GC30-3003-2
File No. S360/S370-30

Systems

IBM 3704 and 3705 Communications Controllers Assembler Language

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. This language is similar to the language associated with the operating system assemblers (OS, DOS, OS/VS, DOS/VS) upon which the communications controller assembler is based.

Chapter 1 introduces the assembler language and describes the major differences between the language and the operating system 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 conditional assembly language.

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 Appendixes G, H, and I contain messages and codes helpful in debugging a source program.

Before using this publication, the reader should be familiar with basic programming concepts and techniques. The prerequisite publication is *Introduction to the IBM 3704 and 3705 Communications Controllers,* GA27-3051. Corequisite to this publication is the *IBM 3704 and 3705 Communications Controllers Principles of Operation,* GC30-3004.

The contents of this publication apply to OS, OS/VS, DOS *and* DOS/VS *users except as noted in the text.*

Third Edition (May 1975)

This is a major revision of, and obsoletes, GC30-3003-1. Refer to the Summary of Amendments in this manual for a list of changes.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 Bibliography (GC20-0001) and associated technical newsletters for the editions that are applicable and current.

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

This manual has been prepared by the IBM System Communications Division, Publications Center, Department E01, P.O. Box 12195, 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.

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 communications controller 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 OS, DOS, OS/VS, and DOS/VS assemblers referred to collectively in this book as *operating system assemblers.* Some of the major differences between the communications controller assembler and the operating system assemblers are:

- no literals are permitted.
- no floating point arithmetic instructions are permitted.
- new machine operation codes are provided.

(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 or loader component of the 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 contains 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. Chapter 3 gives the machine instruction mnemonic codes, examples of machine instructions for each instruction format, and a list of extended mnemonics for certain Branch and certain Store instructions. Appendix B lists all of the machine instructions and gives the format code, mnemonic, and operand format for each.

Auxiliary Functions and Programmer Aids

The assembler language contains mnemonic assembler instruction operation codes by which you may 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 for a detailed description and Appendix D for a summary.)

In addition, the instructions to the assembler provide 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 *Addressing Between Source Modules: Symbolic Linkage* 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 Format and Output* 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 Appendixes G, H, and I 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 include: (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 start-stop and/or BSC stations. 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 operating system assembler languages. For a review of these conventions, see *OS Assembler Language,* GC28-6514 or *OS/VS-DOS/VS-VM/370 Assembler Language* (GC33-4010).

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.

An *operation entry* is:

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

An *operand entry* is:

- One or more operands, each comprising one or more expressions 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 macro 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 appears in *Chapter 5: The IBM Communications Controller Assembler Macro Facility.*
- *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 Assembler Macro Facility.*

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 left-most 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) Instruction,* 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 verifies that the value of a symbol is not negative and not larger than 2^{18}-1. However, for 3704s and models of the 3705, without extended addressing, 2^{18}-1 exceeds the addressable storage range. The difference between the limit of storage and the maximum address allowable in the register (2^{18}-1) is an area which will cause an addressing exception. See *Introduction to the IBM 3704 and 3705 Communications Controllers,* GA27-3051 for a discussion of models and storage capacities by model. For a discussion of extended addressing, storage addressing, and address exception, see *IBM 3704 and 3705 Communications Controllers Principles of Operation,* GC30-3004.

A symbol is said to be defined when it appears as the name of a source statement.

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 EQU instruction to location counter value (EQU*) or to a self-defining term, the length attribute of the symbol is 1. These and other exceptions are noted under the 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, occurs 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); (Note that this limit is lower than that imposed by the operating system assemblers.) Some examples of decimal self-defining terms are: 8, 147, 4092, and 00021.

Note: For the 3704 and models of the 3705 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 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 for machines 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 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 ampersands 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 two ampersands—&&—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 counter is first adjusted to the proper boundary for the item, if adjustment is necessary, and then incremented by the length of the assembled item. Thus, it always points to the next available 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 asterisk 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 rules for coding expressions are:

1. An expression cannot start with an arithmetic operator (+-/*); therefore, the expression -A+BETA is invalid, but the expression 0-A+BETA is valid.
2. An expression must not contain two terms in succession (*Invalid:* 15B'101')
3. No blanks are allowed between an operator and a term nor between two successive operators.
4. An expression can contain up to:

   16 terms, 15 operators (unary and binary), 5 levels of parentheses (for OS, DOS, and DOS/VS)

   20 terms, 19 operators (unary and binary), 6 levels of parentheses (for OS/VS)

   (Parentheses that are part of an operand specification do not count toward this limit)
5. A single relocatable term is not allowed in a multiply or divide operation. (Paired relocatable terms have absolute values and can be multiplied and divided if they are enclosed in parentheses.)

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
```

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, for machines with extended addressing, or to 16 bits, for machines 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 result.

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 $2^{18}-1$ ($2^{16}-1$ for machines without extended addressing) although intermediate results may lie within the 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 above under *Absolute Expressions.*
- When using the communications controller assembler under OS or DOS, the unpaired term must not be directly preceded by a minus sign; -Y+X-Z is invalid. (This restriction does not apply when assembling under OS/VS or DOS/VS.)
- 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          W–X+W
  location counter)     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 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 *EQUR (Equate Symbol to Register Expression) Instruction* in Chapter 4.

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	N–And
B–Branch	O–OR
C–Compare	S–Subtract
I–Insert	T–Test
L–Load	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 3-10, Extended Mnemonics.

When assembling under OS/VS, an error in a machine instruction generally causes the instruction field to be replaced by zeros.

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.)

Figure 3-1 explains the symbols used in the assembler operand field formats that appear in Figures 3-2 through 3-9.

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

Operand Field	*Meaning*
A	A relocatable or absolute expression whose value may be from 0 to 2^{16} -1 (controllers without extended addressing) or from 0 to 2^{18} -1 (controllers with extended addressing).
B	An absolute expression specifying a base register; valid register numbers are 0 through 7.
D	An absolute expression specifying a displacement; valid range is 0-127. *Note:* Displacement for LH and STH instructions must be a multiple of 2; displacement for L and ST instructions must be a multiple of 4.
E	An absolute expression specifying an external register; valid range is 0-127.
I	An absolute expression specifying immediate data. Value of expression: 0-255.
M	An absolute expression specifying a bit of the byte specified by N. Value of expression: 0-7.
N, N1, N2	Absolute expressions specifying a byte. Value of expression: 0 or 1. *0* indicates the high-order or leftmost byte; *1* indicates the low-order or rightmost byte. *Note:* For ACR, SCR, ARI, SRI, and BCT instructions, a value of 1 for N1 or N implies both bytes 0 and 1 rather than just the rightmost byte.
Q, Q1, Q2	Symbolic register expressions that specify a register-byte combination. (See EQUR Instruction in Chapter 4.)
R, R1, R2	Absolute expressions specifying general registers; valid register numbers are 0 through 7. (Only the odd-numbered registers are valid for instructions that allow byte selection.)
S	An absolute or relocatable expression specifying an implied address used with a USING instruction. The assembler selects a proper base and displacement based on the symbol value and the USING information.
T	A relocatable expression specifying 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 that of the instruction which refers to it as an operand; that is, both must be associated with the same control section.

Figure 3-1. Meanings of Instruction Operand Fields

RR Format

The RR instruction format (Figure 3-2) denotes a register-to-register operation.

Basic Machine Format	*Assembler Operand Field Format*	*Applicable Instructions*
RR	{R1(N1) / Q1} , {R2(N2) / 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 3-2. Register-to-Register (RR) Format

Examples of RR Instructions:

```
ALPHA1   LHR    1,2
ALPHA2   LHR    REG1,REG2
BETA1    CR     3,5
BETA2    CR     THREE,FIVE
GAMMA    ACR    3(0),5(1)
GAMMA    ACR    HITHREE,LOFIVE
```

The operands of ALPHA1, BETA1, and GAMMA1 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 (Figure 3-3) denotes a register-to-storage operation.

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

Figure 3-3. Register-to-Storage (RS) Format

Note: Register 0 implies direct addressable storage when used as a base register for RS-format instructions (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 zeros is stored.

Examples of RS-Format Instructions:

```
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)
```

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 (Figure 3-4) denotes a register-to-storage with additional operation instruction.

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

Figure 3-4. Register-to-Storage with Additional Operation (RSA) Format

Examples of RSA-Format Instructions:

