands, separated by commas.

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

The operation field contains the mnemonic operation code of the macro instruction. The mnemonic operation code must be the same as the mnemonic operation code of a macro definition in the source program or in the macro library.

DOS -- The mnemonic operation code must be the same as the mnemonic operation code of a macro definition in the source program or in the Private Source Statement Library.

The macro definition with the same mnemonic operation code is used by the assembler to process the macro instruction. If a macro definition in the source program and one in the macro library have the same mnemonic operation code, the macro definition in the source program is used.

DOS -- If a macro definition in the source program and one in the Private Source Statement Library have the same mnemonic operation code, the macro definition in the source program is used.

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

Macro Instruction Operands

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

DOS -- Any combination of up to 127 characters may be used as a macro instruction operand if the following rules concerning apostrophes, parentheses, equal signs, ampersands, commas, and blanks are observed.

Paired Apostrophes: An operand may contain one or more quoted strings. A quoted string is any sequence of characters that begins and ends with an apostrophe and contains an even number of apostrophes.

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

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

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

'A''B'C'D'

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

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

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

Paired parentheses are a left parenthesis and a following right parenthesis without any other parentheses intervening. If there is more than one pair, each additional pair is determined by removing any pairs already recognized and reapplying the above rule for paired parentheses. For instance, in the following example the first and fourth, the second and third, and the fifth and sixth parentheses are each paired parentheses:

(A(B)C)D(E)

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

(')')

Equal Signs: An equal sign can occur only between paired apostrophes or paired parentheses. The following examples illustrate these rules:

'C=D'
E(F=G)

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

&&123&&&&

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

(A,B)C','

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

'A B C'

The following examples are valid macro instruction operands:

SYMBOL	A+2
123	(TO(8),FROM)
X'189A'	0 (2,3)
L'NAME	AB&&9
'TEN = 10'	'PARENTHESIS IS)'
'QUOTE IS '''	'COMMA IS ,'

The following examples are invalid macro instruction operands:

W'NAME	(odd number of apostrophes)
5A)B	(number of left parentheses does not equal number of right parentheses)
(15 B)	(blank not placed between paired apostrophes)
'ONE' IS '1'	(blank not placed between paired apostrophes)

Statement Format

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

Omitted Operands

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

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

```
EXAMPLE     &A, &B, &C, &D, &E, &F
EXAMPLE     17, *+4,, AREA, FIELD (6)
```

If the symbolic parameter that corresponds to an omitted operand is used in a model statement, a null character value replaces the symbolic parameter in the generated statement; that is, in effect, the symbolic parameter is removed. For example, the first statement, following is a model statement that contains the symbolic parameter &C. If the operand that corresponds to &C was omitted from the macro instruction, the second statement would be generated from the model statement:

```
L     THERE&C. 25,THIS
L     THERE25, THIS
```

Operand Sublists

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

Sublists provide the programmer with a convenient way to refer to a collection of macro instruction operands as a single operand or as a single operand in a collection of operands.

A sublist consists of one or more operands, separated by commas and enclosed in paired parentheses. The entire sublist, including the parentheses, is considered to be one macro instruction operand.

If a macro instruction is written in the alternate statement format, each operand of the sublist may be written on a separate line; the macro instruction may be written on as many lines as necessary.

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

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

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

```
Header               MACRO
Prototype  &NAME     ADD    &NUM,&REG,&AREA
Model                LA     &REG (1), &NUM (1)
Model                LA     &REG (2), &NUM (2)
Model                LA     &REG (3), &NUM (3)
Model                AR     &REG (1), &REG (2)
Model                AR     &REG (1), &REG (3)
Model                ST     &REG (1), &AREA
Trailer              MEND
Macro                ADD    (A,B,C),(R1,R2,R3),SUM
Generated            LA     R1, A
Generated            LA     R2, B
Generated            LA     R3, C
Generated            AR     R1, R2
Generated            AR     R1, R3
Generated            ST     R1, SUM
```

The operand of the macro instruction that corresponds to symbolic parameter &NUM is a sublist. One of the operands in the sublist is referred to in the operand field of three of the model statements. For example, &NUM (1) refers to the first operand in the sublist corresponding to symbolic parameter &NUM. The first operand of the sublist is A. Therefore, A replaces &NUM (1) to form part of the generated statement.

Note: When referring to an operand in a sublist, the left parenthesis of the sublist notation must immediately follow the last character of the symbolic parameter; for example, &NUM (1). A period should not be placed between the left parenthesis and the last character of the symbolic parameter.

A period may be used between these two characters only when the programmer wants to concatenate the left parenthesis with the characters that the symbolic parameter represents. The following example shows what would be generated if a period appeared between the left parenthesis and the last character of the symbolic parameter in the first model statement of the above example.

```
Prototype  &NAME     ADD    &NUM,&REG,&AREA
Model                L      &REG,&NUM.(1)
Macro                ADD    (A,B,C),R1,SUM
Generated            L      Rl,(A,B,C) (1)
```

The symbolic parameter &NUM is used in the operand field of the model statement. The characters (A,B,C) of the macro instruction correspond to &NUM. Since &NUM is immediately followed by a period, &NUM and the period are replaced by (A,B,C). The period does not appear in the generated statement. The resulting generated statement is an invalid assembler language statement.

Inner Macro Instructions

A macro instruction may be used as a model statement in a macro definition. Macro instructions used as model statements are called inner macro instructions. See the seventh statement in the example following.

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

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

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

The ADD macro instruction of the previous example is used as an inner macro instruction in the example following.

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

The assembler then uses the macro definition that corresponds to the inner macro instruction to generate statements to replace the inner macro instruction. The fifth through the tenth generated statements have been generated for the inner macro instruction. See *Operand Sublists* preceding for a description of the inner macro instruction *ADD.*

1 Header		MACRO	
2 Prototype		COMP	&R1, &R2, &S, &T, &U
3 Model		SR	&R1, &R2
4 Model		LA	&R2, &T
5 Model		CR	&R1, &R2
6 Model		BZL	&U
7 *Inner*		ADD	&S, (5,6,7), &R2
8 Model	&U	AR	&R1, &R2
9 Trailer		MEND	
Macro	K	COMP	3,4 (X,Y,Z), J,K
1 Generated		SR	3,4
2 Generated		LA	4,J
3 Generated		CR	3,4
4 Generated		BZL	K
5 Generated		LA	5,X
6 Generated		LA	6,Y
7 Generated		LA	7,Z
8 Generated		AR	5,6
9 Generated		AR	5,7
10 Generated		ST	5,4
11 Generated	K	AR	3,4

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

Note: An ampersand that is part of a symbolic parameter is not considered in determining whether a macro instruction operand contains an even number of consecutive ampersands.

Levels of Macro Instructions

A macro definition that corresponds to an outer macro instruction may contain any number of inner macro instructions. The outer macro instruction is called a first-level macro instruction. Each of the inner macro instructions is called a second-level macro instruction.

The macro definition that corresponds to a second level macro instruction may contain any number of inner macro instructions. These macro instructions are called third-level macro instructions, etc.

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

CONDITIONAL ASSEMBLY INSTRUCTIONS

The conditional assembly instructions allow the programmer to: (1) define and assign values to SET symbols that can be used to vary parts of generated statements, and (2) vary the sequence of generated statements. Thus, the programmer can use these instructions to generate many different sequences of statements from the same macro definition.

There are 13 conditional assembly instructions, 10 of which are described in this portion of the manual. The other three conditional assembly instructions -- GBLA, GBLB, and GBLC -- are described in *Extended Features of the Macro Language,* following. The instructions are:

LCLA	SETA	AIF	ANOP
LCLB	SETB	AGO	
LCLC	SETC	ACTR	

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

Where the use of an instruction outside macro definitions differs from its use within macro definitions, the difference

is described in the subsequent text.

The LCLA, LCLB, and LCLC instructions may be used to define and assign initial values to SET symbols.

The SETA, SETB, and SETC instructions may be used to assign arithmetic, binary, and character values, respectively, to SET symbols. The SETB instruction is described after the SETA and SETC instructions, because the operand field of the SETB instruction is a combination of the operand fields of the SETA and SETC instructions.

The AIF, AGO, and ANOP instructions may be used with sequence symbols to vary the sequence in which statements are processed by the assembler. The programmer can test attributes assigned by the assembler to symbols or macro instruction operands to determine which statements are to be processed. The ACTR instruction may be used to vary the maximum number of AIF and AGO branches.

Examples illustrating the use of conditional assembly instructions are included throughout this discussion. A chart summarizing the elements that can be used in each instruction is shown in Figure 56.

Set Symbols

SET symbols are one type of variable symbol. The symbolic parameters are another type of variable symbol. SET symbols differ from symbolic parameters in three ways: (1) where they can be used in an assembler language source program, (2) how they are assigned values, and (3) whether or not the values assigned to them can be changed.

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

Symbolic parameters are assigned values when the programmer writes a macro instruction, whereas SET symbols are assigned values when the programmer writes SETA, SETB, and SETC conditional assembly instructions.

Each symbolic parameter is assigned a single value for one use of a macro definition, whereas the values assigned to each SETA, SETB, and SETC symbol can change during one use of a macro definition.

Defining SET Symbols

SET symbols must be defined by the programmer before they are used. When a SET symbol is defined, it is assigned an initial value. SET symbols may be assigned new values by means of the SETA, SETB, and SETC instructions. A SET symbol is defined when it appears in the operand field of an LCLA, LCLB, or LCLC instruction.

Using Variable Symbols

The SETA, SETB, and SETC instructions may be used to change the values assigned respectively. When a SET symbol appears in the name, operation, or operand field of a model statement, the current value of the SET symbol (that is, the last value assigned to it) replaces the SET symbol in the statement.

For example, if &A is a symbolic parameter, and the corresponding characters of the macro instruction are the symbol HERE, then HERE replaces each occurrence of &A in the macro definition. However, if &A is a SET symbol, the value assigned to &A can be changed, and a different value can replace each occurrence of &A in the macro definition.

The same variable symbol may not be used as a symbolic parameter and as a SET symbol in the same macro definition.

The following example illustrates this rule:

```
&NAME          MOVE          &TO, &FROM
```

If the preceding statement is a prototype statement, then &NAME, &TO, and &FROM may not be used as SET symbols in the macro definition.

The same variable symbol may not be used as two different types of SET symbols in the same macro definition. Similarly, the same variable symbol may not be used as two different types of SET symbols outside macro definitions.

For example, if &A is a SETA symbol in a macro definition, it cannot be used as a SETC symbol in that definition. Similarly, if &A is a SETA symbol outside macro definitions, it cannot be used as a SETC symbol outside macro definitions.

The same variable symbol may be used in two or more macro definitions and outside macro definitions. If such is the case, the variable symbol will be considered a different variable symbol each time it is used.

For example, if &A is a variable symbol (either SET symbol or symbolic parameter) in one macro definition, it can be used as a variable symbol (either SET symbol or symbolic parameter) in another definition. Similarly, if &A is a variable symbol (SET symbol or symbolic parameter) in a macro definition, it can be used as a SET symbol outside macro definitions.

All variable symbols may be concatenated with other characters, in the same way that symbolic parameters may be concatenated with other characters. The rules for concatenating symbolic parameters with other characters was previously described under *Symbolic Parameters.*

Variable symbols in macro instructions are replaced by the values assigned to them immediately prior to the start of processing the definition. If a SET symbol is used in the operand field of a macro instruction, and the value assigned to the SET symbol is equivalent to the sublist notation, the operand is not considered a sublist.

Attributes

The assembler assigns attributes to macro instruction operands and to symbols in the program: the length - bytes or bits; the count - number of characters comprising a symbol; and the number - the number of operands in a sublist. These attributes may be referred to only in conditional assembly instructions or expressions.

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

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

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

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

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

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

Attribute	*Notation*
Type	T'
Length	L'
Count	K'
Number	N'

You may refer to an attribute in the following way:

1. In a statement that is outside macro definitions, you may write the notation for the attribute immediately followed by a symbol. (For example, T'NAME refers to the type attribute of the symbol NAME.)
2. In a statement that is in a macro definition, you may write the notation for the attribute immediately followed by a symbolic parameter. (For example, L'&NAME refers to the length attribute of the characters in the macro instruction that correspond to symbolic parameter &NAME; L'&NAME (2) refers to the length attribute of the second operand in the sublist that corresponds to symbolic parameter &NAME.)

Type Attribute (T')

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

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

A	A-type address constant, implied length, aligned
B	Binary constant.
C	Character constant
F	Full-word fixed-point constant, implied length, aligned
G	Fixed-point constant, explicit length
H	Half-word fixed-point constant, implied length, aligned
R	A-, V-, R- or Y-type address constant, explicit length
V	V-type address constant, implied length, aligned
X	Hexadecimal constant
Y	Y-type or R-type address constant, implied length, aligned

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

I	Machine instruction
J	Control section name
M	Macro instruction
T	External symbol
W	CW assembler instruction

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

N	Self-defining term
O	Omitted operand

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

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

U   Underfined

You may refer to a type attribute in the operand field of a SETC instruction or to a type attribute in character relations in the operand fields of SETB or AIF instructions.

Length Attribute (L')

The length attribute of a macro instructions operands and symbols is a numeric value.

The length attribute of a symbol (or of a macro instruction operand that is a symbol) is as described in Chapter 2 of this publication. Reference to the length attribute of a variable symbol is illegal except for symbolic parameters in SETA, SETB and AIF statements.

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

You may refer to the length attributes in the operand field of a SETA instruction or to the length attributes in arithmetic relations in the operand fields of SETB or AIF instructions.

Count Attribute (K')

You may refer to the count attribute of macro instruction operands only.

The value of the count attribute is equal to the number of characters in the macro instruction operand. It includes all characters in the operand, excluding the delimiting commas. If the operand is a sublist, that operand includes the beginning and ending parenthesis and the commas within the sublist. The count attribute of an omitted operand is zero. These rules are illustrated by the following examples:

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

If a macro instruction operand contains variable symbols, the character that replace the variable symbols, rather than the variable symbols, are used to determine the count attribute.

You may refer to the count attribute in the operand field of a SETA instruction or to the count attribute in arithmetic relations in the operand fields of SETB and AIF instructions that are part of a macro definition.

Number Attribute (N')

You may refer to the number attribute of macro instruction operands only.

The number attribute is a value equal to the number of operands in an operand sublist. The number of operands in an operand sublist is equal to one plus the number of commas that indicate the end of an operand in the sublist.

The following examples illustrate this rule:

(A, B, C, D, E)	5 operands
(A, ,C, D, E)	5 operands
(A, B, C, D)	4 operands
(, B, C, D, E)	5 operands
(A, B, C, D,)	5 operands
(A, B, C, D, ,)	6 operands

If the macro instruction operand is not a sublist, the number attribute is one. If the macro instruction operand is omitted, the number attribute is zero.

You may refer to the number attribute in the operand field of a SETA instruction or to the number attribute in arithmetic relations in the operand fields of SETB and AIF instructions that are part of a macro definition.

Sequence Symbols

The name field of a statement may contain a sequence symbol. Sequence symbols provide you with the ability to vary the sequence in which statements are processed by the assembler.

A sequence symbol is used in the operand field of an AIF or AGO statement to refer to the statement named by the sequence symbol.

A sequence symbol is considered to be local to a macro definition.

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

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

The following are valid sequence symbols:

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

The following are invalid sequence symbols:

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

.IN AREA (contains a special character (blank) other than initial period)

If a sequence symbol appears in the name field of a macro instruction, and the corresponding prototype statement contains a symbolic parameter in the name field, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro definition.

The following examples illustrates this rule:

```
   Name       Operation    Operand
              MACRO
1  &NAME      MOVE         &TO, &FROM
2  &NAME      ST           2, SAVEAREA
              L            2, &FROM
              ST           2, &TO
              L            2, SAVEAREA
              MEND
3  .SYM       MOVE         FIELDA, FIELDB
4             ST           2, SAVEAREA
              L            2, FIELDB
              ST           2, FIELDA
              L            2, SAVEAREA
```

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

LCLA, LCLB, LCLC - Define Local Set Symbols

The format of these instructions is shown in Figure 48.

Name	*Operation*	*Operand*
Blank	LCLA, LCLB, or LCLC	One or more variable symbols that are to be used as SET symbols, separated by commas.

Figure 48. Local SET Symbol Statement Format

The LCLA, LCLB, and LCLC instructions are used to define and assign initial values to SETA, SETB, and SETC symbols, respectively. The SETA, SETB, and SETC symbols are assigned the initial values of *0*, 0, and null character value, respectively.

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

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

SETA - Set Arithmetic

The SETA instruction may be used to assign an arithmetic value to a SETA symbol. The format of this instruction is shown in Figure 49.

Name	*Operation*	*Operand*
A SETA symbol	SETA	An arithmetic expression

Figure 49. SETA Statement Format

The expression in the operand field is evaluated as a signed 32-bit arithmetic value that is assigned to the SETA symbol in the name field. The minimum and maximum allowable values of the expression are -2^{31} and $+2^{31}-1$, respectively.

The expression may consist of one term or an arithmetic combination of terms. The terms that may be used alone or in combination with each other are self-defining terms, variable symbols, and the length, count, and number attributes. Self-defining terms are described in *Chapter 2, Basic Assembler Language Concepts,* of this publication.

Note: A SETC variable symbol may appear in a SETA expression only if the value of the SETC variable is one of eight decimal digits. The decimal digits will be converted to a positive arithmetic value.