```
ALPHA     ICT     3(0),6
BETA      ICT     HITHREE,SIX
GAMMA     STCT    3(0),SIX
DELTA     STCT    HITHREE,FIVE
```

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

RT Format

The RT instruction format (Figure 3-5) denotes a branch operation.

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

Figure 3-5. Branch Operation (RT) Format

Examples of RT-Format Instructions:

```
ALPHA     BB      3(0,6),ADDR
ALPHA1    BCT     CTR(1),ADDR1
GAMMA     BZL     ADDR3
GAMMA1    BB      LOFIVE(4),ADDR
```

In ALPHA1, CTR 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 (Figure 3-6) denotes a register-to-immediate operand operation.

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

Figure 3-6. Register to Immediate Operand (RI) Format

Examples of RI-Format Instructions:

```
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
```

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 (Figure 3-7) denotes a register-to-immediate address operation.

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

Figure 3-7. Register to Immediate Address (RA) Format

Examples of RA-Format Instructions:

```
ALPHA1    LA      3,1000
ALPHA2    LA      3,ADDR1
BETA1     BAL     4,X'240'
BETA2     BAL     4,ADDR2
```

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

RE Format

The RE instruction format (Figure 3-8) denotes a register-to-external register operation. An external register is a register in the communications controller that the resident control program must access through input and output instructions. (See *External Registers* in *IBM 3704 and 3705 Communications Controllers Principles of Operation* (GC30-3004).)

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

Figure 3-8. Register to External Register (RE) Format

Examples of RE-Format Instructions:

```
ALPHA1    IN     2,10
ALPHA2    IN     REG2,EXTREG10
BETA1     OUT    2,X'3F'
BETA2     OUT    REG2,EXTREG96
```

In the examples, 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 (Figure 3-9) denotes an exit from the active program level.

Note: When assembling under OS/VS, any operands coded in this instruction are treated as comments (not flagged as errors).

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

Figure 3-9. Exit Format

See *Chapter 4: Instruction Set*, in *IBM 3704 and 3705 Communications Controllers Principles of Operation* (GC30-3004).

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 3-10.

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 3-10. 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 machine 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.

PROGRAM SECTIONING

You may write a program for the communications controller as a single source module or you may divide it into two or more source modules. Each module is assembled into a separate object module. These object modules are then combined by the linkage editor into a load module that constitutes the executable program.

A source module may comprise one or more control sections. Each control section is assembled as part of an object module. By writing the proper linkage editor control statements, you can select an entire object module or any of its individual control sections for inclusion in the load module.

Each control section should not exceed 4096 bytes (the largest sequence of source statements that can be accommodated by one base register).

The total number of control sections, dummy sections, and set symbols in a source module must not exceed 255.

Communication Between Parts of a Program

When writing a program, you must arrange for proper communication (1) between control sections within the same source module, and (2) between different source modules. This communication is described under *Addressing*, later in this chapter.

The Source Module

A source module consists of a sequence of source statements in the assembler language. You may include these source statements in the source module in two ways.

1. Write them on a coding form and enter them as input through a terminal or (in punched card form) through a card reader.
2. Write one or more COPY instructions among the source statements being entered. Upon encountering a COPY instruction, the assembler replaces it with a predetermined set of source statements from a library on which the set has previously been placed. These statements then become part of the source module just as if they had been individually entered as in (1) above.

The Beginning of a Source Module

The first statement of a source module can be any assembler language statement, except MEXIT or MEND, that is described in this manual. You can initiate the first control section of a source module by using the START instruction. However, you can—or must—write some source statements before the beginning of the first control section, as described under *Defining a Control Section,* later in this chapter.

Note: No R-type address constant can be assembled in the first two bytes of any control section (CSECT).

The End of a Source Module

A source module assembled under DOS or DOS/VS ends with a single END instruction. Any END instructions following the first one are ignored.

A source module assembled under OS or OS/VS ordinarily ends with a single END instruction. However, you can code several END instructions and use conditional assembly instructions (described in Chapter 5) to determine which one of the END instructions the assembler is to process.

COPY (Copy Predefined Source Code) Instruction

The COPY instruction obtains source-language code from a library and includes it in the program currently being assembled. See Figure 4-1 for the COPY instruction format.

Name	*Operation*	*Operand*
blank	COPY	one symbol

Figure 4-1. COPY Instruction

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 code immediately after the COPY instruction is encountered. The requested code may not contain any COPY, END, ICTL, ISEQ, MACRO, or MEND instructions. (DOS/VS: COPY, MACRO, and MEND are allowed inside the copied source code.)

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

END (End Assembly) Instruction

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 4-2 for the END statement format.

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

Figure 4-2. END Instruction

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 instruction has been read. They will therefore be flagged after the END instruction. If the programmer does not know which 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 libary as system macro definitions.

Defining a Control Section

A control section is the smallest subdivision of a program that can be relocated as a unit. The assembled control sections contain the object code for machine instructions, data constants, and storage areas.

A control section is a block of code that can be relocated, independently of other control sections, when the program is loaded without altering or impairing the operating logic of the program.

The beginning of a control section is normally identified by a CSECT instruction. However, if you wish to specify a tentative starting location, you may use the START instruction to specify the start address of the first control section of the source module. If you use neither a CSECT nor a START instruction to begin a control section, the entire source module is considered to be a single, unnamed control section.

Types of Control Sections

A control section is either an *executable* or a *reference* control section. An executable control section contains machine instructions and begins with a START or CSECT instruction that names the control section. (Alternatively, the START or CSECT instruction may be omitted, resulting in an unnamed control section that begins with the first machine instruction. However, you should name the control section with a CSECT or START instruction so that you can refer to it symbolically from other control sections within the same source module or from other source modules.)

A reference control section is one you initiate by a DSECT, COM, or (OS/VS only) DXD instruction and is not assembled into object code. You can use a reference control section to reserve storage areas or to describe data to which you can refer by executable control sections. These reference control sections are considered to be empty at assembly time; the actual binary data to which they refer is not entered until program execution.

Note: No R-type address constant can be assembled in the first two bytes of any control section (CSECT).

Control Section Location Counter

The assembler maintains a separate location counter for each control section in a source module being assembled. The location counter setting for a control section starts at 0. The location values assigned to the instructions and other data in a control section are therefore relative to the location counter setting at the beginning of that control section.

However, for executable control sections assembled under OS or OS/VS, the location values that appear in the assembly listings do not restart at 0 for each subsequent control section. They continue from the end of the previous control section. For executable control sections assembled under DOS or DOS/VS, the location values in the assembly listings always begin at 0, except location settings initiated by a START instruction with a non-zero operand.

For reference control sections, the location values in the listings always start with location 0.

First Executable Control Section

Any instruction that affects the location counter or uses its current value establishes the beginning of the first executable control section. These instructions are:

Any machine instruction

CW	CSECT	DROP	EQU	START
CNOP	CXD	DS	EQUR	USING
COPY*	DC	END	ORG	

*statements copied by this instruction determine whether COPY will begin the control section.

Note: The DSECT, COM, and (OS/VS only) DXD instruction begin reference control sections and do not establish the first executable control section.

Optional Instructions That Must Precede First Control Section

The following instructions, if used, must appear at the beginning of the source module, preceding the first control section:

- The ICTL instruction (must be the first instruction in the source module, if used). (An invalid ICTL statement [under OS/VS only] will cause all subsequent statements to be printed and interpreted as comments.)
- The OPSYN instruction (OS/VS only) (must precede any source macro definitions)
- Any source macro definitions
- The COPY instruction, if the code to be copied contains only OPSYN instructions or complete macro definitions. Any instructions copied by a COPY instruction or generated by the processing of a macro instruction before the first control section must belong exclusively to one of the following groups of instructions:

  COPY, DXD, EJECT, ENTRY, EXTRN, ISEQ, PRINT, PUNCH, REPRO, SPACE, TITLE, WXTRN
  Comments statements
  Common control sections
  Dummy control sections
  External dummy control sections
  Any conditional assembly instruction
  Macro instructions
- EJECT, ISEQ, PRINT, SPACE, TITLE instructions and comments statements must follow the ICTL instructions, if specified. However, they can precede or appear between macro definitions.
- All other assembler instructions must follow any source macro definitions specified.

START (Start Assembly) Instruction

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 4-3 for the format of the START statement.

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

Figure 4-3. START Instruction

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 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 the 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* below).

CSECT (Identify Control Section) Instruction

The CSECT instruction identifies the beginning or the continuation of a control section. The format is given in Figure 4-4.

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

Figure 4-4. CSECT Instruction

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 statements 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. (DOS/VS: Each CSECT starts at location zero.)

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 considers 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)
Macro Definitions
Conditional Assembly Instructions
Comments
COPY (depends upon the copied code)
EJECT
ENTRY
EXTRN
ICTL
ISEQ
PRINT
PUNCH
REPRO
SPACE
TITLE
WXTRN

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 instruction. A program can contain only one unnamed control section. It is possible to write a program that does not contain CSECT or START instruction, in which case the program will be assembled as one unnamed control section.

DSECT (Identify Dummy Section) Instruction

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 4-5 for the format of the DSECT instruction.

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

(DOS/VS: Blank name field allowed)

Figure 4-5. DSECT Instruction

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, they 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) Instruction

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 object 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 4-6 for the format of the COM instruction.

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

Figure 4-6. COM Instruction

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 in a common control section only 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.

Defining External Dummy Sections (OS/VS Only)

An external dummy section is a reference control section that allows you to describe storage areas for one or more source modules, to be used as:

- work areas for each source module; or
- communication areas between different source modules

To generate and use external dummy sections, you must specify a combination of the following:

- DXD or DSECT instruction
- Q-type address constant
- CXD instructions

The generation and use of external dummy sections is explained in detail in *OS/VS–DOS/VS–VM/370 Assembler Language* (GC33-4010).

DXD (Define External Dummy Section) Instruction

The DXD instruction allows you to identify and define an external dummy section that can be referred to by one or more source modules. This instruction may appear anywhere within a source module (after the ICTL instruction, if any) or after any source macro definitions you may specify within the source module. The format of the DXD instruction appears in Figure 4-7.

Note: The DSECT instruction also defines an external dummy section, but only if the symbol in its name field appears in a Q-type address constant in the same source module. Otherwise, the DSECT instruction defines a dummy section that cannot be referred to externally, that is, from a different source module.

The symbol in the name field of the DXD instruction must appear in the operand of a Q-type address constant. This symbol represents the address of the first byte of the external dummy section defined by the DXD instruction and has a length attribute value of 1.

The subfields in the operand field are specified in the same way as in the DS instruction. The assembler computes the amount of storage and the alignment required for an external dummy section from the area specified in the operand field.

The linkage editor or loader uses the information provided by the assembler to compute the total length of storage required for all external dummy sections specified in a program.

Note: If two or more external dummy sections for different source modules have the same name, the linkage editor uses the most restrictive alignment and computes the total length on the basis of the largest section.

Name	*Operation*	*Operand*
a symbol	DXD	same format as operand of a DS instruction

Figure 4-7. DXD Instruction

CXD (Reserve Storage for External Dummy Section Length) Instruction

The CXD instruction allows you to reserve a fullword area in storage into which the linkage editor or loader inserts the total length of all external dummy sections specified in the source modules that are assembled and linked into one object program.

The CXD instruction can appear in any of the source modules that are to be assembled and linked together. Figure 4-8 shows the format of this instruction.

The symbol in the name field, if specified, represents the address of a fullword area aligned on a fullword boundary. The symbol has a length attribute of 4.

Name	*Operation*	*Operand*
a symbol or blank	CXD	(none)

Figure 4-8. CXD Instruction

ADDRESSING

This section describes the assembler instructions used in symbolic addressing within a source module and between source modules.

Addressing Within a Source Module: Establishing Addressability

By establishing the addressability of a control section, you can refer to the symbolic addresses defined within the control section in the operands of machine instructions. The assembler converts these symbolic addresses into explicit addresses required for the assembled object code of the machine instructions. To do so, the assembler requires (1) a base address from which it can compute displacements to the addresses within the control section, and (2) a base register to hold this address. The USING and DROP assembler instructions convey this information to the assembler.

USING (Use Base Address Register) Instruction

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 instruction that makes that name addressable. See Figure 4-9 for the format of the USING instruction.

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

Figure 4-9. USING Instruction

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) Instruction

The DROP instruction specifies a previously available register that may no longer be used as a base register. See Figure 4-10 for the format of the DROP instruction.

Name	*Operation*	*Operand*
sequence symbol or *(DOS/VS only)* blank	DROP	up to seven absolute expressions of the form r1, r2, . . . ,r7

Figure 4-10. DROP Instruction

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. *(OS/VS and DOS/VS:* A blank operand is allowed and causes all registers to be dropped.)

Addressing Between Source Modules: Symbolic Linkage

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, EXTRN and WXTRN 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 and WXTRN assembler instructions. 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 later in this chapter under *Defining Data.* It is sufficient here to note that this constant may be considered an indirect linkage point. The constant is created from an externally defined symbol, but that symbol need not be identified by an EXTRN or WXTRN instruction.

The BAL and BALR instructions may be used with ENTRY, EXTRN and WXTRN instructions to branch between separately assembled control sections. The BAL instruction operand is coded in an EXTRN or WXTRN instruction 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 or WXTRN instruction. In both cases, the branch label must be identified by an ENTRY instruction in the assembly where it appears.

ENTRY (Identify Entry-Point Symbol) Instruction

The ENTRY instruction identifies linkage symbols that are defined in this program but may be used by some other program. See Figure 4-11 for the format of the ENTRY instruction.

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

Figure 4-11. ENTRY Instruction

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 need not 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) Instruction

The EXTRN instruction identifies linkage symbols that are used by one source module but which are defined in some other source module. Each external symbol must be identified; this includes symbols that name control sections. See Figure 4-12 for the format of the EXTRN instruction.

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

Figure 4-12. EXTRN Instruction

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

A V-type address constant need not be defined by an EXTRN instruction.

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.

WXTRN (Identify Weak External Symbol) Instruction

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

The automatic library call mechanism searches the call library for any unresolved external references. If it finds any of these references, it includes the module where the reference occurs in the load module produced by the linkage editor or loader. Refer to *OS Loader and Linkage Editor* for a full description of the automatic library call mechanism.

See Figure 4-13 for the format of the WXTRN instruction.

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

Figure 4-13. WXTRN Instruction

(DOS: This instruction is not available in the assembler.)

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

SYMBOL AND DATA DEFINITION

This section describes the assembly-time facilities you can use to (1) define and assign values to symbols, (2) define constants and storage areas, and (3) define control words.

Defining Symbols

EQU (Equate Symbol) Instruction

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 4-14 for the format of the EQU statement.

Name	*Operation*	*Operand*
variable symbol or ordinary symbol	EQU	expression (OS, DOS, DOS/VS) { expression 1 expression 1, expression 2 expression 1, expression 2, expression 3 expression 1,, expression 3 (OS/VS) }

Figure 4-14. EQU Instruction

The expression(s) in the operand field may be absolute or relocatable. Any symbols appearing in the expression must be previously defined.

The symbol in the name field is given the same value attribute as the expression in the operand field. The length and relocatability attributes are the same as for the expression unless specified differently (in OS/VS only) in expression 2 and expression 3. 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 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)
```

The value of the expression must be in the range $0–2^{18}-1$. Further information on the use of expression 2 and expression 3 appears in the publication *OS/VS–DOS/VS–VM/370 Assembler Language* (GC33-4010).

EQUR (Equate Symbol to Register Expression) Instruction

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 4-15 for the format of the EQUR instruction.