The arithmetic operators that may be used to combine the terms of an expression are + (addition), - (subtraction), * (multiplication), and / (division).

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

The following are valid operand fields of SETA instructions:

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

The following are invalid operand fields of SETA instructions:

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

Evaluation of Arithmetic Expressions

The procedure used to evaluate the arithmetic expression in the operand field of a SETA instruction is the same as that used to evaluate arithmetic expressions in assembler language statements. The only difference between the two types of arithmetic expressions is the terms that are allowed in each expression.

The following evaluation procedure is used:

1. Each term is given its numerical value.
2. The arithmetic operations are performed, moving from left to right, with multiplication and/or division being performed before addition and subtraction.
3. The computed result is the value assigned to the SETA symbol in the name field.

The arithmetic expression in the operand field of a SETA instruction may contain one or more sequences of arithmetically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence. Only five levels of parentheses are allowed, and an expression may not consist of more than 16 terms. Parentheses required for sublist notation, substring notation, and subscript notation count toward this limit. A counter is maintained for each SETA statement and increased by one for each occurrence of a variable symbol, as well as for the operation entry. The maximum value this counter may attain is 35.

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

```
(L'&HERE+32)*39
&AREA+X' 2D'/(&EXIT–K'&ENTRY+1)
&BETA*10*(&N/25/(&EXIT–K'&ENTRY+1))
```

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

Using SETA Symbols

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

The following example illustrates this rule:

```
            MACRO
   &NAME    MOVE      &TO, &FROM
            LCLA      &A, &B, &C, &D
1  &A       SETA      10
2  &B       SETA      12
3  &C       SETA      &A–&B
4  &D       SETA      &A+&C
   &NAME    ST        2,SAVEAREA
5           L         2,&FROM&C
6           ST        2,&TO&D
            L         2,SAVEAREA
            MEND

   HERE     MOVE      FIELDA,FIELDB
   HERE     ST        2,SAVEAREA
            L         2,FIELDB2
            ST        2,FIELDA8
            L         2,SAVEAREA
```

Statements 1 and 2 assign to the SETA symbols &A and &B the arithmetic values +10 and +12, respectively. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2 is converted to the unsigned integer 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the unsigned integer 8.

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

```
            MACRO
   &NAME    MOVE      &TO, &FROM
            LCLA      &A
1  &A       SETA      5
   &NAME    ST        2,SAVEAREA
2           L         2,&FROM&A
3  &A       SETA      8
4           ST        2,&TO&A
            L         2,SAVEAREA
            MEND

   HERE     MOVE      FIELDA,FIELDB
   HERE     ST        2,SAVEAREA
            L         2,FIELDB5
            ST        2,FIELDA8
            L         2,SAVEAREA
```

Statement 1 assigns the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the unsigned integer 5. Statement 3 assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the unsigned integer 8, instead of 5.

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

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

The following macro definition may be used to add the last operand in an operand sublist to the first operand in an operand sublist and store the result at the first operand. A sample macro instruction and generated statements follow the macro definition.

```
             MACRO
1            ADDX      &NUMBER,&REG
             LCLA      &LAST
2  &LAST     SETA      N'& NUMBER
             L         &REG (1), &NUMBER (1)
3            L         &REG (2), &NUMBER
                          (&LAST)
             AR        &REG (1), &REG (2)
             ST        &REG (1), &NUMBER (1)
             MEND
4            ADDX      (A, B, C, D, E), (3,4)
             L         3, A
             L         4, E
             AR        3, 4
             ST        3, A
```

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

SETC -- Set Character

The SETC instruction is used to assign a character value to a SETC symbol. The format of this instruction is shown in Figure 50.

Name	*Operation*	*Operand*
A SETC symbol	SETC	One operand, of the form described in the following text.

Figure 50. SETC Statement Format

The operand field may consist of the type attribute, a character expression, a substring notation, or a concatenation of substring notations and character expressions. A SETA symbol may appear in the operand of a SETC statement. The result is the character representation of the decimal value, unsigned, with leading zeros removed. If the value is zero, one decimal zero is used.

Type Attribute

The character value assigned to a SETC symbol may be a type attribute. If the type attribute is used, it must appear alone in the operand field. The following example assigns to the SETC symbol &TYPE the letter that is the type attribute of the macro instruction operand corresponding to the symbolic parameter &ABC.

```
&TYPE        SETC        T'&ABC
```

Character Expression

A character expression consists of any combination of up to 255 characters enclosed in apostrophes.

DOS - A character expression consists of any combination of characters enclosed in apostrophes (127 characters maximum).

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

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

```
&ALPHA       SETC        'AB%4'
```

More than one character expression may be concatenated into a single character expression by placing a period between the terminating apostrophe of one character expression and the opening apostrophe of the next character expression. For example, either of the following statements may be used to assign the character value ABCDEF to the SETC symbol &BETA:

```
&BETA        SETC        'ABCDEF'
&BETA        SETC        'ABC'.'DEF'
```

Two apostrophes must be used to represent an apostrophe that is part of a character expression.

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

```
&LENGTH      SETC        'L' 'SYMBOL'
```

Variable symbols may be concatenated with other characters in the operand field of a SETC instruction, according to the general rules for concatenating symbolic parameters with other characters.

If &ALPHA has been assigned the character value AB%4, the following statement may be used to assign the character value AB%4RST to the variable symbol &GAMMA:

```
&GAMMA       SETC        'A&ALPHA.RST'
```

Two ampersands must be used to represent an ampersand that is not part of a variable symbol. Both ampersands become part of the character value assigned to the SETC symbol. They are not replaced by a single ampersand.

The following statement assigns the character value HALF&& to the SETC symbol &AND:

```
&AND          SETC      'HALF&&'
```

Substring Notation

The character value assigned to a SETC symbol may be a substring character value. Substring character values permit the programmer to assign part of a character value to a SETC symbol.

If you want to assign part of a character value to a SETC symbol, you must indicate to the assembler in the operand field of a SETC instruction: (1) the character value itself, and (2) the part of the character value he wants to assign to the SETC symbol. The combination of (1) and (2) in the operand field of a SETC instruction is called a substring notation. The character value that is assigned to the SETC symbol in the name field is called a substring character value.

Substring notation consists of a character expression, immediately followed by two arithmetic expressions that are separated from each other by a comma and are enclosed in parentheses. The two arithmetic expressions may be any expression that is allowed in the operand field of a SETA instruction.

The first expression indicates the first character in the character expression that is to be assigned to the SETC symbol in the name field. The second expression indicates the number of consecutive characters in the character expression (starting with the character indicated by the first expression) that are to be assigned to the SETC symbol. If a substring asks for more characters than are in the character string, only the characters in the string will be assigned.

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

DOS - The maximum size character expression the substring character value can be chosen from is 127 characters.

The following are valid substring notations:

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

The following are invalid substring notations:

'&BETA' (4, 6)
(blanks between character value and arithmetic expressions)
'L''SYMBOL' (142-EXYZ)
(only one arithmetic expression)
'AB&4&ALPHA' (8 &FIELD*2)
(arithmetic expressions not separated by a comma)
'BETA' 4, 6
(arithmetic expressions not enclosed in parentheses)

Using SETC Symbols

The character value assigned to a SETC symbol is substituted for the SETC symbol when it is used in the name, operation, or operand field of a statement.

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

```
             MACRO
   &NAME     MOVE      &TO, &FROM
             LCLC      &PREFIX
1  &PREFIX   SETC      'FIELD'
   &NAME     ST        2,SAVEAREA
2            L         2,&PREFIX&FROM
3            ST        2,&PREFIX&TO
             L         2,SAVEAREA
             MEND

   HERE      MOVE      A, B

   HERE      ST        2,SAVEAREA
             L         2,FIELDB
             ST        2,FIELDA
             L         2,SAVEAREA
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX. In statements 2 and 3, &PREFIX is replaced by FIELD.

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

```
             MACRO
   &NAME     MOVE      &TO, &FROM
             LCLC      &PREFIX
1  &PREFIX   SETC      'FIELD'
   &NAME     ST        2,SAVEAREA
2            L         2,&PREFIX&FROM
3  &PREFIX   SETC      'AREA'
4            ST        2,&PREFIX&TO
             L         2,SAVEAREA
             MEND

   HERE      MOVE      A, B

   HERE      ST        2,SAVEAREA
             L         2,FIELDB
             ST        2,AREA
             L         2,SAVEAREA
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX; therefore, &PREFIX is replaced

by FIELD in statement 2. Statement 3 assigns the character value AREA to &PREFIX; therefore, &PREFIX is replaced by AREA, instead of FIELD, in statement 4.

The following example illustrates the use of a substring notation as the operand field of a SETC instruction:

```
             MACRO
  &NAME      MOVE     &TO, &FROM
             LCLC     &PREFIX
1 &PREFIX    SETC     '&TO' (1, 5)
  &NAME      ST       2,SAVAREA
2            L        2,&PREFIX&FROM
             ST       2,&TO
             L        2,SAVEAREA
             MEND
  HERE       MOVE     FIELDA, B
  HERE       ST       2,SAVEAREA
             L        2,FIELDB
             ST       2,FIELDA
             L        2,SAVEAREA
```

Statement 1 assigns the substring character value FIELD (the first five characters corresponding to symbolic parameter &TO) to the SETC symbol &PREFIX; therefore, FIELD replaces &PREFIX in statement 2.

SETB -- Set Binary

The SETB instruction may be used to assign the binary value 0 or 1 to a SETB symbol. The format of this instruction is shown in Figure 51.

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

Figure 51. SETB Statement Format

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

A logical expression consists of one term or a logical combination of terms. The terms that may be used alone or in combination with each other are arithmetic relations, character relations, and SETB symbols. The logical operators used to combine the terms of an expression are AND, OR, and NOT.

An expression may not contain two terms in succession. A logical expression may contain two operators in sucession only if the first operator is either AND or OR and the second operator is NOT. A logical expression may begin with the operator NOT. It may not begin with the operators AND or OR.

An arithmetic relation consists of two arithmetic expressions, connected by a relational operator. A character relation consists of two character values connected by a relational operator. The relational operators are EQ (equal), NE (not equal), LT (less than), GT (greater than), and GE (greater than or equal).

Any expression that may be used in the operand field of a SETA instruction may be used as an arithmetic expression in the operand field of a SETB instruction. Anthing that may be used in the operand field of a SETC instruction may be used as a character value in the operand field of a SETB instruction. This includes substring and type attribute notations. The maximum size of the character values that can be compared is 255 characters.

DOS - The maximum size of the character values that can be compared is 127 characters.

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

The following are valid operand fields of SETB instructions:

(&AREA+2 GT 29)
('AB%4' EQ '&ALPHA')
(T'&ABC NE T'&XYZ)
(T'&P12 EQ ' F')
(&AREA+2 GT 29 OR &B)
(NOT &B AND &AREA+X'2D' GT 29)
('&C' EQ' MD')
(0)

The following are invalid operand fields of SETB instructions:

&B (not enclosed in parentheses)
(T'&P12 EQ 'F' &B)
(two terms in succession)
('AB%4' EQ 'ALPHA' NOT &B)
(The NOT operator must be preceded by AND or OR)
(AND T'&P12 EQ 'F')
(expression begins with AND)

Evaluation of Logical Expressions

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

1. Each term (that is, arithmetic relation, character relation, or SETB symbol) is evaluated and given its logical value (true or false).
2. The logical operations are performed by moving from

left to right, with NOTs being performed before ANDs, and ANDs being performed before ORs.

3. The computed result is the value assigned to the SETB symbol in the name field.

The logical expression in the operand field of a SETB instruction may contain one or more sequences of logically combined terms that are enclosed in parenthese. A sequence of parenthesized terms may appear within another parenthesized sequence.

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

```
(NOT (&B AND &AREA+X'2D' GT 29))
(&B AND (T'&P12 EQ 'F' OR &B))
```

The parenthesized portion or portions of a logical expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first. Five levels of parentheses are permissible.

Using SETB Symbols

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

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

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

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

```
             MACRO
  &NAME      MOVE      &TO, &FROM
             LCLA      &A1
             LCLB      &B1, &B2
             LCLC      &C1
1 &B1        SETB      (L'&TO EQ 4)
2 &B2        SETB      (K'&TO EQ 0)
3 &A1        SETA      &B1
4 &C1        SETC      '&B2'
             ST        2,SAVEAREA
             L         2,&FROM&A1
             ST        2,&TO&C1
             L         2,SAVEAREA
             MEND

  HERE       MOVE      FIELDA, FIELDB

  HERE       ST        2,SAVEAREA
             L         2,FIELDB1
             ST        2,FIELDA0
             L         2,SAVEAREA
```

Because the operand field of statement 1 is true, &B1 is assigned the binary value 1; therefore, the arithmetic value +1 is substituted for &B1 in statement 3. Because the operand field of statement 2 is false, &B2 is assigned the binary value 0; therefore, the character value 0 is substituted for &B2 in statement 4.

Concatenating Substring Notations and Character Expressions: Substring notations may be concatenated with character expressions in the operand field of a SETC instruction. If a substring notation follows a character expression, the two may be concatenated by placing a period between the terminating apostrophe of the character expression and the opening apostrophe of the substring notation.

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

```
&GAMMA    SETC      'ALPHA' . '&BETA' (2,3)
```

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

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

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

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

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

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

AIF -- Conditional Branch

The AIF instruction is used to conditionally alter the sequence in which source program statements or macro definition statements are processed by the assembler. The assembler assigns a maximum count of 4096 AIF and AGO branches that may be executed in the source program or in a macro definition. When a macro definition calls an inner

macro definition, the current value of the count is saved and a new count of 4096 is set up for the inner macro definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored.

DOS - The assembler assigns a maximum count of 150 AIF and AGO branches that may be executed in the source program or in a macro definition. When a macro definition calls an inner macro definition, the current value of the count is saved and a new count of 150 is set up for the inner macro definition.

The format of this instruction is shown in Figure 52:

Name	*Operation*	*Operand*
A sequence symbol or blank	AIF	A logical expression enclosed in parentheses, immediately followed by a sequence symbol.

Figure 52. AIF Statement Format

Any logical expression that may be used in the operand field of a SETB instruction may be used in the operand field of an AIF instruction. The sequence symbol in the operand field must immediately follow the closing parenthesis of the logical expression.

The logical expression in the operand field is evaluated to determine if it is true or false. If the expression is true, the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is false, the next sequential statement is processed by the assembler.

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

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

The following are valid operand fields of AIF instructions:

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

The following are invalid operand fields of AIF instructions:

(T'&ABC NE T'&XYZ) (no sequence symbol)
X4F2 (no logical expression)
(T'&ABC NE T'&XYZ) .X4F2
(blanks between logical expression and sequence symbol)

The following macro definition may be used to generate the statements needed to move a fullword fixed-point number from one storage area to another. The statements will be generated only if the type attribute of both storage areas is the letter F.

```
           MACRO
   &N      MOVE         &T, &F
1          AIF          (T'&T NE T'&F). END
2          AIF          (T'&T NE 'F') . END
3  &N      ST           2,SAVEAREA
           L            2,&F
           ST           2,&T
           L            2,SAVEAREA
4  .END    MEND
```

The logical expression in the operand field of statement 1 has the value *true* if the type attributes of the two macro instruction operands are not equal. If the type attributes are equal, the expression has the logical value *false*.

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

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

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

AGO -- Unconditional Branch

The AGO instruction is used to unconditionally alter the sequence in which source program or macro definition statements are processed by the assembler. The assembler assigns a maximum count of 4096 AIF and AGO branches that may be executed in the source program or in a macro definition.

When a macro definition calls an inner macro definition, the current value of the count is saved and a new count of 4096 is set up for the inner macro definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored.

DOS -- The assembler assigns a maximum count of 150 AIF and AGO branches that may be executed in the source program or in a macro definition. When a macro definition calls an inner macro definition, the current value of the count is saved and a new count of 150 is set up for the

inner macro instruction. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored.

The format of this instruction is shown in Figure 53:

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

Figure 53. AGO Statement Format

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

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

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

The following example illustrates the use of the AGO instruction:

```
           MACRO
  &NAME    MOVE      &T, &F
1          AIF       (T'&T &Q 'F') . FIRST
2          AGO       .END
3 .FIRST   AIF       (T'&T NE T'&F). END
  &NAME    ST        2,SAVEAREA
           L         2, &F
           ST        2, &T
           L         2, SAVEAREA
4 .END     MEND
```

Statement 1 is used to determine if the type attribute of the first macro instruction operand is the letter F. If th type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler.

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

ACTR -- Conditional Assembly Loop Counter

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

DOS - Different from the standard count of 150

The format of this instruction is as follows in Figure 54:

Name	*Operation*	*Operand*
Blank	ACTR	Any valid SETA expression

Figure 54. ACTR Statement Format

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

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

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

ANOP -- Assembly No Operation

The ANOP instruction facilitates conditional and unconditional branching to statements named by symbols or variable symbols.

The format of this instruction is shown in Figure 55:

Name	*Operation*	*Operand*
A sequence symbol	ANOP	Blank

Figure 55. ANOP Statement Format

If you want to use an AIF or AGO instruction to branch to another statement, you must place a sequence symbol in the name field of the statement to which you want to branch. However, if you have already entered a symbol or variable symbol in the name field of that statement, you cannot place a sequence symbol in the name field. Instead, you must place an ANOP instruction before the statement and then branch to the ANOP instruction. This has the same effect as branching to the statement immediately after the ANOP instruction.