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

Figure 4-15. EQUR Instruction

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 instructions 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
```

Defining Data

There are three assembler instructions for defining data: Define Constant (DC), Define Storage (DS), and Define Control Word (CW).

These instructions 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) Instruction

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 4-16 for the format of the DC instruction.

Name	*Operation*	*Operand*
any symbol blank	DC	one or more operands, separated by commas, written in the format described in the text. (*DOS:* only one operand permitted)

Figure 4-16. DC Instruction

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 (2) 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) Instruction* 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 instructions (such as DS) that do not cause information to be assembled are not zeroed. However, bytes skipped to align a DC instruction 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 would in a DS instruction. A DC instruction with a zero duplication factor does not produce control dictionary entries. See *Forcing Alignment* under *DS (Define Storage) Instruction* below.

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 4-17 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 hexadecimal 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
Q	Address	space reserved for external dummy section offset

Figure 4-17. 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, Q) 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 below.

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 constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage.

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

- If the number of characters in the constant exceeds 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 byte 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 hexadecimal zero is added to an odd number of digits). If a length modifier is given, the constant is handled as follows:

- If the number of hexadecimal 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, using 1s and 0s enclosed in apostrophes. Only one binary constant can be specified in an operand. Duplication and length can be specified. The maximum length of a binary constant is 256 bytes.

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

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 five types of address constants: A, Y, R, V, and Q.

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 linkage 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 maximum value of the expression is $2^{15}-1$. 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.

R-Type Address Constant:
Caution: Relocatable, R-type constants must not be specifield 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 maximum value of the expression is $2^{16}-1$. 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. If length specification is used, a length of two bytes must be designated for a relocatable or complex expression and one or two bytes for an absolute expression.

The primary function of the R-type constant is to provide a two-byte relocatable address constant that can be processed by the OS or OS/VS 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 first two bytes of any control section (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) Instruction

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 4-18 for the format on the DS instruction.

Name	*Operation*	*Operand*
any symbol or blank	DS	one or more operands, separated by commas, written in the format described in the text. *(DOS:* only one operand allowed)

Figure 4-18. DS Instruction

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 in 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 instruction 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 characters 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 charcters 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 instruction 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   0H
AREA    DS   CL128
```

CW (Define Control Word) Instruction

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 aligned on *halfword* boundaries. The CW automatically performs this alignment and causes any skipped bytes to be zeroed. The internal machine format for a control word for a type 2 or type 3 channel adapter is as follows:

Byte	*Bits*	*Use*
1	0-1	Command code
1	2-3	Flags
1-2	4-13	Count
2-4	14-31	Data Address

See Figure 4-19 for the format of the CW instruction.

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 4-19. CW Instruction

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

1. An absolute 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 specifes 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 placed in bits 14-31. The data address must be halfword-aligned.

The following is an example of a CW instruction:

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

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

CONTROLLING THE ASSEMBLER PROGRAM

This section describes the assembler instructions that request the assembler to perform certain functions that it would otherwise perform in a standard, predetermined way. You can use these instructions to:

- Change the standard coding format for writing your source statements.
- Control the final structure of your assembled program.
- Alter the format of the source module and object code printed on the assembly listings.
- Produce punched card output in addition to the object deck.
- Substitute your own mnemonic operation codes for the standard codes of the assembler language (OS/VS only) via the OPSYN assembler instruction.
- Save and restore programming environments, such as the status of the PRINT options and the USING base register assignment.

Structuring a Program

The ORG and CNOP assembler instructions affect the location counter and thereby the structure of a control section. You can use them to interrupt the normal flow of assembly and redefine portions of a control section and to align data on any desired boundary.

ORG (Set Location Counter) Instruction

The ORG instruction is used to alter the setting of the location counter for the current control section. See Figure 4-20 for the ORG instruction format.

Name	*Operation*	*Operand*
sequence symbol or blank *(OS/VS:* any symbol or blank)	ORG	a relocatable expression or blank

Figure 4-20. ORG Instruction

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 instruction 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 instruction must not be used to specify a location below the beginning of the control section in which it appears. *Example:* 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) Instruction

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 an 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 occupies two bytes. See Figure 4-21 for the CNOP instruction format.

Name	*Operation*	*Operand*
sequence symbol or blank *(OS/VS:* any symbol or blank)	CNOP	two absolute expressions of the form *b, w*

Figure 4-21. CNOP Instruction

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 doubleword the location counter is to be set; *b* can be 0, 2, 4, or 6. Operand *w* specifies whether byte *b* is in a fullword (w=4) or doubleword (w=8). 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

Determining Statement Format and Sequence

You can change the standard coding conventions for the assembler language statements or check the sequence of source statements with the ICTL and ISEQ instructions.

ICTL (Input Format Control) Instruction

The ICTL instruction permits altering the normal format of source program statements (see Figure 4-22). 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 4-22. ICTL Instruction

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. *(DOS:* begin column +4). The column after the end column is used to indicate whether or not the next card is a continuation card.

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

Note: An invalid ICTL statement (under OS/VS only) will cause all subsequent statements to be printed and interpreted as comments.

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) Instruction

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 4-23 for the format of the ISEQ instruction.

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

Figure 4-23. ISEQ Instruction

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.

Listing Format and Output

The PRINT, TITLE, EJECT, and SPACE instructions request the assembler to produce listings and identify output cards in the object deck according to your special needs. They allow you to determine printing and page formatting options other than the ones the assembler program assumes by default. Among other things, you can introduce your own page headings, control the line spacing, and suppress unwanted detail.

Note: TITLE, SPACE, and EJECT do 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 instructions, if generated as macro instruction expansion, will ever be listed, regardless of continuation.

PRINT (Print Optional Data) Instruction

The PRINT instruction is used to control printing of the assembly listing; see Figure 4-24.

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

Figure 4-24. PRINT Instruction

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 instructions. A PRINT instruction controls the printing of the assembly listing until another PRINT instruction is encountered. Each option remains in effect until the corresponding opposite option is specified.

Until the first PRINT instruction (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
```

Note: For OS/VS only, the PUSH and POP instructions, described under *Saving and Restoring Programming Environment* later in this chapter, also influence the PRINT options by saving and restoring the PRINT status.

TITLE (Identify Assembly Output) Instruction

The TITLE instruction enables the programmer to identify the assembly listing and assembly output cards. See Figure 4-25 for the format of the TITLE instruction.

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

Figure 4-25. TITLE Instruction

The name field can contain a special symbol of from one to four (*OS/VS:* one to eight) alphabetic or numeric characters, in any combination. The contents of the name field is punched into columns 73-76 *(OS/VS:* 73-80) of all output cards for the program except those produced by the PUNCH and REPRO assembler instructions. Only the first TITLE statement in a program may have a special symbol or variable symbol in the name field. The name field of all subsequent TITLE statements must contain either a sequence symbol or a blank. *(Exception:* For OS/VS, the name field may contain an alphameric character string, or a variable symbol, or a combination of the two. Any of these options has significance only when coded in the first valid TITLE instruction in the program. If coded in subsequent TITLE instructions, they are accepted but are ignored.)

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 (*DOS/VS:* two apostrophes (") are allowed.)

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) Instruction

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 4-26 for the format of the EJECT instruction.

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

Figure 4-26. EJECT Instruction

If the line before the EJECT instruction appears at the bottom of a page, the EJECT instruction has no effect. Two EJECT instructions 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) Instruction

The SPACE instruction is used to insert one or more blank lines in the listing; see Figure 4-27.

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

Figure 4-27. SPACE Instruction

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.

Punching Output Cards

The PUNCH and REPRO instructions allow you to produce punched cards as output from the assembly in addition to those produced for the object module (object deck).

PUNCH (Punch a Card) Instruction

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 4-28 for the PUNCH instruction format.

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

Figure 4-28. PUNCH Instruction

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) Instruction

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 instructions 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 4-29 for the REPRO instruction format.

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

Figure 4-29. REPRO Instruction

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.

Redefining Symbolic Operation Codes (OS/VS Only)

The OPSYN assembler instruction is provided for defining machine and assembler instruction operation codes.

OPSYN (Define Symbolic Operation Code) Instruction

The OPSYN instruction allows you to define your own symbols to represent (1) machine and extended mnemonic branch instructions and (2) assembler instructions (including conditional assembly instructions).

You may also use the OPSYN instruction to prevent the assembler from recognizing a symbol that represents a current operation code.

Figure 4-30 gives the format of the OPSYN instruction.

Within a source module, the OPSYN instruction must appear after the ICTL instruction (if used) and can be preceded only by EJECT, ISEQ, PRINT, SPACE, and TITLE instructions. Also, OPSYN must precede any source macro definitions.

The OPSYN instruction can be used to assign to the symbol or operation code in the name field the meaning of the operation code in the operand field (see Figure 4-30[a]). *Example:* NEW OPSYN AR causes the meaning of the AR operation code to be assigned to the symbol NEW.

The operand of the OPSYN instruction must represent either the operation code of one of the machine or assembler instructions contained within this manual or an operation code defined by a previous OPSYN instruction.

The OPSYN instruction can also be used to prevent the assembler from recognizing a current operation code (see Figure 4-30[b]). In this case the operand field must be blank. *Example:* AR OPSYN. This instruction causes the AR instruction to lose its properties as an operation code.

A third use of the OPSYN instruction is for assigning the properties of one instruction to two different operation codes. *Example:* L OPSYN LR. This instruction assigns to the mnemonic code L the properties of mnemonic code LR; L and LR thus possess the same properties.

When the same symbol appears in the name field of successive OPSYN instructions, the latest definition is the effective one. *Example:* Given the sequence

```
STORE    OPSYN    ST
STORE    OPSYN    STH
```

the operation code STORE represents the STH instruction, not the ST instruction.

Name	*Operation*	*Operand*
(a) any symbol or operation code	OPSYN	an operation code
(b) an operation code	OPSYN	blank

Figure 4-30. OPSYN Instruction

Saving and Restoring Programming Environments (OS/VS Only)

The PUSH and POP assembler instructions can be used to save and restore the status of PRINT options and the base register assignment of your program.

PUSH (Save Current PRINT/USING Status) Instruction

The PUSH instruction allows you to save the current PRINT or USING status in "push-down" storage on a last-in, first-out basis. You can restore the status later, also on a last-in, first-out basis, by using a corresponding POP instruction. Figure 4-31 shows the format of the PUSH instruction.

Name	*Operation*	*Operand*	
			Options:
a sequence symbol or blank	PUSH	PRINT	1
		USING	2
		PRINT,USING	3
		USING,PRINT	4

Figure 4-31. PUSH Instruction

Specify one of the four options shown in the operand field. The PUSH instruction does not change the status of the current PRINT or USING instructions; the status is only saved.

Note: When the PUSH instruction is used in combination with the POP instruction, a maximum of four nests of PUSH PRINT–POP PRINT or PUSH USING–POP USING are allowed.

POP (Restore PRINT/USING Status) Instruction

The POP instruction allows you to restore the PRINT or USING status saved by the most recent PUSH instruction. Figure 4-32 shows the format of the POP instruction.

Name	*Operation*	*Operand*	
			Options:
a sequence symbol or blank	POP	PRINT	1
		USING	2
		PRINT,USING	3
		USING,PRINT	4

Figure 4-32. POP Instruction

Specify one of the four options shown in the operand field. The POP instruction causes the status of the current PRINT or USING instruction to be overridden by the PRINT or USING status saved by the last PUSH instruction.

Note: When the POP instruction is used in combination with the PUSH instruction, a maximum of four nests of PUSH PRINT–POP PRINT or PUSH USING–POP USING are allowed.

INTRODUCTION

The IBM Communications Controller macro facility is an extension of the Communications Controller assembler language. The facility 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 you to write the operands of a macro instruction in a fixed order. Keyword macro instructions permit you to write the operands of a macro instruction in a variable order. Mixed-mode macro instructions permit you 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 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.

Within the definition you can code COPY, MEXIT, MNOTE, or conditional assembly instructions.

The COPY instruction can be used to copy model statements and 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 (System Source Statement Library)

The same macro definition may be made available to more than one source program by placing the macro definition in the macro library (*DOS, DOS/VS:* System source statement library). The 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 library, it may be used by writing its corresponding macro instruction in a source program. Macro definitions must be in the system macro (or source statement) library under the same name as the prototype. The procedure for placing macro definitions in the macro library is described in:

For DOS: *DOS System Control and Services* (GC24-5036). (DOS macros are placed in the "A" sublibrary of the system source statement library.)

For OS: *OS Utilities* (GC28-6586).

For DOS/VS: *DOS/VS System Control Statements* (GC33-5376).

For OS/VS: *OS/VS Utilities* (GC35-0005).

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 you 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, you 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 you each time you write 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 you 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 you 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 you 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.

THE MACRO DEFINITION

A macro definition consists of (in the order shown):

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

Macro definitions appearing in a source program must appear before all PUNCH and REPRO instructions 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:* one continuation card).

Macro Definition Header (MACRO) Instruction

The macro definition header instruction indicates the beginning of a macro definition. It must be the first instruction in every macro definition. The format of this instruction shown in Figure 5-1.

Name	*Operation*	*Operand*
Blank	MACRO	Blank

Figure 5-1. MACRO Instruction

Macro Definition Trailer (MEND) Instruction

The macro definition trailer instruction indicates the end of a macro definition. It can appear only once within a macro definition and must be the last instruction in every macro definition. The format of this instruction is shown in Figure 5-2.

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

Figure 5-2. MEND Instruction

Prototype Statement

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 5-3.

Name	*Operation*	*Operand*
A symbolic parameter or blank	A symbol	0 to 200 (*DOS:* 0 to 100) symbolic (positional and/or keyword) parameters separated by commas (*OS/VS:* no fixed maximum number of parameters)

Figure 5-3. Macro Instruction Prototype Statement

The symbolic parameters are used in the macro definition to represent the name field and operands of the corresponding macro instruction. Symbolic parameters may be either positional or keyword parameters. Parameters are described 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 (unless, for OS/VS only, operation codes have been changed by an OPSYN instruction).

The operand field may contain 0 to 200 (*DOS:* 0 to 100) symbolic parameters, separated by commas (*OS/VS:* no fixed maximum number of parameters). If there are no symbolic parameters, comments may not appear.

The following are examples of a prototype statement:

```
&NAME    MOVE    &TO,&FROM
&NAME    MOVE    &TO=AREA2,&FROM=AREA1
```

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         &OPD1,&OPD2,&OPD3                  X
                      &OPD8,&OPD9  THIS IS THE           X
                      NORMAL STATEMENT FORMAT
&NAME2    OP2         &OPD1,       THIS IS THE           X
                      &OPD2,&OPD3, ALTERNATE             X
                      &OPD4        STATEMENT             X
                                   FORMAT
&NAME3    OP3         &OPD1,       THIS IS A             X
                      &OPD2,OPD3,  COMB-
                      &OPD4,                             X
                      &OPD9        INATION OF
                                   BOTH                  X
                      STATEMENT                          X
                      FORMATS
```

Alternate Statement Format

The prototype statement may be written in either normal assembler language format (all symbolic parameters precede any remarks) or 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:

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

Symbolic Parameters

A symbolic parameter is a type of variable symbol that is assigned values when you write a macro instruction.

Symbolic parameters allow you to pass values into the body of a macro definition from the calling macro instruction. You declare these parameters in the macro prototype statement. They can serve as points to substitution in the body of the macro definition and are replaced by the values assigned to them by the calling macro instruction. You may vary the statements that are generated for each occurrence of a macro instruction by changing the values of the symbolic parameters each time you code the macro instruction in the source program.

By using symbolic parameters with meaningful names you can indicate the purpose for which the parameters (or substituted values) are used.

Symbolic parameters must be valid variable symbols. They have a local scope; that is, the values they are assigned apply only to the macro definition in which they have been declared. The value of the parameter remains constant throughout the processing of the macro definition.

There are two kinds of symbolic parameter: *positional* and *keyword*. Each positional or keyword parameter used in the body of a macro definition must be declared in the prototype statement.

Positional Parameters

You should use a positional parameter in a macro definition if you wish to change the value of the parameter each time you call the macro definition. This is because it is easier, when coding the corresponding macro instruction, to supply the value for a positional parameter than for a keyword parameter. You need only write the desired value in the proper operand position within the macro instruction.

Values are assigned to positional parameters by the corresponding positional operands coded in the macro instruction that calls the macro definition.

A positional 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 positional parameter.

Examples of *valid* positional parameters:

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

Examples of *invalid* positional 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 positional parameters in a model statement must appear in the prototype statement of the macro definition.

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

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

In the following example, the characters HERE, FIELDA, and FIELDB of the MOVE macro instruction correspond to the positional parameters &NAME, &TO, 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 positional parameter appears in the comments field of a model statement, it is not replaced by the corresponding parameter of the calling macro instruction.

Keyword Parameters

You should use a keyword parameter in a macro definition for a value that changes infrequently from one call of the macro definition to another. By specifying a standard default value to be assigned to the keyword parameter, you can omit the corresponding keyword operand from the calling macro instruction.

Keyword parameters are particularly appropriate when the macro definition requires many parameters, only a few of which need be changed from their standard default values for any given call of the macro definition. Thus, when writing the macro instruction, you need code only those operands that correspond to the values to be changed.

A keyword parameter comprises (in order, without intervening blanks): (1) an ampersand (&), (2) a keyword of one to seven alphameric characters (the first must be amphabetic), (3) an equal sign; and optionally (4) a standard (default) value. (The standard value must not include a keyword.)