The following example illustrates the use of the ANOP instruction:

```
         MACRO
  &NAME  MOVE      &T, &F
         LCLC      &TYPE
1        AIF       (T'&T EQ 'F'). FTYPE
2 &TYPE  SETC      'H'
3 .FTYPE ANOP
4. &NAME ST&TYPE   2, SAVEAREA
         L&TYPE    2, &F
         ST&TYPE   2, &T
         L&TYPE    2, SAVEAREA
         MEND
```

Statement 1 is used to determine if the type attribute of the first macro instruction operand is the letter F. If type attribute is not the letter F, statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, statement 4 should be processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (statement 3) must be placed before statement 4.

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

Conditional Assembly Elements

The following chart (Figure 56) summarizes the elements that can be used in each conditional assembly instruction. Each row in this chart indicates which elements can be used in a single conditional assembly instruction. Each column is used to indicate the conditional assembly instructions in which a particular element can be used.

The intersection of a column and a row indicates whether an element can be used in an instruction, and if so, in what fields of the instruction the element can be used. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions.

		Variable Symbols *SET Symbols*					*Attributes*		
	S.P.	SETA	SETB	SETC	T'	L'	K'	N'	S.S.
SETA	O	N,O	O	O	O[3]		O	O	
SETB	O	O	N,O	O	O[1]	O[2]	O[2]	O[2]	
SETC	O	O		O	N,O	O			
AIF	O	O	O	O	O[1]	O[2]	O[2]	O[2]	N,O
AGO									N,O
ANOP									N
ACTR	O	O	O	O[3]	O		O	O	

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

Figure 56. Elements of Conditional Assembly Instructions

EXTENDED FEATURES OF THE MACRO LANGUAGE

The extended features of the macro language allow you to:

1. Terminate processing of a macro definition
2. Generate error messages
3. Define global SET symbols
4. Define subscripted SET symbols
5. Use system variable symbols
6. Prepare keyword and mixed-mode macro definitions and write keyword and mixed-mode macro definitions.

MEXIT -- Macro Definition Exit

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

Name	*Operation*	*Operand*
A sequence symbol or blank	MEXIT	Blank

Figure 57. MEXIT Statement Format

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

If the assembler processes a MEXIT instruction that is in a macro definition corresponding to an outer macro instruction, the next statement processed by the assembler is the next statement outside macro definitions.

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

MEXIT should not be confused with MEND. MEND indicates the end of a macro definition. MEND must be the last statement of all macro definitions, including those that contain one or more MEXIT instructions.

The following example illustrates the use of the MEXIT instruction:

```
        MACRO
  &NAME MOVE    &T,&F
1       AIF     (T'&T EQ 'F').OK
2       MEXIT
3 .OK   ANOP
  &NAME ST      2,SAVEAREA
        L       2,&F
        ST      2,&T
        L       2,SAVEAREA
        MEND
```

Statement 1 is used to determine if the type attribute of the first macro instruction operand is the letter F. If the type attribute is the letter F, the assembler processes the remainder of the macro definition, starting with statement 3. If the type attribute is not the letter F, the next statement processed by the assembler is statement 2. Statement 2 indicates to the assembler that it is to terminate processing of the macro definition.

MNOTE -- Request for Error Message

The MNOTE instruction may be used to request the assembler to generate an error message. The format of this instruction is shown in Figure 58.

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

Figure 58. MNOTE Statement Format

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

```
MNO    MNOTE   severity code,'message'
MNP    MNOTE      ,'message'
MNQ    MNOTE   'message'
```

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

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

DOS - In DOS the severity code is for your information only. It is not used by the DOS assembler or control program.

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

Two apostrophes must be used to represent an apostrophe enclosed in apostrophes in the operand field of an MNOTE instruction. One apostrophe is listed for each pair of apostrophes in the operand field. If any variable symbols are used in the operand field of an MNOTE instruction, they are replaced by the values assigned to them. Two ampersands must be used to represent an ampersand that is not part of a variable symbol in the operand field of an MNOTE statement. One ampersand is listed for each pair of ampersands in the operand field.

The following example illustrates the use of the MNOTE instruction:

```
        MACRO
  &NAME MOVE    &T,&F
        MNOTE   *,'MOVE MACRO GEN'
1       AIF     (T'&T NE T'&F).M1
2       AIF     (T'&T NE 'F').M2
3 &NAME ST      2,SAVEAREA
        L       2,&F
        ST      2,&T
        L       2,SAVEAREA
        MEXIT
4 .M1   MNOTE   'TYPE NOT SAME'
        MEXIT
5 .M2   MNOTE   'TYPE NOT F'
        MEND
```

Statement 1 is used to determine if the type attribute of both macro instruction operands are the same. If they are, statement 2 is the next statement processed by the assembler. If they are not, statement 4 is the next statement processed by the assembler. Statement 4 causes an error message to be printed in the source program listing indicating that the type attributes are not the same.

Statement 2 is used to determine if the type attribute of the first macro instruction operand is the letter F. If

the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the attribute is not the letter F, statement 5 is the next statement processed by the assembler. Statement 5 causes an error message (indicating that the type attribute is not F) to be printed in the source program listing.

Global and Local Variable Symbols

The following are local variable symbols:

1. Symbolic parameters
2. Local SET symbols
3. System variable symbols

Global SET symbols are the only global variable symbols.

The GBLA, GBLB, and GBLC instructions define global SET symbols, just as the LCLA, LCLB, and LCLC instructions define the SET symbols described above. Hereafter, SET symbols defined by LCLA, LCLB, and LCLC instructions are called local SET symbols.

Global SET symbols communicate values between statements in one or more macro definitions and statements outside macro definitions. However, local SET symbols communicate values between statements in the same macro definition or between statements outside macro definitions.

If a local SET symbol is defined in two or more macro definitions or in a macro definition and outside macro definitions, the SET symbol is considered to be a different SET symbol in each case. However, a global SET symbol is the same SET symbol each place it is defined.

A SET symbol must be defined as a global SET symbol in each macro definition in which it is to be used as such. A SET symbol must be defined as a global SET symbol outside macro definitions, if it is to be used in this manner.

If the same SET symbol is defined as a global SET symbol in one or more places and as a local SET symbol elsewhere, it is considered the same symbol wherever it is defined as a global SET symbol, and as a different symbol wherever it is defined as a local SET symbol.

Defining Local and Global SET Symbols

Local SET symbols are defined when they appear in the operand field of an LCLA, LCLB, or LCLC instruction (described under *Defining SET Symbols*, previously discussed).

Global SET symbols are defined when they appear in the operand field of a GBLA, GBLB, or GBLC instruction. The instruction format is shown in Figure 59.

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

Figure 59. Global SET Symbol Statement Format

The GBLA, GBLB, and GBLC instructions define global SETA, SETB, and SETC symbols, respectively, and assign the same initial values as the corresponding types of local SET symbols. However, a global SET symbol is assigned an initial value by only the first GBLA, GBLB, or GBLC instruction processed in which the symbol appears. Subsequent GBLA, GBLB, or GBLC instructions processed by the assembler do not affect the value assigned to the SET symbol.

You should not define any global SET symbols whose first four characters are &SYS.

If a GBLA, GBLB, or GBLC instruction is part of a macro definition, it must immediately follow the prototype statement, or another GBLA, GBLB, or GBLC instruction. GBLA, GBLB, and GBLC instructions outside macro definitions must appear (1) after all macro definitions in the source program, (2) before all conditional assembly instructions and PUNCH and REPRO statements outside macro definitions, and (3) before the first control section of the program.

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

Using Local and Global SET Symbols

The following examples illustrate the use of global and local SET symbols. Each example consists of two parts. The first part is an assembler language source program. The second part shows the statements that would be generated by the assembler after it processed the statements in the source program.

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

```
         MACRO
  &NAME  LOADA
1        LCLA     &A
2 &NAME  LR       5,&A
3 &A     SETA     &A+1
         MEND
4        LCLA     &A
  FIRST  LOADA
5        LR       5,&A
         LOADA
6        LR       5,&A
         END      FIRST
  FIRST  LR       5,0
         LR       5,0
         LR       5,0
         LR       5,0
         END      FIRST
```

&A is defined as a local SETA symbol in a macro definition (statement 1) and outside macro definitions (statement 4). &A is used twice within the macro definition (statements 2 and 3) and twice outside macro definitions (statements 5 and 6).

Since &A is a local SETA symbol in the macro definition and outside macro definitions, it is one SETA symbol in the macro definition, and another SETA symbol outside macro definitions. Therefore, statement 3 (which is in the macro definition) does not affect the value used for &A in statements 5 and 6 (which are outside macro definitions). Moreover, the use of LOADA between statements 5 and 6 alters &A from its previous value as a local symbol within that macro definition since the first act of the macro definition is to set &A to zero.

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

```
         MACRO
  &NAME  LOADA
1        GBLA     &A
2 &NAME  LR       5,&A
3 &A     SETA     &A+1
         MEND
4        GBLA     &A
  FIRST  LOADA
5        LR       5,&A
         LOADA
6        LR       5,&A
         END      FIRST
  FIRST  LR       5,0
         LR       5,1
         LR       5,1
         LR       5,2
         END      FIRST
```

&A is defined as a global SETA symbol in a macro definition (statement 1) and outside macro definitions (statement 4). &A is used twice within the macro definition (statements 2 and 3) and twice outside macro definitions (statements 5 and 6).

Since &A is a global SETA symbol in the macro definition and outside macro definitions, it is the same SETA symbol in both cases. Therefore, statement 3 (which is in the macro definition) affects the value used for &A in statements 5 and 6 (which are outside macro definitions).

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

&A is defined as a local SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statements 2, 3, 5, and 6).

Since &A is a local SETA symbol in each macro definition, &A may have a different value in one macro definition, from that in another. Therefore, statement 3 (which is in one macro definition) does not affect the value used for &A in statement 5 (which is in the other macro definition). Similarly, statement 6 does not affect the value used for &A in statement 2.

```
         MACRO
  &NAME  LOADA
1        LCLA     &A
2 &NAME  LR       5,&A
3 &A     SETA     &A+1
         MEND
         MACRO
         LOADB
4        LCLA     &A
5        LR       5,&A
6 &A     SETA     &A+1
         MEND
  FIRST  LOADA
         LOADB
         LOADA
         LOADB
         END      FIRST
  FIRST  LR       5,0
         LR       5,0
         LR       5,0
         LR       5,0
         END      FIRST
```

Example 4: This example illustrates how a SET symbol can be used to communicate values between statements that are part of two different macro definitions.

```
            MACRO
  &NAME     LOADA
1           GBLA      &A
2 &NAME     LR        5,&A
3 &A        SETA      &A+1
            MEND
            MACRO
            LOADB
4           GBLA      &A
5           LR        5,&A
6 &A        SETA      &A+1
            MEND
  FIRST     LOADA
            LOADB
            LOADA
            LOADB
            END       FIRST
  FIRST     LR        5,0
            LR        5,1
            LR        5,2
            LR        5,3
            END       FIRST
```

&A is defined as a global SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statements 2, 3, 5, and 6).

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

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

```
            MACRO
  &NAME     LOADA
1           GBLA      &A
2 &NAME     LR        5,&A
3 &A        SETA      &A+1
            MEND
            MACRO
            LOADB
4           GBLA      &A
5           LR        5,&A
6 &A        SETA      &A+1
            MEND
7           LCLA      &A
  FIRST     LOADA
            LOADB
8           LR        5,&A
            LOADA
            LOADB
9           LR        5,&A
            END       FIRST
  FIRST     LR        5,0
            LR        5,1
            LR        5,0
            LR        5,2
            LR        5,3
            LR        5,0
            END       FIRST
```

&A is defined as a global SETA symbol in two different macro definitions (statements 1 and 4), but it is defined as a local SETA symbol outside macro definitions (statement 7). &A is used twice within each macro definition and twice outside macro definitions (statements 2, 3, 5, 6, 8 and 9).

Since &A is a global SETA symbol in each macro definition, it is the same SETA symbol in each macro definition. However, since &A is a local SETA symbol outside macro definitions, it is a different SETA symbol outside macro definitions.

Therefore, statement 3 (which is in one macro definition) affects the value used for &A in statement 5 (which is in the other macro definition), but it does not affect the value used for &A in statements 8 and 9 (which are outside macro definitions). Similarly, statement 6 affects the value used for &A in statement 2, but it does not affect the value used for &A in statements 8 and 9.

Subscripted SET Symbols

Both global and local SET symbols may be defined as subscripted SET symbols. The local SET symbols defined previously were all nonsubscripted SET symbols.

Subscripted SET symbols provide the programmer with a convenient way to use one SET symbol plus a subscript to refer to many arithmetic, binary, or character values.

A subscripted SET symbol consists of a SET symbol immediately followed by a subscript that is enclosed in parentheses. The subscript may be any arithmetic expression that is allowed in the operand field of a SETA statement. The subscript may not be 0 or negative.

The following are valid subscripted SET symbols:

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

The following are invalid subscripted SET symbols:

&X4F2	(no subscript)
(25)	(no SET symbol)
&X4F2 (25)	(subscript does not immediately follow SET symbol)

Defining Subscripted SET Symbols: To use a subscripted SET symbol, you must write in a GBLA, GBLB, GBLC, LCLA, LCLB, or LCLC instruction, a SET symbol immediately followed by a decimal integer enclosed in parentheses. The decimal integer, called a dimension, indicates the number of SET variables associated with the SET symbol. Every variable associated with a SET symbol is assigned an initial value that is the same as the initial value assigned to the corresponding type of nonsubscripted SET symbol.

If a subscripted SET symbol is defined as global, the same dimension must be used with the SET symbol each time it is defined as global.

The maximum dimension that can be used with a SETA, SETB, or SETC symbol is 2500.

DOS - The maximum dimension that can be used with a SETA, SETB, or SETC symbol is 255.

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

The following statements define the global SET symbols &SBOX, &WBOX, and &PSW, and the local SET symbol &TSW. &SBOX has 50 arithmetic variables associated with it, &WBOX has 20 character variables, &PSW and &TSW each have 230 binary variables.

```
GBLA        &SBOX (50)
GBLC        &WBOX (20)
GBLB        &PSW (230)
LCLB        &TSW (230)
```

Using Subscripted SET Symbols: After you have associated a number of SET variables with a SET symbol, you may assign values to each of the variables and use them in other statements.

If the statements in the previous example were part of a macro definition, (and &A was defined as a SETA symbol in the same definition), the following statements could be part of the same macro definition:

```
1  &A          SETA    5
2  &PSW (&A)   SETB    (6 LT 2)
3  &TSW (9)    SETB    (&PSW (&A))
4              L       3, &SBOX (45)
5              L       4, &WBOX (17)
6              L       5, AREA
7              AR      2, 3
8              CR      5, 4
```

Statement 1 assigns the arithmetic value 5 to the nonsubscripted SETA symbol &A. Statements 2 and 3 then assign the binary value 0 to subscripted SETB symbols &PSW (5) and &TSW (9), respectively. Statements 4, 5 and 6 generate statements that load register 3, 4, and 5 with the values in storage represented by &SBOX (45), &WBOX (17) and AREA, respectively. Statements 7 and 8 generate statements that add register 3 to register 2 and compare the contents of register 4 with the contents of register 5.

System Variable Symbols

System variable symbols are local variable symbols that are assigned values automatically by the assembler. There are three system variable symbols: &SYSNDX, &SYSECT, and &SYSLIST. System variable symbols may be used in the name, operation and operand fields of statements in macro definitions, but not in statements outside macro definitions. The may not be defined as symbolic parameters or SET symbols, nor may they be assigned values by SETA, SETB, and SETC instructions.

&SYSNDX -- Macro Instruction Index

The system variable symbol &SYSNDX may be concatenated with other characters to create unique names for statements generated from the same model statement.

&SYSNDX is assigned the four-digit number 0001 for the first macro instruction processed by the assembler, and it is incremented by one for each subsequent inner and outer macro instruction processed.

If &SYSNDX is used in a model statement, SETC or MNOTE instruction, or a character relation in a SETB or AIF instruction, the value substituted for &SYSNDX is the four-digit number of the macro instruction being processed, including leading zeros.

If &SYSNDX appears in arithmatic expressions (for example, in the operand field of a SETA instruction), the value used for &SYSNDX is an arithmetic value.

Throughout one use of a macro definition, the value of &SYSNDX may be considered a constant, independent of any inner macro instruction in that definition.

The example in the next column illustrates these rules. It is assumed that the first macro instruction processed, OUTER 1, is the 106th macro instruction processed by the assembler.

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

```
                 MACRO
                 INNER
                 GBLC     &NDXNUM
1  A&SYSNDX      SR       2, 5
                 CR       2, 5
2                BZL      B&NDXNUM
```

```
3              B           A&SYSNDX
               MEND

               MACRO
  &NAME        OUTER1
               GBLC        &NDXNUM
4 &NDXNUM      SETC        '&SYSNDX'
  &NAME        SR          2,4
               AR          2,6
5              INNER1
6 B&SYSNDX     LA          2,100
               MEND

7 ALPHA        OUTER1
8 BETA         OUTER1

  ALPHA        SR          2,4
               AR          2,6
  A0107        SR          2,5
               CR          2,5
               BZL         B0106
               B           A0107
  B0106        LA          2,1000
  BETA         SR          2,4
  A            AR          2,6
  A0109        SR          2,5
               CR          2,5
               BZL         B0108
               B           A0109
  B0108        LA          2,1000
```

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

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

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

&SYSECT -- Current Control Section

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

When &SYSECT is used in a macro definition, the value substituted for &SYSECT is the name of the last CSECT, DSECT, or START statement that occurs before the macro instruction. If no named CSECT, DSECT, or START statements occur before a macro instruction, &SYSECT is assigned a null character value for that macro instruction.