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 parameter statement.

Examples of *valid* keyword parameters:

```
&READER=            (standard value omitted)
&LOOP2=SYMBOL       (standard value supplied)
```

Examples of *invalid* keyword parameters:

```
CARDAREA=           (& omitted)
&TYPE               (= omitted)
&AREA=X'189A'       (standard value does not immediately
                     follow =)
```

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

The following is an example of a macro definition using keyword parameters. Note that the keyword parameters in the model statement also appear in the prototype statement.

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

Concatenating Symbolic Parameters

If a symbolic parameter (positional or keyword) 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 to 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, a period is required directly following the parameter. A period is optional if the symbolic parameter is to be concatenated with (1) another symbolic parameter or (2) a special character other than a left parenthesis or another period that follows it.

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

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

```
Header              MACRO
Prototype  &NAME    MOVE    &P,&S,&R1,&R2
Model (1)  &NAME    ST      &R1,&S.(&R2)
Model (2)           L       &R1,&P.B
Model (3)           ST      &R1,&P.A
Model (4)           L       &R1,&S.(&R2)
Trailer             MEND

Macro      HERE     MOVE    FIELD,SAVE,2,4

Generated (1) HERE  ST      2,SAVE(4)
Generated (2)       L       2,FIELDB
Generated (3)       ST      2,FIELDA
Generated (4)       L       2,SAVE(4)
```

The symbolic parameter &P is used in model statements (2) and (3) 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 model statements (1) and (4) to vary the operand fields of the corresponding generated statements (1) and (4). &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.

Model Statements

Model statements are the macro definition statements from which assembler language statements are generated at pre-assembly time. They allow you to determine the form of the statements to be generated. By specifying variable symbols as points of substitution in a model statement, you can vary the contents of the statements generated from that model statement. You can also use model statements into which you substitute values in open code.

Zero or more model statements may follow the prototype statement. A model statement consists of from one to four fields, separated by one or more blanks. 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* under *The Macro Instruction,* later in this chapter.

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, or ISEQ instructions. The characters * and .* may not be substituted for a variable symbol.

Operation Field

The operation field may contain (1) machine instruction (2) any assembler instruction listed in Chapter 4 (except END, ICTL, ISEQ, or PRINT), (3) a macro instruction, or (4) 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
- Conditional assembly instructions
- ICTL, ISEQ, MACRO, MEND, OPSYN, PRINT, or REPRO instructions
- END (DOS, DOS/VS restriction only)

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 Instructions,* below.

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

Operand Field

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

Comments Field

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

Processing Statements

The body of a macro definition can contain processing statements that, for example, can alter the content and sequence of the statements generated, or issue error messages within macro assembly expansions. Processing statements include conditional assembly instructions, inner macro instructions, and the COPY, MNOTE, and MEXIT instructions.

Conditional Assembly Instructions

Conditional assembly instructions allow you to determine at pre-assembly time the content of generated statements and the sequence in which they are generated. These instructions are:

GBLA, GBLB, GBLC, LCLA, LCLB, LCLC	(declaration of initial values of global and local SET symbols)
SETA, SETB, SETC	(assignment of new values to SET symbols)
AIF, AGO, ANOP	(branching instructions)
ACTR	(setting loop counter)

These instructions are discussed under *Conditional Assembly Instructions* later in this chapter.

Inner Macro Instructions

Macro instructions can be "nested" inside macro definitions, allowing you to call other macro definitions from within your own definitions. Nesting of macro instructions is described under *The Macro Instruction,* later in this chapter.

COPY Instruction

COPY instructions 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 instruction is shown in Figure 5-4.

Name	*Operation*	*Operand*
Blank	COPY	A symbol

Figure 5-4. COPY Instruction

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

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

MNOTE (Request Macro Error Message) Instruction

The MNOTE instruction may be used to request the assembler to generate an error message in the source program listing. The format of this instruction is shown in Figure 5-5.

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 single apostrophes (*DOS/VS:* Two apostrophes (") allowed.)

Figure 5-5. MNOTE Instruction

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 defintion. 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. (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 instruction. 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 attributes of both macro instruction operands are the same. If they are, statement 2 is the next statement processed by the assembler. If they are not, statement 4 is the next statement processed. 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 so, statement 3 is the next statement processed by the assembler. If not, statement 5 is the next statement processed. Statement 5 causes an error message (indicating that the type attribute is not F) to be printed in the source program listing.

MEXIT (Macro Definition Exit)

The MEXIT instruction causes the assembler to terminate processing of a macro definition. The format of this instruction is shown in Figure 5-6.

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

Figure 5-6. MEXIT Instruction

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 within a macro definition corresponding to an inner macro instruction, the next statement processed by the assembler is the next statement after the inner macro instruction in the macro definition.

MEXIT should not be confused with MEND. MEND indicates the end of a macro definition. MEND must be the last statement of each macro definition, 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 so, the assembler processes the remainder of the macro definition, starting with statement 3. If not, the next statement processed is statement 2. Statement 2 causes the assembler to terminate processing of the macro definition.

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, you may list all or selected macro instructions as follows:

For OS: See *OS Utilities* (GC28-6586) for using IEBPTPCH utility to list macros from the SYS1.GENLIB or SYS1.MACLIB library.

For DOS: See *DOS System Control and Service* (GC24-5036) for using SSERV utility to list macros from the system source statement library. (DOS system macros appear in the "A" sublibrary of this library.)

For OS/VS: See *OS/VS Utilities* (GC35-0005) for listing macros from macro libraries.

For DOS/VS: See *DOS/VS System Control Statements* (GC33-5376) for using the SSERV or ESERV utility to list macros from the system source statement library.

Note: The macros listed from the operating system libraries will include System/370 machine instruction codes instead of 3704/3705 machine codes; however, the use of the assembler language components in writing macro definitions is the same.

System Variable Symbols

System variable symbols are local variable symbols that are assigned values automatically by the assembler. There are six system variable symbols:

```
&SYSDATE  (OS/VS only)
&SYSECT
&SYSLIST
&SYSNDX
&SYSPARM  (OS/VS, DOS/VS only)
&SYSTIME  (OS/VS only)
```

System variable symbols may be used in the name, operation and operand fields of statements in macro definitions, but not in statements outside macro definitions. They may not be defined as symbolic parameters or SET symbols, nor may they be assigned values by SETA, SETB, and SETC instructions.

&SYSDATE – Current Date (OS and OS/VS Only)

The global system variable symbol &SYSDATE can be used to obtain the date on which your source module is assembled. The date is printed in the page heading of assembly listings.

The symbol &SYSDATE is assigned a read-only value of the format *mm/dd/yy* (*mm* = month, *dd* =day,yy= year).

Note: The value of the type attribute of &SYSDATE (T'&SYSDATE) is always U and the value of the count attribute (K'&SYSDATE) is always eight.

&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. (Inner and outer macro instructions are discussed under *Nesting in Macro Definitions,* later in this chapter.)

When &SYSECT is used in a macro definition, the value substituted for &SYSECT is the name of the last CSECT, DSECT, or START instruction that occurs before the macro instruction. If no named CSECT, DSECT, or START instructions occur before a macro instruction, &SYSECT is assigned a null character value for that macro instruction.

CSECT or DSECT instructions processed in a macro defintion 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 instructions or inner macro instructions in that definition.

In the example that follows; statement 8 is the last CSECT, DSECT, or START instructions 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 instruction 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 instruction for statement 4. This is the last CSECT, DSECT, or START instruction 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.

Statement 1 is used to generate a CSECT instruction for statement 5. This is the last CSECT, DSECT, or START instruction 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.

```
                    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)
```

&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 *n*th positional macro instruction operand. In addition, if the *n*th operand is a sublist, then &SYSLIST (n,m) may be used to refer to the *m*th 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:* 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 *n*th operand is omitted, N' is zero; if the *n*th operand is not a sublist, N' is one.

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

1. A sublist is considered to be one operand.
2. The count includes 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

&SYSNDX – Macro Instruction Index

The system variable symbol &SYSNDX may be concatenated with other characters to create unique names for statements generated from the same model statement.

&SYSNDX is assigned the four-digit number 0001 for the first macro instruction processed by the assembler, and it is incremented by one for each subsequent inner and outer macro instruction processed.

If &SYSNDX is used in a model statement, SETC or MNOTE instruction, or a character relation in a SETB or AIF instruction, the value substituted for &SYSNDX is the four-digit number of the macro instruction being processed, including leading zeros.