CSECT or DSECT statements processed in a macro definition affect the value for &SYSECT for any subsequent inner macro instructions in that definition, and for any other outer and inner macro instructions.

Throughout the use of a macro definition, the value of &SYSECT may be considered a constant, independent of any CSECT or DSECT statements or inner macro instructions in that definition.

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

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

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

```
    Name         Operation     Operand

                 MACRO
                 INNER         &INCSECT
1   &INCSECT     CSECT
2                DC            A(&SYSECT)
                 MEND

                 MACRO
                 OUTER1
3   CSOUT1       CSECT
                 DS            100C
4                INNER         INA
5                INNER         INB
6                DC            A(&SYSECT)
                 MEND

                 MACRO
7                OUTER2
                 DC            A(&SYSECT)
                 MEND

8   MAINPROG     CSECT
                 DS            200C
9                OUTER1
10               OUTER2

    MAINPROG     CSECT
                 DS            200C
```

```
CSOUT1    CSECT
          DS      100C
INA       CSECT
          DC      A(CSOUT1)
INB       CSECT
          DC      A(INA)
          DC      A(MAINPROG)
          DC      A(INB)
```

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

&SYSLIST -- Macro Instruction Operand

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

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

&SYSLIST(n) may be used to refer to the nth positional macro instruction operand. In addition, if the nth operand is a sublist, then &SYSLIST (n,m) may be used to refer to the mth operand in the sublist, where n and m may be any arithmetic expressions allowed in the operand field of a SETA statement. M may be equal to, or greater than, 1 and N has a range of from 1 to 200.

DOS - A range of 1 to 100.

The type, length, and count attributes of &SYSLIST(n) and &SYSLIST(n,m) and the number attributes of &SYSLIST(n) and &SYSLIST may be used in conditional assembly instructions. N'&SYSLIST may be used to refer to the total number of positional operands in a macro instruction statement. N'&SYSLIST(n) may be used to refer to the number of operands in a sublist. If the nth operand is omitted, N' is zero; if the nth operand is not a sublist, N' is one.

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

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

Examples:

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

Keyword Macro Definitions and Instructions

Keyword macro definitions provide an alternate way of preparing macro definitions.

A keyword macro definition enables you to reduce the number of operands in each macro instruction that corresponds to the definition and to write the operands in any order.

The macro instructions that correspond to the macro definitions just described (hereafter called positional macro instructions and positional macro definitions, respectively) require the operands to be written in the same order as the corresponding symbolic parameters in the operand field of the prototype statement.

In a keyword macro definition, you can assign standard values to any symbolic parameters that appear in the operand field of the prototype statement. The standard value assigned to a symbolic parameter is substituted for the symbolic parameter, if you do not write anything in the operand field of the macro instruction to correspond to the symbolic parameter.

When a keyword macro instruction is written, you need write only one operand for each symbolic parameter whose value you want to change.

Keyword macro definitions are prepared in the same way as positional macro definitions, except that the prototype statement is written differently and &SYSLIST may not be used in the definition.

Keyword Prototype

The format of this statement is shown in Figure 60.

Name	*Operation*	*Operand*
A symbolic parameter or blank	A symbol	One or more operands of the form described in the following text, separated by commas.

Figure 60. Keyword Prototype Statement Format

Each operand must consist of a symbolic parameter, immediately followed by an equal sign and, optionally, followed by a standard value. This value must not include a keyword.

A standard value that is part of an operand must immediately follow the equal sign.

Anything that may be used as an operand in a macro instruction, except variable symbols, may be used as a standard value in a keyword prototype statement. The

rules for forming valid macro instruction operands have been previously discussed.

The following are valid keyword prototype operands:

&READER=
&LOOP2=SYMBOL

The following are invalid keyword prototype operands:

CARDAREA	(no symbolic parameter)
&TYPE	(no equal sign)
&TWO=123	(equal sign does not immediately follow symbolic parameter)
&AREA=X'189A'	(standard value does not immediately follow equal sign)

The following keyword prototype statement contains a symbolic parameter in the name field and four operands in the operand field. The first two operands contain standard values. The mnemonic operation code is MOVE.

```
&N        MOVE      &R=2,&A=S,&T=,&F=
```

Keyword Macro Instruction

After you have prepared a keyword macro definition, you may use it by writing a keyword macro instruction.

The format of a keyword macro instruction is shown in Figure 61.

Name	*Operation*	*Operand*
A symbol, sequence symbol, or blank	Mnemonic operation code	Zero or more operands of the form described in the following text, separated by commas.

Figure 61. Keyword Macro Instruction Format

Each operand consists of a keyword immediately followed by an equal sign and an optional value which may not include a keyword. Anything that may be used as an operand in a positional macro instruction may be used as a value in a keyword macro instruction. The rules for forming valid positional macro instruction operands are detailed in the preceding text under *Macro Instruction Prototype.*

A keyword consists of one through seven letters and digits, the first of which must be a letter.

The keyword part of each keyword macro instruction operand must correspond to one of the symbolic parameters that appears in the operand field of the keyword prototype statement. A keyword corresponds to a symbolic parameter if the characters of the keyword are identical to the characters of the symbolic parameter that follow the ampersand.

The following are valid keyword macro instruction operands:

LOOP2=SYMBOL
TO=

The following are invalid keyword macro instruction operands:

&X4F2=0(2, 3)	(keyword does not begin with a letter)
CARDAREA=A+2	(keyword is more than seven characters)
=(TO (8) , (FROM))	(no keyword)

The operands in a keyword macro instruction may be written in any order. If an operand appeared in a keyword prototype statement, a corresponding operand does not have to appear in the keyword macro instruction. If an operand is omitted, the comma that would have separated it from the next operand need not be written.

The following rules are used to replace the symbolic parameters in the statements of a keyword macro definition:

1. If a symbolic parameter appears in the name field of the prototype statement and the name field of the macro instruction contains a symbol, the symbolic parameter is replaced by the symbol. If the name field of the macro instruction is blank or contains a sequence symbol, the symbolic parameter is replaced by a null character value.
2. If a symbolic parameter appears in the operand field of the prototype statement and the macro instruction contains a keyword that corresponds to the symbolic parameter, the value assigned to the keyword replaces the symbolic parameter.
3. If a symbolic parameter was assigned a standard value by a prototype statement and the macro instruction does not contain a keyword that corresponds to the symbolic parameter, the standard value assigned to the symbolic parameter replaces the symbolic parameter. Otherwise, the symbolic parameter is replaced by a null character value.

   Note: If a standard value is a self-defining term, the type attribute assigned to the standard value is the letter N. If a standard value is omitted, the type attribute assigned to the standard value is the letter O. All other standard values are assigned the type attribute U.

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

Statement 1 assigns the standard values 2 and S to the symbolic parameters &R and &A, respectively. Statement 6 assigns the values FA, FB, and THERE to the keywords

T, F, and A, respectively. The symbol HERE is used in the name field of statement 6.

Since a symbolic parameter (&N) appears in the name field of the prototype statement (statement 1) and the corresponding characters (HERE) of the macro instruction (statement 6) are a symbol, &N is replaced by HERE in statement 2.

```
           MACRO
1  &N      MOVE    &R=2,&A=S,&T=,&F=
2  &N      ST      &R, &A
3          L       &R, &F
4          ST      &R, &T
5          L       &R, &A
           MEND
6  HERE    MOVE    T=FA,F=FB,A=THERE

   HERE    ST      2, THERE
           L       2, FB
           ST      2, FA
           L       2, THERE
```

Since &T appears in the operand field of statement 1 and statement 6 contains the keyword (T) that corresponds to &T, the value assigned to T (FA) replaces &T in statement 4. Similarly, FB and THERE replaces &F and &A in statement 3 and in statements 2 and 5, respectively. Note that the value assigned to &A in statement 6 is used instead of the value assigned to &A in statement 1.

Since &R appears in the operand field of statement 1, and statement 6 does not contain a corresponding keyword, the value assigned to &R (2) replaces &R in statements 2, 3, 4, and 5.

Operand Sublists: The value assigned to a keyword and the standard value assigned to a symbolic parameter may be an operand sublist. Anything that may be used as an operand sublist in a positional macro instruction may be used as a value in a keyword macro instruction and as a standard value in a keyword prototype statement. The rules for forming valid operand sublists are detailed in the preceding text under *Operand Sublists.*

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

Mixed-Mode Macro Definitions and Instructions

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

Mixed-mode macro definitions are prepared in the same way as positional macro definitions, except that the prototype statement is written differently. If &SYSLIST is used, it refers only to the positional operands in the prototype. Subscripting past the last positional parameter will yield an empty string and a type attribute of "0".

Mixed-Mode Prototype

The format of this statement is shown in Figure 62;

Name	*Operation*	*Operand*
A symbolic parameter or blank	A symbol	Two or more operands of the form described in the following text, separated by commas.

Figure 62. Mixed-Mode Prototype Statement Format

The operands must be valid operands of positional and keyword prototype statements. All of the positional operands must precede the first keyword operand. The rules for forming positional operands are discussed under *Macro Instruction Prototype.* The rules for forming keyword operands have been previously discussed under *Keyword Prototype.*

The following sample mixed-mode prototype statement contains three positional operands and two keyword operands:

```
&N          MOVE     &TY,&P,&R,&TO=,&F=
```

Mixed-Mode Macro Instruction

The format of a mixed-mode macro instruction is shown in Figure 63:

Name	*Operation*	*Operand*
A symbol, sequence symbol, or blank	Mnemonic operation code	Zero or more operands of the form described in the following text, separated by commas.

Figure 63. Mixed-Mode Macro Instruction Format

The operand field consists of two parts. The first part corresponds to the positional prototype operands. This part of the operand field is written in the same way that the operand field of a positional macro instruction is written.

The second part of the operand field corresponds to the keyword prototype operands. This part of the operand

field is written in the same way that the operand field of a keyword macro instruction is written. The rules for writing keyword macro instructions have been described previously under *Keyword Macro Instruction.*

The following mixed-mode macro definition, mixed-mode macro instruction, and generated statements illustrate these facilities:

```
   Name      Operation   Operand

             MACRO
1  &N        MOVE        &TY,&P,&R,&TO=&F=
   &N        ST&TY       &R, SAVE
             L&TY        &R, &P&F
             ST&TY       &R, &P&TO
             L&TY        &R, SAVE
2  HERE      MOVE        H,,2,F=FB,TO=FA
```

```
HERE     STH      2, SAVE
         LH       2, FB
         STH      2, FA
         LH       2, SAVE
```

The prototype statement (statement 1) contains three positional operands (&TY, &P, and &R) and two keyword operands (&TO and &F). In the macro instruction (statement 2), the positional operands are written in the same order as the positional operands in the prototype statement (the second operand is omitted). The keyword operands are written in an order that is different from the order of keyword operands in the prototype statement.

Mixed-mode inner macro instructions may be used as model statements in mixed-mode, keyword, and positional macro definitions. Keyword and positional inner macro instructions may be used as model statements in mixed-mode macro definitions.

Appendix A: Assembler Languages -- Features Comparison

Features not shown below are common to all assemblers. In the chart:
Dash = Not allowed.
X = As defined in Operating System/360 Assembler Language Manual, GC28-6514

Feature	Basic Programming Support Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	DOS/TOS Assembler	OS Assembler	Communications Controller Assembler-DOS	Communications Controller Assembler-OS
No. of Continuation Cards/Statement (exclusive of macro instructions)	0	0	1	1	2	1	2
Input Character Code	EBCDIC	BCD & EBCDIC	EBCDIC	EBCDIC	EBCDIC	EBCDIC	EBCDIC
ELEMENTS:							
Maximum characters per symbol	6	6	8	8	8	8	8
Character self-defining terms	1 Char. only	X	X	X	X	X	X
Binary self-defining terms	- -	- -	X	X	X	X	X
Length attribute reference	- -	- -	X	X	X	X	X
Literals	- -	- -	X	X	X	- -	- -
Extended mnemonics	- -	X	X	X	X	X	X
Maximum Location Counter value	$2^{16} - 1$	$2^{24} - 1$	$2^{24} - 1$	$2^{24} - 1$	$2^{24} - 1$	$2^{18} - 1$	$2^{18} - 1$
Multiple Control Sections per assembly	- -	- -	X	X	X	X	X
EXPRESSIONS:							
Operators	+-*	+-*/	+-*/	+-*/	+-*/	+-*/	+-*/
Number of terms	3	16	3	16	16	16	16
Levels of parentheses	- -	- -	1	5	5	5	5
Complex relocatability	- -	- -	X	X	X	X	X
ASSEMBLER INSTRUCTIONS:							
DC and DS							
Expressions allowed as modifiers	- -	- -	- -	X	X	X	X
Multiple operands	- -	- -	- -	X	X	- -	X
Multiple constants in an operand	- -	- -	Except Address Constants	X	X	- -	X
Bit length specifications	- -	- -	- -	X	X	- -	- -
Scale modifier	- -	- -	X	X	X	- -	- -
Exponent modifier	- -	- -	X	X	X	- -	- -
DC types	Except B, P, Z V, Y, S, L	Except B, V, L	Except L	Except L	X	Except E, D, L, P, Z, Q, S	Except E, D, L, P, Z, Q, S
DC duplication factor	Except A	X	Except S	X	X	X	X

Figure 64. Features Comparison (Part 1 of 3)

Feature	Basic Programming Support Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	DOS/TOS Assembler	OS Assembler	Communications Controller Assembler-DOS	Communications Controller Assembler-OS
DC duplication factor of zero	--	--	Except S	X	X	X	X
DC length modifier	Except H, E, D	X	X	X	X	X	X
DS types	Only C, H, F, D	Only C, H, F, D	Except L	Except L	X	Except E, D, L, P, Z, Q, S	Except E, D, L, P, Z, Q, S
DS length modifier	Only C	Only C	X	X	X	X	X
DS maximum length modifier	256	256	256	65,535	65,535	65,535	65,535
DS constant subfield permitted	--	--	X	X	X	X	X
COPY	--	--	--	X	X	X	X
CSECT	--	--	X	X	X	X	X
DSECT	--	--	X	X	X	X	X
ISEQ	--	--	X	X	X	X	X
LTORG	--	--	X	X	X	--	--
PRINT	--	--	X	X	X	X	X
TITLE	--	--	X	X	X	X	X
COM	--	--	--	X	X	X	X
ICTL	1 operand (1 or 25 only)	1 operand	X	X	X	X	X
USING	2 operands (operand 1 relocatable only)	2-17 operands (operand 1 relocatable only)	6 operands	X	X	X	X
DROP	1 operand only	X	5 operands	X	X	X	X
CCW	operand 2 (relocatable only)	X	X	X	X	--	--
ORG	no blank operand	no blank operand	X	X	X	X	X
ENTRY	1 operand only	1 operand only	1 operand only	X	X	X	X
EXTRN	1 operand only (max 14)	1 operand only	1 operand only	X	X	X	X
CNOP	2 decimal digits	2 decimal digits	2 decimal digits	X	X	X	X
PUNCH	--	--	--	X	X	X	X
REPRO	--	--	X	X	X	X	X
Macro instructions	--	--	X	X	X	X	X
OPSYN	--	--	--	--	X[1]	--	--
EQU	X	X	X	X	X	X	X
EQUR	--	--	--	--	--	X	X
CXD	X	X	X	X	X	--	--
DXD	X	X	X	X	X	--	--
CW	--	--	--	--	--	X	X

[1] Assembler F only

Figure 64. Features Comparison (Part 2 of 3)

Macro Facility Features	BPS 8K Tape, BOS 8K Disk Assemblers	BOS 16K Disk/Tape Assembler	OS Assembler	Communications Controller Assembler -DOS	Communications Controller Assembler -OS
Operand Sublists	- -	X	X	X	X
Attributes of macro-instruction operands inside macro definitions and symbols used in conditional assembly instructions outside macro definitions.	- -	X	X	X	X
Subscripted SET symbols	- -	X	X	X	X
Maximum number of operands	49	100[1]	200	100	200
Conditional assembly instructions outside macro definition	- -	X	X	X	X
Maximum number of SET symbols					
global SETA	16	*	*	*	*
global SETB	128	*	*	*	*
global SETC	16	*	*	*	*
local SETA	16	*	*	*	*
local SETB	128	*	*	*	*
local SETC	0	*	*	*	*

* The number of SET symbols permitted is variable, depending upon available main storage.

Note: The maximum size of a character expression is 127 DOS/TOS Assembler D and 255 characters in OS and Assembler F.

200 for Assembler F

Figure 64. Features Comparison (Part 3 of 3)

Instruction	*Format Code*	*Mnemonic*	*Operand Field Format**
Branch	RT	B	T
Branch on C Latch	RT	BCL	T
Branch on Z Latch	RT	BZL	T
Branch on Bit	RT	BB	R (N, M), T
Branch on Count	RT	BCT	R (N), T
Branch and Link	RA	BAL	R, A
Branch and Link Register	RR	BALR	R1, R2
Add Register	RR	AR	R1, R2
Add Halfword Register	RR	AHR	R1, R2
Add Character Register	RR	ACR	R1 (N1), R2 (N2)
Add Register Immediate	RI	ARI	R (N), I
Subtract Register	RR	SR	R1, R2
Subtract Halfword Register	RR	SHR	R1, R2
Subtract Character Register	RR	SCR	R1 (N1), R2 (N2)
Subtract Register Immediate	RI	SRI	R (N), I
Insert Character	RS	IC	R (N), D (B)
Insert Character and Count	RSA	ICT	R (N), B
Load	RS	L	R, D (B)
Load Halfword	RS	LH	R, D (B)
Load Register	RR	LR	R1, R2
Load Halfword Register	RR	LHR	R1, R2
Load Character Register	RR	LCR	R1 (N1), R2 (N2)
Load Register Immediate	RI	LRI	R (N), I
Load Address	RA	LA	R, A
Load with Offset Register	RS	LOR	R1, R2
Load Halfword with Offset Reg.	RR	LHOR	R1, R2
Load Character with Offset Reg.	RR	LCOR	R1 (N1), R2 (N2)
Store	RS	ST	R, D (B)
Store Halfword	RS	STH	R, D (B)
Store Character	RS	STC	R (N), D (B)
Store Character and Count	RSA	STCT	R (N), B
Compare Register	RR	CR	R1, R2
Compare Halfword Register	RR	CHR	R1, R2
Compare Character Register	RR	CCR	R1 (N1), R2 (N2)
Compare Register Immediate	RI	CRI	R (N), I
AND Register	RR	NR	R1, R2
AND Halfword Register	RR	NHR	R1, R2
AND Character Register	RR	NCR	R1 (N1), R2 (N2)
AND Register Immediate	RI	NRI	R (N), I
OR Register	RR	OR	R1, R2
OR Halfword Register	RR	OHR	R1, R2
OR Character Register	RR	OCR	R1 (N1), R2 (N2)
OR Register Immediate	RI	ORI	R (N), I
Exclusive OR Register	RR	XR	R1, R2
Exclusive OR Halfword Register	RR	XHR	R1, R1

Figure 65. Instruction Format (Part 1 of 2)

Instruction	*Format Code*	*Mnemonic*	*Operand Field Format**
Exclusive OR Register Immediate	RI	XRI	R (N), I
Exclusive OR Character Register	RR	XCR	R1 (N1), R2 (N2)
Test Register Under Mask	RI	TRM	R (N), I
Exit	EXIT	EXIT	
Input	RE	IN	R, E
Output	RE	OUT	R, E

Notes:

**Operand Field Symbol*	*Description*
B	an absolute expression that specifies a base register.
D	an absolute expression that specifies a displacement.
E	an absolute expression that specifies an external register.
I	an absolute expression that provides immediate data.
M	an absolute expression that specifies a bit.
N	N, N1, and N2 are absolute expressions that specify a byte. The value may be either 0 or 1.
Q	Q, Q1, and Q2 are symbolic register expressions that specify a register-byte combination. (See EQUR.)
R	R, R1, and R2 are absolute expressions that specify general registers. Registers are numbered 0 through 7.
S	Either an absolute or relocatable expression specifying an implied address (used in conjunction with a USING statement).
T	A relocatable expression that specifies a transfer address.

Figure 65. Instruction Format (Part 2 of 2)

Type	*Implied Length (Bytes)*	*Alignment*	*Length Modifier Range*	*Specified By*	*Number of Constants Per Operand*	*Truncation/Padding Side*
C	as needed	byte	1 to 256*	characters	one	right
X	as needed	byte	1 to 256*	hexadecimal digits	one	left
B	as needed	byte	1 to 256	binary digits	one	left
F	4	fullword	1 to 8	decimal digits	multiple	left
H	2	halfword	1 to 8	decimal digits	multiple	left
A	4	fullword	1 to 4**	any expression	multiple	left
V	4	fullword	3 or 4	relocatable symbol	multiple	left
R	2	halfword	2 only	any expression	multiple	left
Y	2	halfword	1 to 2	any expression	multiple	left

*In a DS assembler instruction C and X type constants may have length specification to 65535.

**Errors will be flagged if significant bits are truncated or if the value specified cannot be contained in the implied length of the constant.

Figure 66. Summary of Constants

OPERATION	*NAME ENTRY*	*OPERAND ENTRY*
ACTR	Must not be present	An arithmetic SETA expression
AGO	A sequence symbol or not present	A sequence symbol
AIF	A sequence symbol or not present	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP	A sequence symbol	Must not be present
CW	Any symbol or not present	Four operands, separated by commas
CNOP	A sequence symbol or not present	Two absolute expressions, separated by a comma
COM	A sequence symbol or not present	Must not be present
COPY	Must not be present	A symbol
CSECT	Any symbol or not present	Must not be present
DC	Any symbol or not present	One or more operands, separated by commas
DROP	A sequence symbol or not present	One to sixteen absolute expressions, separated by commas
DS	Any symbol or not present	One or more operands, separated by commas
DSECT	A variable symbol or an ordinary symbol	Must not be present
EJECT	A sequence symbol or not present	Must not be present
END	A sequence symbol or not present	A relocatable expression or not present
ENTRY	A sequence symbol or not present	One or more relocatable symbols, separated by commas
EQU	A variable symbol or an ordinary symbol	An absolute or relocatable expression
EQUR	A variable symbol or an ordinary symbol	An expression grouping of the form R (N) or Q.
EXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas

OPERATION	*NAME ENTRY*	*OPERAND ENTRY*
GBLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas [2]
GBLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas [2]
GBLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas [2]
ICTL	Must not be present	One to three decimal values, separated by commas
ISEQ	Must not be present	Two decimal values, separated by a comma
LCLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas [2]
LCLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas [2]
LCLC	Must not be present	One or more variable symbols separated by commas
MACRO [1]	Must not be present	Must not be present
MEND [1]	A sequence symbol or not present	Must not be present
MEXIT [1]	A sequence symbol or not present	Must not be present
MNOTE [1]	A sequence symbol, a variable symbol or not present	A severity code, followed by a comma, followed by any combination of characters enclosed in apostrophes
ORG	A sequence symbol or not present	A relocatable expression or not present
PRINT	A sequence symbol or not present	One to three operands
PUNCH	A sequence symbol or not present	One to eighty characters, enclosed in apostrophes
REPRO	A sequence symbol or not present	Must not be present
SETA	A SETA symbol	An arithmetic expression
SETB	A SETB symbol	A 0 or a 1, or logical expression, enclosed in parentheses
SETC	A SETC symbol	A type attribute, a character expression, a substring notation, or a concatenation of character expressions and substring notations
SPACE	A sequence symbol or not present	A decimal self-defining term or not present
START	Any symbol or not present	A self-defining term or not present

OPERATION	*NAME ENTRY*	*OPERAND ENTRY*
TITLE [3]	A special symbol (0 to 4 characters), a sequence symbol, a variable symbol, or not present	One to 100 characters, enclosed in apostrophes
USING	A sequence symbol or not present	An absolute or relocatable expression followed by 1 to 16 absolute expressions, separated by commas

Notes:

1 May be used only as part of a macro definition.

2 SET symbols may be defined as subscripted SET symbols.

3 See *Chapter 4, Communications Controller Assembler Instructions,* for a description of the name entry.

ASSEMBLER STATEMENTS

Instruction	*Name Entry*	*Operand Entry*
Model Statements [3,4]	An ordinary symbol, a variable symbol, sequence variable symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present.	Any combination of characters (including variable symbols)
Prototype Statement [1]	A symbolic parameter or not present	Zero or more operands that are symbolic parameters, separated by commas, followed by zero or more operands (separated by commas) of the form symbolic parameter, equal sign, optional standard value
Macro Instruction Statement [1]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, [2] or not present	Zero or more positional operands, separated by commas, followed by zero or more keyword operands (separated by commas) of the form keyword, equal sign, value [2]
Assembler Language Statement [4]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present	Any combination of characters (including variable symbols)

Notes:

1 May be used only as part of a macro definition.

2 Variable symbols appearing in a macro instruction are replaced by their values before the macro instruction is processed.

3 Variable symbols may be used to generate assembler language mnemonic operation codes as listed in Chapter 4, except ACTR, COPY, END, ICTL, CSECT, DSECT, ISEQ, PRINT, REPRO, and START. Variable symbols may not be used in the name and operand entries of the following instructions: COPY, END, ICTL, and ISEQ. Variable symbols may not be used in the name entry of the ACTR instruction.

4 No substitution for variables in the line following a REPRO statement is performed.

Figure 67. Assembler Statements

Figures 68, 69, 70, and 71 in this appendix summarize the macro language.

Figure 68 indicates which macro language elements may be used in the name and operand entries of each statement.

Figure 69 is a summary of the expressions that may be used in macro instruction statements.

Figure 70 is a summary of the attributes that may be used in each expression.

Figure 71 is a summary of the variable symbols that may be used in each expression.

	Variable Symbols										Attributes				
		Global SET Symbols			Local SET Symbols			System Variable Symbols							
Statement	Symbolic Parameter	SETA	SETB	SETC	SETA	SETB	SETC	&SYSNDX	&SYSECT	&SYSLIST	Type	Length	Count	Number	Sequence Symbol
MACRO															
Prototype Statement	Name Operand														
GBLA		Operand													
GBLB			Operand												
GBLC				Operand											
LCLA					Operand										
LCLB						Operand									
LCLC							Operand								
Model Statement	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand					Name
SETA	Operand[2]	Name Operand	Operand[3]	Operand[9]	Name Operand	Operand[3]	Operand[9]	Operand		Operand[2]		Operand	Operand	Operand	
SETB	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	
SETC	Operand	Operand[7]	Operand[8]	Name Operand	Operand[7]	Operand[8]	Name Operand	Operand	Operand	Operand	Operand				
AIF	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	Name Operand
AGO															Name Operand
ACTR	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[3]	Operand[2]	Operand		Operand[2]		Operand	Operand	Operand	
ANOP															Name
MEXIT															Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand					Name
MEND															Name
Outer Macro		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand								Name
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand					Name
Assembler Language Statement		Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand								Name

1. Variable symbols in macro-instructions are replaced by their values before processing.
2. Only if value is self-defining term.
3. Converted to arithmetic +1 or +0.
4. Only in character relations.
5. Only in arithmetic relations.
6. Only in arithmetic or character relations.
7. Converted to unsigned number.
8. Converted to character 1 or 0.
9. Only if one to eight decimal digits.

Figure 68. Macro Language Elements

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions
May contain	1. Self-defining terms 2. Length, count, and number attributes 3. SETA and SETB symbols 4. SETC symbols whose value is 1-8 decimal digits 5. Symbolic parameters if the corresponding operand is a self-defining term 6. &SYSLIST(n) if the corresponding operand is a self-defining term 7. &SYSLIST(n,m) if the corresponding operand is a self-defining term 8. &SYSNDX	1. Any combination of characters enclosed in apostrophes 2. Any variable symbol enclosed in apostrophes 3. A concatenation of variable symbols and other characters enclosed in apostrophes 4. A request for a type attribute	1. SETB symbols 2. Arithmetic relations [1] 3. Character relations [2]
Operators are	+,-,*, and / parentheses permitted	concatenation, with a period (.)	AND, OR, and NOT parentheses permitted
Range of values	-2^{31} to $+2^{31}-1$	0 through 255 characters [3]	0 (false) or 1 (true)
May be used in	1. SETA operands 2. Arithmetic relations 3. Subscripted SET symbols 4. &SYSLIST 5. Substring notation 6. Sublist notation	1. SETC operands [3] 2. Character relations [2]	1. SETB operands 2. AIF operands

[1] An arithmetic relation consists of two arithmetic expressions related by the operators GT, LT, EQ, NE, GE, or LE.

[2] A character relation consists of two character expressions related by the operator GT, LT, EQ, NE, GE, or LE. The type attribute notation and the substring notation may also be used in character relations. The maximum size of the character expressions that can be compared is 255 characters for OS and 127 characters for DOS, see chapter 5 under SETC - SET CHARACTER. If the two character expressions are unequal size, then the smaller one will always compare less than the larger.

[3] Maximum of eight characters will be assigned.

Figure 69. Conditional Assembly Expressions

Attribute	Notation	May be used with:	May be used only if type attribute is:	May be used in *
Type	T'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m) inside macro definitions	(May always be used)	1. SETC operand fields 2. Character relations
Length	L'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m) inside macro definitions	Any leter except M, N, O, T, and U	Arithmetic expressions
Count	K'	Symbolic parameters corresponding to macro instruction operands, &SYSLIST(n), and &SYSLIST(n,m) inside macro definitions	Any letter	Arithmetic expressions
Number	N'	Symbolic parameters, &SYSLIST, and &SYSLIST(n) inside macro definitions	Any letter	Arithmetic expressions

***NOTE: There are definite restrictions in the use of these attributes. Refer to text, Chapter 5, under <u>Attributes</u>.**

Figure 70. Attributes

Variable Symbol	Defined by:	Initialized, or set to:	Value changed by:	May be used in:
Symbolic[1] parameter	Prototype statement	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions, if operand is self-defining term 2. Character expressions
SETA	LCLA or GBLA instruction	0	SETA instruction	1. Arithmetic expressions 2. Character expressions
SETB	LCLB or GBLB instruction	0	SETB instruction	1. Arithmetic expressions 2. Character expressions 3. Logical expressions
SETC	LCLC or GBLC instruction	Null character value	SETC instruction	1. Arithmetic expressions, if value is self-defining term 2. Character expressions
&SYSNDX[1]	The assembler	Macro instruction index	(Constant throughout definition; unique for each macro instruction)	1. Arithmetic expressions 2. Character expressions
&SYSECT[1]	The assembler	Control section in which macro instruction appears	(Constant throughout definition; set by CSECT, DSECT, and START)	Character expressions
&SYSLIST[1]	The assembler	Not applicable	Not applicable	N'&SYSLIST in arithmetic expressions
&SYSLIST(n)[1] &SYSLIST(n,m)[1]	The assembler	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions

[1] May be used only in macro definitions.

Figure 71. Variable Symbols

COMMUNICATIONS CONTROLLER JOB CONTROL LANGUAGE FOR OS

Figure 72 shows the control statements necessary to assemble a Communications Controller program under OS.

```
1 //ASM          EXEC    PGM=IFKASM, REGION=50K

2 //SYSLIB       DD      DSNAME=SYS1.MAC3705, DISP=SHR

3 //SYSUT1       DD      DSNAME=&SYSUT1, UNIT=SYSSQ, SPACE=(1700, (400, 50)),
  //                     SEP=(SYSLIB)

4 //SYSUT2       DD      DSNAME=&SYSUT2, UNIT=SYSSQ, SPACE=(1700, (400, 50))

5 //SYSUT3       DD      DSNAME=SYSUT3, SPACE=(7200, (400,50))
  //                     UNIT=(SYSSQ, SEP=(SYSUT2, SYSUT1, SYSLIB))

6 //SYSPRINT     DD      SYSOUT=A

7 //SYSPUNCH     DD      SYSOUT=B

  --------

  --------

  //SYSIN        DD      *

  Program to be assembled

  /*
```

Notes:

1 PARM=or COND=parameters may be added to this statement by the EXEC statement that calls the procedure. The system name IFKASM identifies the IBM Communications Controller Assembler.

2 This statement identifies the macro library data set. The data set name SYS1.MAC3705 is an IBM designation.

3, 4, 5 These statements specify the assembler utility data sets. The device classname used here, SYSSQ, may represent a collection of tape drives, or direct-access units, or both. The I/O units assigned to this name are specified by the installation when the system is generated. A unit name, (for example, 2311) may be substituted for SYSSQ. The DSNAME parameters guarantee use of dedicated work files if this feature is part of the scheduler.

The SEP=subparameter in statement 5 and the SPACE=parameter in statements 3, 4, and 5 are effective only if the device assigned is a direct-access device: otherwise they are ignored. The space required depends upon the make up of the source program. The *Job Control Language* publication explains space allocation.

6 This statement defines the standard system output class, SYSOUT=A, as the destination for the assembler listing.

7 This statement describes the data set that will contain the object module produced by the assembler.

Figure 72. Job Control Statement for Assembly Under OS

You may catalog the procedure to simplify your assembly, see the IEBUPDTE Program, in the publication *IBM System/360 Operating System: Utilities,* GC28-6586.

COMMUNICATIONS CONTROLLER JOB CONTROL FOR DOS

Figure 73 lists the control cards necessary to assemble a Communications Controller program under DOS. The card groups are listed in the order in which they must appear. All job control cards enter the system via SYSRDR; all others, via SYSIPT. The same device may be assigned for both SYSRDR and SYSIPT. If the device is a disk file, the combined file must be designated as SYSIN. Job Control statements are described in *IBM System/360 Disk Operating System, System Control and System Service Programs,* GC24-5036.

Note 1: Only those assignments and options not already in effect are required.

Note 2: Assignments for SYSIN and/or SYSOUT must be accomplished by permanent assignments. For details see the publication *DOS System Control and System Service Programs.*

Card Group	*Card Arrangement*	*Comments*
Job Control	// JOB	First card in group; always required
	// ASSGN SYSSLB, . .	Required for macros and copy code
	// ASSGN SYSIPT, . .	Source program input
	// ASSGN SYSLST, . .	Program listing

	// ASSGN SYS001, . . .
	// ASSGN SYS002, . . .
	// ASSGN SYS003, . . .
		Work files

	// ASSGN SYSPCH, . .	Required when DECK option is specified
	// ASSGN SYSLNK, . .	Required when assemble-and-execute is specified
	//OPTION DECK, . . .	Optional; used to indicate desired assembler functions
	// EXEC IFTASM	Required
Assembler Input	Source Deck	Source statements (machine, assembler, and macro instructions)
	/*	Indicates end-of-data set
Job Control	/&	End of job statement

1 SYSSLB is assigned to a private source statement library.

Figure 73. Job Control Statements for Assembly Under DOS

ASSEMBLER STORAGE REQUIREMENTS

OS Storage Requirements

The primary storage requirement for the assembler when operating in an MFT partition is a minimum of 48K bytes. The Assembler requires a minimum of 50K when operating in an MVT region.

Auxiliary Storage Requirements

The residence requirements are as follows:

Three Directory records

Device type	Tracks needed
2301	8
2302	29
2303	32
2311	40
2314	22

The work space requirements are described in Figure 74.

Dictionary Capacities

The capacity of the general dictionary (global dictionary and all local dictionaries) is up to 64 blocks of 1024 bytes each. The division of the dictionary into global and local sections is done dynamically: as the global dictionary becomes larger, it occupies blocks taken from the local dictionary area. Thus, the global dictionary is always core-resident. As it expands into the local dictionary area, the local dictionaries may overflow onto a utility file. The size of the dictionaries in core depends upon core availability. The minimum core allocation is three blocks for the global and two blocks for each local dictionary.

If an assembly is terminated, at collection time, with either a GLOBAL DICTIONARY FULL message or a LOCAL DICTIONARY FULL message, you can take one or more of the following steps:

1. Split the assembly into two or more parts and assemble each separately.
2. Allocate more core for the assembler (the global and local dictionaries together can occupy up to 64K).
3. Specify a smaller SYSLIB blocksize and try the assembly again.
4. Specify a smaller blocksize for the utility files (normal minimum is 1700 bytes).

If the assembly is terminated, at generation time, with a GENERATION TIME DICTIONARY AREA OVERFLOW message, you should allocate more core to the assembler and reassemble your program.

The assembler can usually handle 400 ordinary symbols without overflow in its minimum core, see Primary Storage Requirements above. The assembler can process one additional symbol for each 18 bytes above minimum core storage.

DOS Storage Requirements

The primary storage requirements for the Assembler is a minimum of 12K.

The auxiliary storage requirements are as follows:

- Residence requirements,

  Core Image Library

Device type	Tracks needed
2311	46
2314	23

  Relocatable Library

Device type	Tracks needed
2311	68
2314	40

- Work file requirements,
  The number of tracks can be determined from figure 75. Note that figure 75 is expressed in number of bytes. The approximate number of tracks can be calculated by dividing the number of bytes that are required by 3000 for a 2311, or by 6000 for a 2314 file. These numbers represent the approximate number of text bytes, per track, for a 2311 file and a 2314 file, respectively.

  For assemblies with macros, you must count the number of statements in the macro definitions and use the procedure just described.

Data Set	Number of Source Cards	Assembler Operating In	Number of Tracks Required								
			2301 Drum	2302 Disk	2303 Drum	2311 Disk	2314 Disk	2321 Data Cell	2305-1 Drum	2305-2 Drum	3330 Disk
SYSUT1	150	50K	2	6	6	8	5	14	3	3	3
		100K	2	8	8	8	8	15	3	3	3
		200K	2	8	8	8	8	15	3	3	3
	500	50K	4	15	15	20	11	35	6	6	6
		100K	5	19	19	20	19	37	6	6	6
		200K	5	19	19	20	19	37	6	6	6
	1000	50K	7	29	29	38	29	67	10	10	11
		100K	9	34	34	37	34	68	10	10	11
		200K	9	34	34	37	34	68	10	10	11
SYSUT2	150	50K	2	6	6	7	6	13	2	2	3
		100K	2	7	7	7	7	13	2	2	3
		200K	2	7	7	7	7	13	2	2	3
	500	50K	4	14	14	18	14	32	5	5	5
		100K	5	17	17	18	17	33	5	5	6
		200K	5	17	17	18	17	33	5	5	6
	1000	50K	7	26	26	34	26	60	9	9	10
		100K	8	30	30	33	30	60	9	9	10
		200K	8	30	30	33	30	60	9	9	10
SYSUT3	150	50K	1	3	3	3	3	6	1	1	1
		100K	1	3	3	3	3	6	1	1	1
		200K	1	3	3	3	3	6	1	1	1
	500	50K	1	4	4	5	4	9	2	2	2
		100K	2	5	5	5	5	10	2	2	2
		200K	2	5	5	5	5	10	2	2	2
	1000	50K	2	6	6	8	6	14	3	2	3
		100K	2	8	8	8	8	15	3	3	3
		200K	2	8	8	8	8	15	3	3	3

Note: These estimates are based on the assumption that no macro instructions are used in the source program. The storage required for SYSUT3 increases when macro instructions are used, and it is approximately equal to the storage required for SYSUT1, for a 100 card program.

Figure 74. Work Space for Assembly Under OS

	Number of Bytes per Statement			
	SYSLNK	SYS001	SYS002	SYS003
1 for 1 Statements	15	150	150	36

Figure 75. Work File Requirements for a 2311 and a 2314 Under DOS

Note: Only three files are required for an assembly SYS001, SYS002, and SYS003; SYSLNK would be used when you specify LINK on the OPTION card. Each statement places a space requirement on each file, for example, a 10 statement source program with a call to one macro containing 20 statements will need the following bytes on each file. Assume a 2311 is used.

SYSLNK

15(10) + 15(20) = 15(30) = 450 bytes

450/3000 = .15 = 1 track

SYS001 and SYS002

150(10) + 150(20) = 150(30) = 4500

4500/3000 = 1.5 = 2 tracks

SYS003

36(10) + 36(20) = 36(30) = 1080

1080/3000 = .36= 1 track

Appendix G: Communications Controller Assembler Messages

Component Name	IFK = OS IFT = DOS
Program Producing Message	IBM Communications Controller Assembler program during assembly of assembler instructions
Audience and Where Produced	For programmer: Assembler listing in SYSPRINT data set For operator: Console
Message Format	ss, ***IFKnnn text (in SYSPRINT) xx IFKnnn text (on console) ss Severity code indicating effect of error on execution of program being assembled: * Informational message; no effect on execution 0 Informational message; normal execution is expected 4 Warning message; successful execution is probable 8 Error; execution may fail 12 Serious error; successful execution is improbable 16 Terminal error; successful execution is impossible 20 Assembler program terminated abnormally nnn Message serisl number text Message text xx Message reply identification (absent, if operator reply not reqiiired)

Note: IFT messages ending with an "I" are printed on both SYSLST and SYSLOG unless one of the messages indicates that SYSLST or an unidentifiable unit is defective, in which case they will appear on SYSLOG only. The messages appearing on SYSLOG will be prefaced by an "A". 110I and 111I errors can be detected at any point during assembly.

112I through 115I errors are detected immediately upon assembly attempt -- no assembly listing is printed. In either case the assembly is terminated, the source is bypassed to a /* or EOF, and control is returned to the supervisor via EOJ. The subsequent steps of a multiple step JOB are not bypassed unless they also are defective.

IFK001 DUPLICATION FACTOR ERROR
IFT001

Explanation: A duplication factor is not an absolute expression. There is an * in duplication factor expression. There is invalid syntax in expression.

Severity Code: 12

Programmer Response: The duplication factor must be specified by an absolute expression enclosed in parentheses or by an unsigned decimal self-defining term. (See *Data Definition Instructions* in Chapter 4.)

IFK002 RELOCATABLE DUPLICATION FACTOR
IFT002

Explanation: A relocatable expression has been used to specify the duplication factor.

Severity Code: 12

Programmer Response: The duplication factor must be specified by either an unsigned decimal self-defining term, or by an absolute expression that is enclosed within parentheses.

IFK003 LENGTH ERROR
IFT003

Explanation: The length specification is out of permissible range or specified invalidly; * in length expression; invalid syntax in expression; no left-parenthesis delimiter for expression.

Severity Code: 12

Programmer Response: Ensure that the length specification is within permissible range and that the syntax is valid.

IFK004 RELOCATABLE LENGTH
IFT004

Explanation: A relocatable expression has been used to specify length.

Severity Code: 12

Programmer Responses: The length specification must be either an unsigned decimal self-defining term, or an absolute expression enclosed within parentheses.

IFK005 INVALID SYNTAX IN OPERAND
IFT005

Explanation: Syntax invalid (for example, symbolic register expression combined with another term).

Severity Code: 12

Programmer Response: Ensure that the syntax in the operand of the particular instruction used is correct.

IFK006 INVALID ORIGIN
IFT006

Explanation: The location counter has been reset to a value less than the starting address of the control section; ORG operand is not a simply relocatable expression or specifies an address outside the control section.

Severity Code: 12

Programmer Response: Ensure that the use of the ORG instruction does not reset the location counter to an address outside the control section.

IFK007 LOCATION COUNTER ERROR
IFT007

Explanation: Either the location counter has exceeded 2^{18}-1, or it has passed out of the control section in the negative direction.

Severity Code: 12

Programmer Response: This control section is too large. It must be broken into several smaller control sections and reassembled. Possibly an error was made in coding an ORG or DS instruction. Ensure that the instruction is free from error on reassemble. (See *Location Counter Reference* under *Terms,* in Chapter 2.)

IFK008 INVALID DISPLACEMENT
IFT008

Explanation: The transfer address of a branch instruction is outside the allowable range or the displacement of a base register instruction is outside the allowable range.

Severity Code: 8

Programmer Response: Ensure that either the transfer address, or the displacement of a base register instruction is inside the allowable range. (See *Location Counter Reference* under *Terms* in Chapter 2 and USING under *Base Register Instructions,* in Chapter 4.)

IFK009 MISSING OPERAND
IFT009

Explanation: Statement requires an operand entry and none is present.

Severity Code: 12

Programmer Response: Insert operand entry where indicated and reassemble program.

IFK010 INCORRECT REGISTER SPECIFICATION
IFT010

Explanation: The value specifying the register is not an absolute value within the range 0-7, an even register is specified where an odd register is required, or a register was used where none can be specified.

Servity Code: 12

Programmer Response: Ensure that the registers used are within the range of 0-7 and that the use of a register is permissible in the operation.

IFK011 INVALID ORIGIN FOR RELOCATABLE R-TYPE CONSTANT
IFT011

Explanation: An R-type address constant is assembled at location 0.

Severity Code: 8

Programmer Response: Probable user error. Ensure that the instruction is not assembled at location 0.

IFK012 (No message is assigned to this number.)
IFT012

IFK013 (No message is assigned to this number.)
IFT013

IFK014 (No message is assigned to this number.)
IFT014

IFK015 (No message is assigned to this number.)
IFT015

IFK016 INVALID NAME
IFT016

Explanation: A name entry is incorrectly specified; for example, it contains more than eight characters, it does not begin with a letter, or it has a special character imbedded.

Severity Code: 8

Programmer Response: Ensure that all name entries contain no more than eight characters, that they begin with a letter, and that they do not have any special characters imbedded.

IFK017 DATA ITEM TOO LARGE
IFT017

Explanation: The constant is too large for the data type or for the explicit length.

Severity Code: 8

Programmer Response: Lower the value or reduce the length of the constant to within permissible range. See Chapter 4 for a discussion of values for the various data types.

IFK018 INVALID SYMBOL
IFT018

Explanation: The symbol specification is invalid; for example, it has more than eight characters, or it has an imbedded special character.

Severity Code: 8

Programmer Response: Ensure that symbols have no more than eight characters and that they contain no imbedded special characters.

IFK019 EXTERNAL NAME ERROR
IFT019

Explanation: A CSECT and a DSECT statement have the same name: a symbol is used more than once in an EXTRN.

Severity Code: 8

Programmer Response: Replace the duplicate CSECT or DSECT name or symbol name in EXTRN.

IFK020 INVALID IMMEDIATE FIELD
IFT020

Explanation: The value of the immediate operand exceeds 255; the operand requires more than one byte of storage; the operand is not an acceptable type.

Severity Code: 8

Programmer Response: Ensure that the immediate operand value does not exceed 255, and that it does not require more than one byte of storage. Also ensure that the operand type is acceptable.

IFK021 SYMBOL NOT PREVIOUSLY DEFINED
IFT021

Explanation: An expression requiring that all symbols be previously defined contains at least one symbol not predefined.

Severity Code: 8

Programmer Response: Define the symbol requiring definition and reassemble the program.

IFK022 ESD TABLE OVERFLOW
IFT022

Explanation: The combined number of control sections and dummy sections plus the number of unique symbols in EXTRN statements and V-type constants exceeds 255.

Severity Code: 12

Programmer Response: Ensure that the combined number of CSECTs and DSECTs plus the number of unique symbols in EXTRN statements and V-type constants do not exceed 255

IFK023 PREVIOUSLY DEFINED NAME
IFT023

Explanation: The symbol which appears in the name field has appeared in the name field of a previous statement.

Severity Code: 8

Programmer Response: Redefine the duplicate symbol in the name field and reassemble the program.

IFK024 UNDEFINED SYMBOL
IFT024

Explanation: A symbol being referred to has not been defined in the program.

Severity Code: 8

Programmer Response: Ensure that all symbols being referred to have been defined. (See *Symbols* under *Terms,* in Chapter 2.)

IFK025 RELOCATABILITY ERROR
IFT025

Explanation: A relocatable expression, a complex relocatable expression, or a symbolic register is specified where an absolute expression is required; an absolute expression, symbolic register, or complex relocatable expression is specified where a relocatable expression is required; a relocatable term is involved in multiplication or division.

Severity Code: 8

Programmer Response: Ensure that where absolute expressions are required, only absolute expressions are specified. Ensure that where relocatable expressions are required, only relocatable expressions are specified. Ensure that relocatable terms are not involved in multiplication or division. (See *Absolute and Relocatable Expressions* under *Expressions,* in Chapter 3.)

IFK026 TOO MANY LEVELS OF PARENTHESES
IFT026

Explanation: An expression specifies more than 5 levels of parentheses.

Severity Code: 12

Programmer Response: Ensure that no expression contains more than 5 levels of parentheses. (See *Terms in Parentheses* under *Terms,* in Chapter 2.)

IFK027 TOO MANY TERMS
IFT027

Explanation: More than 16 terms are specified in an expression.

Severity Code: 12

Programmer Response: Ensure that no more than 16 terms are specified in an expression.

IFK028 REGISTER NOT USED
IFT028

Explanation: A register specified in a DROP statement is not currently in use.

Severity Code: 4

Programmer Response: Execution is probable, the DROP statement was probably not needed. (See *DROP* under *Base Register Instructions,* in Chapter 4.)

IFK029 CW ERROR
IFT029

Explanation: The command code or FLAG value exceeds 3, or the count exceeds 1023 in a CW Instruction.

Severity Code: 8

Programmer Response: Ensure that the command code or FLAG value does not exceed 3 and that the count does

not exceed 1023. (See *CW* under *Data Definitions Instructions,* in Chapter 4.)

IFK030 INVALID CNOP
IFT030

Explanation: An invalid combination of operands is specified.

Severity Code: 12

Programmer Response: Ensure that the CNOP statement operands are properly specified. (See *CNOP* under *Program Control Instructions,* Chapter 4.)

IFK031 UNKNOWN TYPE
IFT031

Explanation: Incorrect type designation is specified in a DC or DS.

Severity Code: 8

Programmer Response: Ensure that the type designations specified in a DC or DS are correct.

IFK032 OP-CODE NOT ALLOWED TO BE GENERATED
IFT032

Explanation: Variable symbols may not be used to generate:

- Macro instructions
- Assembler instructions not appearing in Chapter 4
- END, ICTL, ISEQ, PRINT, or REPRO instructions.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK033 ALIGNMENT ERROR
IFT033

Explanation: The address referred to is not aligned to the proper boundary for this instruction, for example, the START operand is not a multiple of 8, or the RS instruction displacement is not divisible by 2 or 4.

Severity Code: 4

Programmer Response: Make sure that the address referred to is aligned to the proper boundary for this instruction.

IFK034 INVALID OP-CODE
IFT034

Explanation: Syntax error; for example, there are more than eight characters; or the operation field is not followed by a blank.

Severity Code: 8

Programmer Response: Ensure that syntax is correct; that is, a blank separates the operation field from the operand field, and that there is a comma between operands.

IFK035 ADDRESSABILITY ERROR
IFT035

Explanation: The address referred to does not fall within the range of a USING instruction.

Severity Code: 8

Programmer Response: Make sure the address referred to falls within the range of a USING instruction, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK036 (No message is assigned to this number)
IFT036 OPERAND FIELD MUST BE BLANK

Explanation: Operand found for an operation code which does not allow operands. (This message may be produced by the assembler if an operand is present in a COM or EJECT, statement when the operation field has been created by variable symbol substitution. Operands in these statements are not used but are not in error).

Severity Code: Variable

Programmer Response: Remove the illegal operand, if necessary, and reassemble.

IFK037 MNOTE STATEMENT
IFT037

Explanation: This indicates that an MNOTE statement has been generated from a macro definition. The text and severity code of the MNOTE statement will be found in line in the listing.

Severity Code: Variable

Programmer Response: Ensure that the error noted has been corrected, and reassemble.

IFK038 ENTRY ERROR
IFT038

Explanation: There might be more than 100 ENTRY operands in this program. A symbol in the operand of an ENTRY statement appears in more than one ENTRY statement; it is undefined; it is defined in a dummy section or in blank common; or it is, equated to a symbol defined by an EXTRN statement.

Severity Code: 8

Programmer Response: Ensure that all ENTRY operands are defined, not duplicated in another ENTRY statement.

IFK039 INVALID DELIMITER
IFT039

Explanation: This message can be caused by any syntax error; for example; missing delimiter, special character used which is not a valid delimiter, delimiter used illegally, operand missing, (that is, nothing between delimiters), unpaired parentheses, imbedded blank in expression.

Severity Code: 12

Programmer Response: Ensure that any of the conditions listed is corrected and reassemble.

IFK040 GENERATED RECORD TOO LONG
IFT040

Explanation: There are more than 236 characters in a generated statement (DOS - more thant 187 characters).

Severity Code: 12

Programmer Response: Ensure that there are no more than the maximum number of characters in a generated statement.

IFK041 UNDECLARED VARIABLE SYMBOL
IFT041

Explanation: A variable symbol is not declared in a defined SET symbol statement or in a macro prototype.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK042 SINGLE TERM LOGICAL EXPRESSION IS NOT A SETB SYMBOL
IFT042

Explanation: The single term logical expression has not been declared as SETB symbol. A single term logical explanation is valid only for a SETB symbol.

Severity Code: 8

Programmer Response: Make sure that the single term logical expression in question is declared as a SETB symbol. (See SETB under *Conditional Assembly Instructions,* in Chapter 5.)

IFK043 SET SYMBOL PREVIOUSLY DEFINED
IFT043

Explanation: A SET symbol has been previously defined.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK044 SET SYMBOL USAGE INCONSISTENT WITH DECLARATION
IFT044

Explanation: A SET symbol has been declared undimensioned, but is subscripted, or has been declared dimensioned, but is unsubscripted.

Severity Code: 8

Programmer Response: Ensure that SET symbol usage is consistent with SET symbol declarations. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK045 ILLEGAL SYMBOLIC PARAMETER
IFT045

Explanation: An attribute has been requested for a variable symbol which is not a legal symbolic parameter.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.

IFK046 AT LEAST ONE Y TYPE OR R TYPE CONSTANT IN ASSEMBLY
IFT046

Explanation: One or more relocatable Y-type or R-type constants in assembly; relocation may result in an address greater than two bytes in length.

Severity Code: 4

Programmer Response: Use an A-type constant if your program will be link edited above 64K bytes.

IFK047 SEQUENCE SYMBOL PREVIOUSLY DEFINED
IFT047

Explanation: Invalid use of sequence symbol. This error results from erroneously coding the same sequence symbol more than once in a single macro definition.

Severity Code: 12

Programmer Response: Ensure that there is no duplication of sequence symbols in a single macro definition. (See *Sequence Symbols* under *Conditional Assembly Instructions*, in Chapter 5.)

IFK048 SYMBOLIC PARAMETER PREVIOUSLY DEFINED OR SYSTEM VARIABLE SYMBOL DECLARED AS SYMBOLIC PARAMETER
IFT048

Explanation: A symbolic parameter has been previously defined, or a system variable symbol has been declared as a symbolic parameter.

Severity Code: 12

Programmer Response: See Variable Symbols under *Introduction* in Chapter 5, and Symbolic Parameters under *Macro Definitions,* also in Chapter 5. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.

IFK049 VARIABLE SYMBOL MATCHES A PARAMETER
IFT049

Explanation: A variable symbol is identical to a parameter resulting in a doubly defined symbol.

Severity Code: 12

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK050 INCONSISTENT GLOBAL DECLARATIONS
IFT050

Explanation: A global SET variable symbol (that is, defined in more than one macro definition, or in a macro definition and in the source program) is inconsistent in SET type or dimension.

Severity Code: 8

Programmer Response: Make sure all SET symbols, global or local, are consistent in type or dimension, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the

PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK051 MACRO DEFINITION PREVIOUSLY DEFINED
IFT051

Explanation: A prototype operation field is the same as a machine or assembler instruction or a previous prototype. This message is not produced when a programmer macro matches a system macro. The programmer macro will be assembled with no indication of the corresponding system macro.

Severity Code: 12

Programmer Response: Ensure that the programmer macros are not previously defined and also that the operation field of the macro prototype is not identical to a machine or assembler operand. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK052 NAME FIELD CONTAINS ILLEGAL SET SYMBOL
IFT052

Explanation: SET symbol in name field does not correspond to the SET statement type.

Severity Code: 8

Programmer Response: Ensure that SET symbols in the name fields correspond to SET statement types, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK053 GLOBAL DICTIONARY FULL
IFT053

Explanation: The global dictionary is full; assembly is terminated.

Severity Code: 12

Programmer Response: Probable user error. Do one or more of the following:

1. Split the assembly into two or more parts and assemble each separately.
2. Allocate more core for the assembler (OS--the global and local dictionaries, together, can occupy up to 64K).
3. (OS only) Specify a smaller SYSLIB blocksize. Thus, if BLKSIZE=3600, try BLKSIZE=1800, or BLKSIZE=1200. Reblock the library to the size chosen, and try the assembly again.

IFK054 LOCAL DICTIONARY FULL
IFT054

Explanation: The local dictionary is full; current macro is aborted or if the operation is in open code, assembly is terminated.

Severity Code: 12

Programmer Response: Probable user error. Do one or more of the following:

1. Split the assembly into two or more parts, and assemble each separately.
2. Allocate more core for the assembler (OS--the global and local dictionaries, together, can occupy up to 64K).
3. (OS only) Specify a smaller SYSLIB blocksize. Thus, if BLK(OS only) SIZE=3600, try BLKSIZE=1800 or BLKSIZE=1200. Reblock the library to the size chosen, and try the assembly again.

IFK055 INVALID ASSEMBLER OPTION(S) ON THE EXECUTE CARD
IFT055 (No message is assigned to this number.)

Explanation: An assembler option specified on the EXECUTE card is invalid (OS only).

Severity Code: 8

Programmer Response: Make sure all assembler options specified are correct and reassemble if necessary. If problem recurs, do the following before calling IBM:

- Make sure that MSGLEVEL=(1, 1) was specified in the JOB statement.
- Have the user source program, user macro definitions, and associated listings available. (See *Appendix F, Storage Requirements and Job Control Language.*)

IFK056 ARITHMETIC OVERFLOW
IFT056

Explanation: The intermediate or final result of an expression is not within the range of -2^{31} to $2^{31}-1$.

Severity Code: 8

Programmer Response: Ensure that the intermediate or final result of expression is within the range of -2^{31} to $2^{31}-1$.

IFK057 SUBSCRIPT EXCEEDS MAXIMUM DIMENSION
IFT057

Explanation: &SYSLIST or symbolic parameter subscript exceeds 200 (DOS - exceeds 100) or is negative or zero; or SET symbol subscript exceeds dimension.

Severity Code: 8

Programmer Response: Ensure that the &SYSLIST or symbolic parameter subscript does not exceed the maximum allowable number and that it is a positive number. (See *Extended Features of the Macro Language* in Chapter 5.)

IFK058 RE-ENTRANT CHECK FAILED
IFT058 (No message is assigned to this number.)

Explanation: An instruction has been detected which, when executed, might store data into a control section or a common area. This message is generated only when requested by control cards and it simply indicates a possible re-entrant error.

Severity Code: 4

Programmer Response: Ensure that the detected instruction does not store data in a control section or a common area.

Note: The DOS assembler does not check for reentry; therefore, there is no DOS message.

IFK059 UNDEFINED SEQUENCE SYMBOL
IFT059

Explanation: An operand sequence symbol does not appear as a sequence symbol in a name field.

Severity Code: 12

Programmer Response: Ensure that the operand sequence symbol in question appears in a name field. (See *Sequence Symbols* under *Conditional Assembly Instructions* in Chapter 5.)

IFK060 ILLEGAL ATTRIBUTE NOTATION
IFT060

Explanation: L was requested for a parameter whose type attribute does not allow these attributes to be requested.

Severity Code: 8

Programmer Response: Remove the L' request for the parameter in question and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available. (See *Attributes* under *Conditional Assembly Instructions* in Chapter 5.)

IFK061 ACTR COUNTER EXCEEDED
IFT061

Explanation: Conditional assembly loop counter has been exceeded; conditional assembly has been terminated.

Severity Code: 12

Programmer Response: Ensure that the number of AGO and AIF statements do not exceed the standard value of 4096 for OS 150 for DOS or the value assigned by you through the ACTR instruction. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available. (See *ACTR* under *Macro Definitions* in Chapter 5.)

IFK062 GENERATED STRING GREATER THAN 255 CHARACTERS
IFT062 GENERATED STRING GREATER THAN 127 CHARACTERS

Explanation: The maximum size character expression from which the character value can be chosen is 255 characters for OS; 127 for DOS.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK063 EXPRESSION 1 OF SUBSTRING IS ZERO OR MINUS
IFT063

Explanation: Expression 1 of the substring notation indicates the first character in the character expression that is to be assigned. It, therefore, must be a positive value.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available. (See *Substring Notation* under *SETC - Set Character,* in Chapter 5.)

IFK064 EXPRESSION 2 OF SUBSTRING IS ZERO OR MINUS
IFT064

Explanation: Expression 2 in substring notation indicates the number of consecutive characters in the character expression that are to be assigned to the SETC symbol. It, therefore, must have a positive value.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available. (See *Substring Notation* under *SETC - Set Character,* in Chapter 5.)

IFK065 INVALID OR ILLEGAL TERM IN ARITHMETIC EXPRESSICN
IFT065

Explanation: The value of a SETC symbol used in the arithmetic expression is not composed of decimal digits, or the parameter is not a self-defining term.

Severity Code: 8

Programmer Response: Ensure that the value of a SECT symbol used in the arithmetic expression is composed of decimal digits and that the parameter is a self-defining term. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK066 UNDEFINED OR DUPLICATE KEY WORD OPERAND OR EXCESSIVE POSITIONAL OPERANDS
IFT066

Explanation: The same keyword operand occurs more than once in the macro instruction; a keyword is not defined in a prototype statement; in a mixed mode macro instruction, more positional operands are specified than are specified in the prototype.

Severity Code: 12

Programmer Response: Ensure that there are no duplicate or underfined keyword operands, and that there are no more positional operands than are specified in the prototype.

IFK067 EXPRESSICN 1 CF SUBSTRING GREATER THAN LENGTH OF CHARACTER EXPRESSION
IFT067

Explanation: Expression 1 of the substring must not be greater than the length of the character expression to which it refers.

Severity Code: 8

Programmer Response: Ensure that expression 1 of the substring is not greater than the length of the character expression to which it refers. (See *Substring Notation* under *SETC - Set Character,* in Chapter 5.)

IFK068 GENERATICN TIME DICTIONARY AREA OVERFLOWED
IFT068

Explanation: Not enough storage allocated to the assembler; for OS (only), the blocksize is too large.

Severity Code: 12

Programmer Response: Probable user error. Do one or more of the following before calling IBM for programming support:

1. Split the assembly into two or more parts and assemble each separately.
2. Allocate more core to the assembler (the global and local dictionaries, together, can occupy up to 64K).
3. (For OS only) Specify a smaller SYSLIB blocksize. Thus, if BLKSIZE=3600, try BLKSIZE=1800 or BLKSIZE=1200, reblock the library to the size chosen, and try the assembly again.
4. Have the user source program, user macro definitions, and associated listings available.

IFK069 EXPRESSICN 2 CF SUBSTRING GREATER THAN 8 CHARACTERS
IFT069

Explanation: Expression 2 of substring must not be greater than 8.

Severity Code: 8

Programmer Response: Respecify the value of expression 2 to some value not greater than eight characters, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK070 (No message is assigned to this number.)
IFT070

IFK071 ILLEGAL OCCURRENCE CF LCL, GBL, OR ACTR STATEMENT
IFT071

Explanation: Local or global declaration; or the ACTR statement is not in proper place in the program.

Severity Code: 8

Programmer Response: Ensure that the local or global declaration or ACTR statement is in the proper place, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK072 ILLEGAL RANGE ON ISEQ STATEMENT
IFT072

Explanation: One or more columns to be sequenced-checked are between the "begin" and "end" columns of the statement.

Severity Code: 4

Programmer Response: Ensure that any column to be sequence-checked falls outside the range of the "begin" and "end" columns of the statement.

- Have the user source program, user macro definitions, and associated listings available.

IFK073 ILLEGAL NAME FIELD
IFT073

Explanation: Either a statement which requires a name has been written without a name; or a statement which has a name is not allowed to have a name; or a name entry required to be a sequence symbol is not a sequence symbol.

Severity Code: 8

Programmer Response: Ensure that statements requiring a name have one; that any statement having an illegal name be corrected by removing the name; and that any name required to be a sequence symbol is a sequence symbol.

IFK074 ILLEGAL STATEMENT IN COPY CODE OR SYSTEM MACRO
IFT074

Explanation: A statement being copied was a COPY, END, ICTL, ISEQ, MACRO, MEND, or a model statement in a macro containing an END, PRINT, COPY, ISEQ, ICTL.

Severity Code: 8

Programmer Response: Check statements to be copied to ensure that they are not illegal.

IFK075 ILLEGAL STATEMENT OUTSIDE OF A MACRO DEFINITION
IFT075

Explanation: A statement that is allowed only in a macro definition was encountered in OPEN code; for example, period asterisk (.*), MNOTE statement.

Severity Code: 8

Programmer Response: Ensure that statements that are allowed only in macro definitions are not used in OPEN code.

IFK076 SEQUENCE ERROR
IFT076

Explanation: A statement with a sequence number lower than the preceding statement was found when using the ISEQ instruction.

Severity Code: 12

Programmer Response: Ensure that all statements with sequence numbers after the ISEQ instruction are in proper sequence. (See *ISEQ,* in Chapter 4.)

IFK077 ILLEGAL CONTINUATION CARD
IFT077

Explanation: Either there are too many continuation cards; or there are non-blanks between the "begin" and "continue" columns on the continuation card; or a card not intended as a continuation was treated as such because of a punch in the continuation column of the preceding card.

Severity Code: 8

Programmer Response: Ensure that the rules for the use of continuation cards are observed:

1. A non-blank character must be in column 72.
2. A continuation card begins in column 16.
3. The limit on the number of continuation cards must be observed. (See *ICTL*, in Chapter 4.)

IFK078 (No message is assigned to this number.)
IFT078 MACRO MNEMONIC OP-CODE TABLE OVERFLOW

Explanation: Not enough storage has been allocated to the assembler; or there is an unusually large number of macro mnemonic op-codes, causing the table to overflow. (See Appendix F, Storage Requirements and Job Control Language.)

Severity Code: 12

Programmer Response: Probable user error. Do one or more of the following:

1. Split the assembly into two or more parts and assemble each separately.
2. Allocate more core to the assembler.

IFK079 ILLEGAL STATEMENT IN MACRO DEFINITION
IFT079

Explanation: This operation is not allowed within a macro definition.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement. For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK080 ILLEGAL START CARD
IFT080

Explanation: Statements affecting, or depending upon, the location counter have been encountered before a START statement.

Severity Code: 8

Programmer Response: Ensure that there is no statement affecting, or depending upon, the location counter before a START statement. (See *START* in Chapter 4.) For DOS--execute the DOS SSERV program for a copy of the book specified in the COPY statement.

IFK081 ILLEGAL FORMAT IN GBL OR LCL STATEMENTS
IFT081

Explanation: An operand is not a variable symbol.

Severity Code: 8

Programmer Response: Ensure that the format in GBL or LCL statements is correct; that is, that all operands are variable symbols.

IFK082 ILLEGAL DIMENSION SPECIFICATION IN GBL OR LCL STATEMENT
IFT082

Explanation: Dimension is other than 1 to 2500.

Severity Code: 8

Programmer Response: Ensure that the dimension specification in each global or local statement is within the range of 1 to 2500 for OS, 1 to 255 for DOS.

IFK083 SET STATEMENT NAME FIELD NOT A VARIABLE SYMBOL
IFT083

Explanation: The name field in a SET statement is not a variable symbol.

Severity Code: 8

Programmer Response: Ensure that the name field in the SET statement is a variable symbol.

IFK084 ILLEGAL OPERAND FIELD FORMAT IN CONDITIONAL ASSEMBLY STATEMENT
IFT084

Explanation: Syntax is invalid (for example; AIF statement operand does not start with a left parenthesis): operand of AGO is not a sequence symbol; operand of PUNCH, TITLE, MNOTE is not enclosed in quotes.

Severity Code: 8

Programmer Response: Ensure that the syntax in conditional assembly statements is valid. The preceding *explanation* gives examples.

IFK085 INVALID SYNTAX IN EXPRESSION
IFT085

Explanation: Invalid delimiter; too many terms in the expression; too many levels of parentheses; two operators in succession; two terms in succession; or illegal character.

Severity Code: 8

Programmer Response: Ensure that the syntax in expression is valid. The preceding *explanation* gives examples.

IFK086 ILLEGAL USAGE OF SYSTEM VARIABLE SYMBOL
IFT086

Explanation: A system variable symbol appears in the name field of a SET statement, is used in a mixed mode or keyword macro definition, is declared in a GBL or LCL statement, or is an unsubscripted &SYSLIST in a context other than N'&SYSLIST.

Severity Code: 4

Programmer Response: Ensure that system variable symbols do not appear illegally. The preceding *explanation* gives some examples.

IFK087 NO ENDING APOSTROPHE
IFT087

Explanation: There is an unpaired apostrophe or ampersand in the statement.

Severity Code: 8

Programmer Response: Ensure that each apostrophe or ampersand is paired, where necessary.

IFK088 UNDEFINED OPERATION CODE
IFT088

Explanation: A symbol in the operation code field does not correspond to a valid machine or assembler operation code or to any operation code in a macro prototype statement.

Severity Code: 12

Programmer Response: Ensure that the proper operation codes are used; in every instance.

IFK089 INVALID ATTRIBUTE NOTATION
IFT089

Explanation: Syntax error inside a macro definition; for example, the argument of the attribute reference is not a symbolic parameter.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source, program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement for DOS, execute the DOS SSERV program to obtain a copy of the book specified in the COPY statement.

IFK090 INVALID SUBSCRIPT
IFT090

Explanation: Syntax error (for example; double subscript where a single subscript is required, or vice versa; there is no right parenthesis after subscript).

Severity Code: 8

Programmer Response: Ensure that the syntax of subscripts used is correct. The preceding *explanation* gives examples.

IFK091 INVALID SELF-DEFINING TERM
IFT091

Explanation: Value is too large or is inconsistent with the data type; that is, one byte of immediate data is greater than X'FF'.

Severity Code: 8

Programmer Response: Ensure that the value is consistent with the data type.

IFK092 INVALID FORMAT FOR VARIABLE SYMBOL
IFT092

Explanation: The first character after the ampersand is not alphabetic; or the variable symbol contains more than eight characters, or a double ampersand was not used in a TITLE card or a character self-defining term.

Severity Code: 8

Programmer Response: Ensure that the format for variable symbols is correct; for example, that there are no more than eight characters and that the first character after the ampersand is alphabetic.

IFK093 UNBALANCED PARENTHESIS OR EXCESSIVE LEFT PARENTHESES
IFT093

Explanation: End of statement encountered before all parenthesis levels are satisfied; may be caused by an imbedded blank or other unexpected terminator, or by failure to have a punch in the continuation column.

Severity Code: 8

Programmer Response: Ensure that there is both a left and a right parenthesis. Some examples of unbalanced parentheses are provided in the preceding explanation.

IFK094 INVALID OR ILLEGAL NAME OR OPERATION IN PROTOTYPE STATEMENT
IFT094

Explanation: Name is not blank or is not a variable symbol, or variable symbol in name field is subscripted, or there is a violation of rules for forming a variable symbol (must begin with an ampersand (&) and be followed by from one to seven letters and/or numbers, the first of which must be a letter); or statement following the MACRO statement is not a valid prototype statement.

Severity Code: 12

Programmer Response: Ensure that the name or operation in the prototype statement is legal and valid.

IFK095 ENTRY TABLE OVERFLOW
IFT095

Explanation: Number of ENTRY symbols (that is, ENTRY instruction operands) exceeds 100.

Severity Code: 8

Programmer Response: Make sure that the number of ENTRY symbols does not exceed 100.

IFK096 MACRO INSTRUCTICN OR PROTO TYPE OPERAND EXCEEDS 255 CHARACTERS
IFI096 MACRO INSTRUCTION OR PROTO TYPE OPERAND EXCEEDS 127 CHARACTERS

Explanation: Macro instruction or prototype operand exceeds the maximum length allowed: 255 for OS or 127 for DOS.

Severity Code: 12

Programmer Response: Ensure that the macro instruction or prototype operand does not exceed the maximum number of characters allowable.

IFK097 INVALID FORMAT IN MACRO INSTRUCTION OPERAND OR PROTOTYPE PARAMETER
IFT097

Explanation: This message can be caused by:

1. Illegal "=".
2. A single "&" appears somewhere in the standard value assigned to a prototype keyword parameter.
3. First character of a prototype parameter is not "&".
4. Prototype parameter is a subscripted variable symbol.
5. Invalid use of alternate format in prototoye statement; for example:

```
10        16          72
PROTO     &A, &B,
                   or
PROTO     &A, &B      X
               &C
```

6. Unintelligible prototype parameter; for example, "&A*" or "&A&&".
7. Illegal (non-assembler) character appears in prototype parameter or macro instruction operand.

Severity Code: 12

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. The preceding *explanation* gives some examples. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.

IFK098 EXCESSIVE NUMBER OF OPERANDS OR PARAMETERS
IFT098

Explanation: Either the prototype has more than 200 parameters, (DOS more than 100 parameters) or the macro instruction has more than 200 operands (DOS more than 100 operands).

Severity Code: 12

Programmer Response: Ensure that the prototype contains no more than 200 parameters for OS (100 for DOS), or that the macro instruction contains no more than 200 operands for OS (100 for DOS).

IFK099 POSITIONAL MACRO INSTRUC TION OPERAND, PROTOTYPE PARAMETER, EXTRA COMMA FCLLOWS KEYWORD
IFT099

Explanation: A keyword macro has been improperly coded.

Severity Code: 12

Programmer Response: Ensure that the proper operand is used after a keyword. Y-type address constant 47.

IFK100 STATEMENT CCMPLEXITY EXCEEDED
IFT100

Explanation: For OS, more than 50 operands in an assembler instruction (32 for DC and DS statements) or more than 50 terms in a statement; for DOS, more than 35 operands in an assembler instruction (1 for DC and 1 for DS) or more than 50 terms in a statement.

Severity Code: 8

Programmer Response: Ensure that the complexity of each statement is not exceeded.

IFK101 EOD ON SYSIN
IFT101 EOD ON SYSIN CR SYSIPT

Explanation: EOD before END card.

Severity Code: 12

Programmer Response: Ensure that there is an END card in the deck. Make sure /* does not precede the END card.

IFK102 INVALID OR ILLEGAL ICTL
IFT102

Explanation: The operands of the ICTL are out of range, or the ICTL is not the first statement in the input deck.

Severity Code: 16

Programmer Response: Ensure that the ICTL is the first statement in the input deck and that the operands are in the proper range. (See *ICTL,* in Chapter 4.)

IFK103 ILLEGAL NAME IN OPERAND FIELD OF COPY CARD
IFT103

Explanation: Syntax error; for example, symbol has more than eight characters or has an illegal character.

Severity Code: 12

Programmer Response: Ensure that the operand of the copy statement conforms to the rules for names. Probable user error.

IFK104 COPY CODE NOT FOUND
IFT104

Explanation: The operand of a COPY statement specified COPY text which cannot be found in the library.

Severity Code: 12

Programmer Response: Ensure that the correct name was used for COPY text in the library.
Also ensure that the COPY code really exists in the library if the correct name was specified. Probable user error.

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.

IFK105 EOD ON SYSTEM MACRO LIBRARY
IFT105 EOD ON SOURCE STATEMENT LIBRARY

Explanation: EOD before MEND card; MEND statement missing from macro definition: COPY code not found while editing a macro; macro definition truncated; or EOF encountered while reading a macro or copy code.

Severity Code: 12

Programmer Response: Probable user error. Make sure source is correct, and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.

IFK106 (No message is assigned to this number.)
IFT106

IFK107 INVALID OPERAND
IFT107

Explanation: Invalid syntax in DC operand (for example; invalid hexadecimal character in hexadecimal DC); operand string too long for X, B, C, DCs; operand unrecognizable (contains invalid value, or incorrectly specified).

Severity Code: 4

Programmer Response: Make sure that syntax in the DC operand is correct. The preceding *explanation* gives good examples of what may be incorrect. (See *DC - Define Constant,* in Chapter 4.)

IFK108 PREMATURE EOD
IFT108

Explanation: Indicates an internal assembler error or a machine error.

Severity Code: 16

Programmer Response: Reassemble; if the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL= (1, 1) was specified in the JOB statement.

IFK109 (No message is assigned to this number.)
IFT109

IFK110 EXPRESSICN VALUE TOC LARGE
IFT110

Explanation: Value of expression is greater than 262,143. Expressions in EQU and ORG statements are flagged if (1) they include terms previously defined as negative values, or (2) positive terms give a result of more than 18 bits in magnitude.

Severity Code: 8

Programmer Response: Probable user error. Make sure source is correct and reassemble if necessary. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.

IFK111 SYSGO DD CARD MISSING NCLCAD OPTION USED

Explanation: DD statement for SYSGO is incorrect or missing; NOLOAD option is taken.

Severity Code: 16

Programmer Response: Probable user error. If necessary, supply the missing DD statement or make sure that the information on the DD statement is correct and reassemble. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL= (1, 1) was specified in the JOB statement.

IFT111I ABORT--UNEXPECTED EOF ON SYSxxx

Explanation: EOF has occured on an assembler work file that does not support multivolume files. It usually results from a short tape, or from reading a tape reflective marker. This is probably user error.

System Response: The job step is terminated.

Programmer Response: If the problem recurs, have the system log, printer output, and the job stress available to complete your problem determination action.

Operator Response: (1) If SYSxxx is assigned to a tape, mount a longer tape or use a 1600 BPI tape drive instead of an 800 BPI drive, or (2) reassign the work files to disk and rerun the job, or (3) if SYSxxx is assigned to a disk, submit larger extents and rerun the job.

IFK112 SYSPUNCH DD CARD MISSING NODECK OPTICN USED

Explanation: DD statement for SYSPUNCH is incorrect or missing; NODECR option is taken.

Severity Code: 16

Programmer Response: Probable user error. If necessary, supply the missing DD statement or make sure that information on DD statement is correct and reassemble. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL=(1, 1) was specified in the JOB statement.

IFK113 INVALID BYTE SELECTION IFT113

Explanation: Byte specification is not an absolute expression of value 0 or 1.

Programmer Response: Make sure that byte selection is an absolute expression of value of 0 or 1.

Operator Response: Issue the LISTIO command to check the assignments, and enter the correct work file assignments if possible.

IFT114I ABORT--NO UNIT ASSIGNED FOR SYSPCH

Explanation: The OPTION [DECK] is in effect and SYSPCH is not assigned. This is probably a user error.

System Response: The job step is terminated.

Programmer Response: Submit an assignment for SYSPCH, or specify OPTION [NODECK] and resubmit the job.
If the problem recurs, do the following to complete your problem determination action:

1. Retain the LISTIO listing.
2. Have the job stream, program listing, and system log available.

Operator Response: Execute the LISTIO command and verify assignments. Submit an assign for SYSPCH and rerun the job.

IFT115 ABORT-PERMANENT I/O ERROR ON SYSxxx

Explanation: An unrecoverable error on the named file prevents further processing: if the named file is SYSxxx, the unit code of the DTF that caused the error does not match any valid unit. This is usually the result of an accidental overlap that destroys the DTF. This is probably a hardware error.

System Response: The job step is terminated.

Programmer Response: Rerun the job, using another disk pack or tape reel, or use another unit for the disk pack or tape reel.
If the problem recurs, do the following to complete your problem determination action:

1. Execute the ROD command and EREP, and retain the output.
2. Have the job stream and system log available.

Operator Response: Execute the LISTIO command for SYSxxx to determine the physical unit to which it is

assigned. Move the disk pack or tape reel to another physical device and reassign SYSxxx to that unit, or mount another disk pack or tape reel and rerun the job.

IFT115I ABORT--INVALID DUAL ASSGN SYSPCH-SYSIPT [SYSLST]

Explanation: SYSPCH and SYSIPT are both assigned to the same unit, which is not a 1442N 1 or 1520B 1 card reader, or SYSPCH and SYSLST are both assigned to the same unit, which is not a disk. This is probably a user error.

System Response: The job step is terminated.

Programmer Response: Check the LISTIO listing to determine the dual assignments. Reassign the indicated logical units to separate devices or, the required device type.
If the problem recurs, retain the LISTIO output, the job stream, system log, and supervisor listing to complete your problem determination actions.

Operator Response: Execute LISTIO to determine the current assignments. Reassign the two indicated logical units to separate devices or to the required device type.

IFT116I ABORT--INVALID PHYSICAL UNIT FOR SYSxxx

Explanation: The assignment for a work file(s) are not valid:

- The device type is not valid, or the assembler is link edited for devices different from those assigned.
- The UA (unassign) or IGN (ignore) option was specified for the assembler.
- The specified mode setting is not valid.
- For the assembler, the work file device types are not consistent. (SYS003 is correct.)

Only the first invalid unit is named in the message. This is probably a user error.

System Response: The job step is terminated.

Programmer Response: Use the LISTIO output to determine the cause for the message. Use CSERV to display the phase named "ASSEMBLY" and check byte X'1C', bits 5, 6, and 7 for the device type specified at link-edit time as work files.

Bit 5: 1=2400
Bit 6: 1=2314
Bit 7: 1-2111 Correct the assignment and resubmit the job.
If the problem recurs, do the following to complete your problem determination action:

1. Have the LISTIO and CSERV output available.
1. Have the job stream and system output available.

Operator Response: Issue the LISTIO command to check the assignments and enter the correct work file assignments if possible.

IFK116 (No message is assigned to this number.)

IFK117 (No message is assigned to this number.)
IFT117

IFK118 (No message is assigned to this number.)
IFT118

IFK119 ILLEGAL EXTERNAL REGISTER
IFT119

Explanation: External register specification is not an absolute expression from 0 to 127.

Programmer Response: Respecify the register, using an absolute expression from 0 to 127.

IFT120 INVALID BIT SELECTION
IFK120

Explanation: Bit specification is not an absolute expression from 0 to 7.

Programmer Response: Respecify the bit selection using bits starting with 0 through 7.

IFT121 INVALID USE OF SYMBOLIC REGISTER
IFK121

Explanation: A symbolic register expression is specified where an absolute, relocatable, or complex relocatable expression is required, or a symbolic register expression appears in a multiterm expression.

Programmer Response: Replace the invalidly specified symbolic register expression with the appropriate absolute, relocatable or complex relocatable expression required for reassemble. See EQUR in Chapter 4 for a discussion of symbolic registers.

IFK997 SYSPRINT DD CARD MISSING NOLIST OPTION USED
IFT997I (No message is assigned to this number.)

Explanation: DD statement for SYSPRINT is incorrect or missing; NOLIST option taken.

System Response: Printed on console typewriter.

Severity Code: 0

Programmer Response: Probable user error. If necessary, supply the missing DD statement or make sure

that information on the DD statement is correct; reassemble. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL=(1, 1) was specified in the JOB statement.

```
IFK998 ASSEMBLY TERMINATED
       MISSING DATA SET FOR
       (dd name)
IFT998I (No message is assigned
        to this number.)
```

Explanation: DD statement(s) for data set(s), SYSIN, SYSUT1, SYSUT2, SYSUT3, and/or SYSPRINT is incorrect or missing.

System Response: Printed on SYSPRINT, if possible; otherwise, on the console typewriter.

Severity Code: 20

Programmer Response: Probable user error. Supply the missing DD statement(s) or make sure that information on DD statement(s) is correct; reassemble. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL= (1,1) was specified in the JOB statement.

```
IFK999   ASSEMBLY TERMINATED,
         JOBNAME, STEPNAME,  UNIT
         ADDRESS,  DEVICE
         DDNAME, OPERATION
         ATTEMPTED, ERROR
         DESCRIPTION.
IFT999I  (No message is assigned
         to this number.)
```

Explanation: Indicates a permanent I/O error. This message is produced by the SYNADAF macro instruction.

System Response: Printed on SYSPRINT, if possible; otherwise, on the console typewriter.

Severity Code: 20

Programmer Response: Reassemble. If the problem recurs, do the following before calling IBM:

- Have the user source program, user macro definitions, and associated listings available.
- If the COPY statement was used, execute the OS IEBPTPCH utility program to obtain a copy of the PDS member specified in the COPY statement.
- Make sure that MSGLEVEL= (1,1) was specified in the JOB statement.

Index

READER'S COMMENT FORM

IBM 3705
Communications Controller
Assembler Language

Order No. GC30-3003-0

- How did you use this publication?

  As a reference source ☐
  As a classroom text ☐
  As . . . . ☐

- Based on your own experience, rate this publication . . .

As a reference source:	. . . . . Very Good	. . . . Good	. . . . . Fair	. . . . . Poor	. . . . . Very Poor
As a text:	. . . . . Very Good	. . . . Good	. . . . . Fair	. . . . . Poor	. . . . . Very Poor

- What is your occupation? . . . .

- We would appreciate your other comments; please give specific page and line references where appropriate. If you wish a reply, be sure to include your name and address.

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

YOUR COMMENTS, PLEASE . . .

Your answers to the questions on the back of this form, together with your comments, help us produce better publications for your use. Each reply is carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in using your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

Cut Along Line

Fold Fold

FIRST CLASS
PERMIT NO. 569
RESEARCH TRIANGLE PARK
NORTH CAROLINA

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

POSTAGE WILL BE PAID BY . . .

IBM Corporation
P. O. Box 12275
Research Triangle Park
North Carolina 27709

Attention: Publications Center, Dept. E01

Fold Fold

IBM 3705 Communications Controller Assembler Language Printed in U.S.A. GC30-3003-0

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GC30-3003-0

IBM 3705 Communications Controller Assembler Language Printed in U.S.A. GC30-3003-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)