If &SYSNDX appears in arithmetic expressions (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.

&SYSPARM – Pass System Parameter (OS/VS and DOS/VS Only)

The global system variable symbol &SYSPARM can be used to communicate with an assembly source module through the job control language. Through &SYSPARM, you pass a character string into the source module from a job control language EXEC statement or from a program that dynamically invokes the assembler. Thus, you can set a character value from outside a source module and then examine it as part of the source module at pre-assembly time during conditional assembly processing.

The symbol &SYSPARM is assigned a read-only value in a job control statement or in a field established by a program that dynamically invokes the assembler. It is treated within the source program as a global SETC symbol except that its value cannot be changed. The largest value &SYSPARM can hold, for OS/VS, is 255 characters, which can be specified by a program invoking the assembler. However, if the

PARM field of the EXEC statement is used to specify the value, the PARM field restrictions limit its maximum possible length to 57 characters. The largest value &SYSPARM can hold for DOS/VS is eight characters.

For further detail on the use of &SYSPARM, see *OS/VS–DOS/VS–VM/370 Assembler Language* (GC33-4010).

&SYSTIME – Current Time of Day (OS and OS/VS Only)

The global system variable symbol &SYSTIME can be used to obtain the time of day at which your source module is assembled. The time is printed in the page heading of assembly listings.

The symbol &SYSTIME is assigned a read-only value of the format *hh.mm* (*hh* = hours, *mm* = minutes) in the 24-hour system. (*Example:* 10.15 = 10:15 a.m.; 22.15 = 10:15 p.m.)

Note: The value of the type attribute of &SYSTIME (T'&SYSTIME) is always U and the value of the count attribute (K'&SYSTIME) is always five.

Listing Options (OS/VS Only)

In addition to the print options that can be set from inside a source module, you can set other listing options from outside the module by means of the PARM parameter of the EXEC job control statement or by a program that dynamically invokes the assembler. The two options available (only under OS/VS) are LIBMAC and MCALL.

LIBMAC Option

The LIBMAC option allows you to print, in the assembly listings, (1) the library macro definitions called from your source module and (2) any statements in open code following the first END statement (coded or generated) processed by the assembler.

The option NOLIBMAC, which is the default option, suppresses the listing of the foregoing items.

MCALL Option

The MCALL option allows you to list all the inner macro instructions that the assembler processes. The option NOMCALL, which is the default option, suppresses the listing of inner macro instructions.

THE MACRO INSTRUCTION

The macro instruction provides the assembler with:

- The name of the macro definition to be processed.
- The values to be passed to the macro definition. The assembler uses the information either in processing the macro definition or for substituting values into model statements within the definition.

The output from a macro definition can be:

- A sequence of statements generated from the model statements for further processing at assembly time.
- Values assigned to global SET symbols. These values can be used in other macro instructions and in open code.

Statement Format

The format of a macro instruction is shown in Figure 5-7.

Name	*Operation*	*Operand*
Any symbol or blank	Mnemonic operation code	0-200 operands (*DOS:* 0-100 operands), separated by commas.

Figure 5-7. Macro Instruction Format

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 (OS, OS/VS:) in the source program or in the macro library, or (DOS, DOS/VS:) 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 (or 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 corresponds to (*DOS:* 127 characters) the placement and order of the symbolic parameters in the operand field of the prototype statement.

Alternate Statement Format

You may write macro instructions using the same alternate format that can be used to write prototype statements. If you use this format, a blank does not always indicate the end of the operand field. The alternate format is described earlier in this chapter under *Prototype Statement.*

Macro Instruction Operands

Any combination of up to 255 characters *(DOS:* 127 characters) may be used as a macro instruction operand, provided that the following rules concerning apostrophes, parentheses, equal signs, ampersands, commas, and blanks are observed.

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

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

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

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 *n*th left parenthesis must appear to the left of the *n*th 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 and Outer 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)
```

Omitted Operands

If a positional 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.

If a keyword operand that appears in the prototype statement is omitted from the macro instruction, no comma is required to indicate its omission. The default value specified in the prototype statement will be used when the macro is expanded.

The following example shows a macro instruction preceded by its corresponding prototype statement. The positional 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 (1) to a collection of macro instruction operands as a single operand or (2) to 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 *n*th 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 *n*th 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.

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 you wish 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       R1,(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.

Nesting in Macro Instructions

A "nested" macro instruction is a macro instruction that you specify as one of the model statements within the body of a macro definition. Use of the nesting technique allows you to call for the expansion of a macro definition from within another macro definition.

Inner and Outer Macro Instructions

Any macro instruction you write in the open code of a source module is an *outer* macro instruction or call. Any macro instruction that appears within a macro definition is an *inner* 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 parameters 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 replaces the inner macro instruction.

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

In this example, the inner macro instruction contains two symbolic parameters, &S and &T. The characters (X,Y,Z) and J of the macro instruction correspond to &S

and &T, respectively. Therefore, these characters replace the symbolic parameters in the operand field of the inner macro instruction.

The assembler then uses the macro definition that corresponds to the inner macro instruction to generate statements to replace the inner macro instruction. The fifth through the tenth generated statements have been generated for the inner macro instruction. See *Operand Sublists* above 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 are covered under *Data Attributes*.

Levels of Nesting

The code generated by a macro definition called by an inner macro call is nested inside the code generated by the macro definition that contains the inner macro call. In the called macro definition you can include a macro call to another macro definition. Thus, you can nest macro calls at different levels.

The zero level includes other macro calls (calls that appear in open code). The first level of nesting includes inner macro calls that appear inside macro definitions called from the zero level; the second level of nesting includes inner macro calls inside macro definitions called from the first level; and so on up to the nesting limit.

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

THE CONDITIONAL ASSEMBLY LANGUAGE

The conditional assembly language allows you to perform general arithmetic and logical computations as well as many of the other functions you can perform with any other programming language. In addition, by writing conditional assembly instructions in combination with other assembler language statements you can:

- Select sequences of these source statements, called model statements, from which machine and assembler instructions are generated.
- Vary the contents of these model statements during generation.

The assembler processes the instructions and expressions of the conditional assembly language at pre-assembly time. Then, at assembly time, it processes the generated instructions. Conditional assembly instructions, however, are not processed after pre-assembly time.

The conditional assembly language is most versatile when used to interact with symbolic parameters and the system variable symbols inside a macro definition. However, you can also use the conditional assembly instructions in open code.

Elements and Functions

The elements of the conditional assembly language are:

- SET symbols that represent data.
- Attributes that represent different characteristics of data.
- Sequence symbols that act as labels for branching to statements at pre-assembly time.

The functions of the conditional assembly language are:

- Declaring SET symbols as variables for use by the conditional assembly language in its computations.
- Assigning values to the declared SET symbols.
- Evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, or as condition tests for branch instructions.
- Selecting characters from strings for substitution in and concatenation to other strings, or for inspection in condition tests.
- Branching and exiting from conditional assembly loops.

Conditional Assembly Instructions

The conditional assembly instructions allow you 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, you can use these instructions to generate many different sequences of statements from the same macro definition,

There are 13 conditional assembly instructions:

```
LCLA    GBLA    SETA    AIF     ACTR
LCLB    GBLB    SETB    AGO
LCLC    GBLC    SETC    ANOP
```

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

Where the use of an instruction in open code differs from its use within macro definitions, the difference is described in the text.

The LCLA, LCLB, LCLC, GBLA, GBLB, and GBLC 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. You may 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 5-17.

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 you write a macro instruction, whereas SET symbols are assigned values when you write 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.

Declaring SET Symbols

You must declare SET symbols before using them. When a SET symbol is declared, 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 declared when it appears in the operand field of an LCLA, LCLB, LCLC, GBLA, GBLB, or GBLC instruction.

Using Variable Symbols

The SETA, SETB, and SETC instructions may be used to change the values initially assigned to SET symbols by local or global declarations. 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.

SET symbols and symbolic parameters may be contrasted as follows. 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. On the other hand, 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 (that is, within open code).

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 in open code.

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 is considered a different variable symbol each time it is used.

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

All variable symbols may be concatenated with other characters, in the same way that symbolic parameters may be concatenated with other characters. The rules for concatenating symbolic parameters with other characters are described earlier in this chapter 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.

Data Attributes

The assembler assigns four attributes to macro instruction operands and to symbols in the program: the type of operand or symbol; 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.

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, you 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 is represented by a notation, as follows.

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

You may refer to an attribute in the following ways:

1. In a statement that is outside macro definitions (that is, in open code), 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 within 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 instructions:

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
Q	Q-type address constant (OS/VS only)
R	A-, V-, R- or Y-type (or Q-type [OS/VS]) 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 instructions or that appear in the operand field of an EXTRN or WXTRN instruction:

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 instructions, to any symbols occurring more than once in the name field of source statements, and to all symbols naming statements with expressions as modifiers.

U	Undefined

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 macro instruction 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 invalid 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 parentheses 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 characters that replace the variable symbol, 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 allow you 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 instruction 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, or 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.

Examples of *valid* sequence symbols:

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

Examples of *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 example illustrates this rule:

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

The symbolic parameter &NAME is used in the name field of the prototype statement (statement 1) and the first model statement (statement 2). In the macro instruction (statement 3), 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 (Declare Local SET Symbol) Instructions

The LCLA, LCLB, and LCLC instructions declare (assign initial values to) local SETA, SETB, and SETC symbols, respectively. The SETA, SETB, and SETC symbols are assigned the initial values of 0, 0, and null character value, respectively.

The format of these instructions is shown in Figure 5-8.

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 5-8. LCLA, LCLB, LCLC Instructions

You should not declare 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 (that is, in open code) must appear after all GBLA, GBLB, and GBLC instructions outside macro definitions, before all conditional assembly instructions and PUNCH and REPRO instructions outside macro definitions, and before the first control section of the program.

GBLA, GBLB, GBLC (Declare Global SET Symbol) Instructions

The GBLA, GBLB, and GBLC instructions declare (assign initial values to) global SET symbols, just as the LCLA, LCLB, and LCLC instructions declare local SET symbols.

The format of these instructions is shown in Figure 5-9.

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 5-9. GBLA, GBLB, GBLC Instructions

The GBLA, GBLB, and GBLC instructions declare (assign initial values to) global SETA, SETB, and SETC symbols respectively, and assign the same initial values (0, 0, and null, respectively), as the corresponding types of local instructions (LCLA, LCLB, LCLC). 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 declare any global SET symbols whose first four characters are &SYS.

GBLA, GBLB, or GBLC instructions within a macro definition must immediately follow (1) the prototype statement, or (2) 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 instructions outside macro definitions, and (3) before the first control section of the program.

Within a macro definition, all GBLA, GBLB, and GBLC instructions must appear before any LCLA, LCLB, and LCLC instructions. The same is true when these instructions appear in open code.

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 comprises the source statements (macro definitions and open code). The second part shows the statements that would be generated by the assembler after it processed the source statements.

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.

Source statements:

```
                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
```

Generated code:

```
     FIRST       LR      5, 0
                 LR      5, 0
                 LR      5, 0
                 LR      5, 0
                 END     FIRST
```

&A is *declared* as a local SETA symbol in a macro definition (statement 1) and outside the macro definition (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 the macro definition. Therefore, statement 3 (which is within the macro definition) does not affect the value used for &A in statements 5 and 6 (which are outside the definition). 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.

Source statements:

```
                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
```

Generated code:

```
     FIRST       LR      5, 0
                 LR      5, 1
                 LR      5, 1
                 LR      5, 2
                 END     FIRST
```

&A is *declared* as a global SETA symbol within a macro definition (statement 1) and outside the macro definition (statement 4). &A is *used* twice within the macro definition (statements 2 and 3) and twice outside the macro definition (statements 5 and 6).

Since &A is a global SETA symbol both within the macro definition and outside the definition, it is the same SETA symbol in both cases. Therefore, statement 3 (within the macro definition) affects the value used for &A in statements 5 and 6 (outside the macro definition).

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 declared 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 within each macro definition, the value of &A may differ in the two definitions. Therefore, statement 3 (within one macro definition) does not affect the value used for &A in statement 5 (within the other macro definition). Similarly, statement 6 does not affect the value used for &A in statement 2.

Source statements:

```
                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
```

Generated code:

```
    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.

Source statements:

```
                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
```

Generated code:

```
    FIRST       LR      5,0
                LR      5,1
                LR      5,2
                LR      5,3
                END     FIRST
```

&A is declared 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 (within one macro definition) affects the value used for &A in statement 5 (within the other macro definition). Similarly, statement 6 affects the value used for &A in statement 2.

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

Source statements:

```
                MACRO
    &NAME       LOADA   &A
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
```

Generated code:

```
    FIRST       LR      5,0
                LR      5,1
                LR      5,0
                LR      5,2
                LR      5,3
                LR      5,0
                END     FIRST
```

&A is *declared* as a global SETA symbol in two different macro definitions (statements 1 and 4), but it is *declared* as a local SETA symbol outside these macro definitions (statement 7). &A is used twice within each macro definition and twice outside the 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 definition. However, since &A is a *local* SETA symbol outside the macro definitions, it is a *different* SETA symbol outside the macro definitions.

Therefore, statement 3 (within one macro definition) affects the value used for &A in statement 5 (within the other macro definition), but it does not affect the value used for &A in statements 8 and 9 (which are outside the 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 declared as subscripted SET symbols. The local SET symbols declared previously were all nonsubscripted SET symbols.

Subscripted SET symbols provide a convenient way to use a single 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)
```

Declaring Subscripted SET Symbols: To use a subscripted SET symbol, you must declare (by 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 identical to the initial value assigned to the corresponding type of nonsubscript SET symbol.

If a subscripted SET symbol is global, the same dimension must be used with the SET symbol each time it is declared as global.

The maximum dimension that can be used with a SETA, SETB, or SETC symbol is as follows:

DOS: 255
DOS/VS: 255
OS: 2500
OS/VS: 32,767

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.

Example: The following statements declare 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 declared 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 registers 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.

Assigning Values to SET Symbols

The SETA, SETC, and SETB instructions are used for assigning new values to local and global SET symbols initially declared by LCLA, LCLB, LCLC, GBLA, GBLB, and GBLC instructions.

SETA (Set Arithmetic) Instruction

The SETA instruction may be used to assign an arithmetic value to a SETA symbol. The format of this instruction is shown in Figure 5-10.

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

Figure 5-10. SETA Instruction

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 of this publication.

Note: A SETC variable symbol may appear in a SETA expression only if the value of the SETC variable is one to eight decimal digits. The decimal digits are 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. Five levels of parentheses are allowed (*OS/VS:* 6 levels) and an expression may not consist of more than 16 terms (*OS/VS:* 20 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 are evaluated before the remainder of the terms in the expression is 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:

Source statements:

```
                 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
```

Generated code:

```
    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:

Source statements:

```
                 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
Generated code:
   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.

```
Source statements:
               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
               ADDX        (A, B, C, D, E), (3,4)
Generated code:
               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) Instruction

The SETC instruction is used to assign a character value to a SETC symbol. The format of this instruction is shown in Figure 5-11.

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

Figure 5-11. SETC Instruction

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 (*DOS:* up to 127 characters) enclosed in apostrophes.

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

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 you to assign part of a character value to a SETC symbol.

If you wish 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 you wish 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 each 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 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 (*DOS:* 127 characters). If a value greater than 8 is specified, the leftmost 8 characters will be used.

The following are valid substring notations:

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

The following are invalid substring notations:

```
'&BETA'(4,6)
```
(blanks between character values 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:

Source 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
```

Generated code:

```
    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:

Source statements:

```
                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
```

Generated code:

```
    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:

Source statements:

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

Generated code:

```
   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) Instruction

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 5-12.

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 5-12. SETB Instruction

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

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

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

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

Any expression that may be used in the operand field of a SETA instruction may be used as an arithmetic expression in the operand field of a SETB instruction. Anything that may be used in the operand field of a SETC instruction may be used as a character value in the operand field of a SETB instruction. This includes substring and type attribute notations. The maximum size of the character values that can be compared is 255 characters (*DOS:* 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 assembler evaluates a logical expression in the operand field of a SETB instruction as follows:

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 parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence.

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

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

The parenthesized portion or portions of a logical expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first. Five levels of parentheses are permissible (*OS/VS:* 6 levels).

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.

Source statements:

```
                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
```

Generated code:

```
    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 value 5RS, either of the following statements may be used to assign &WORD the character vaue AB%45RS:

```
&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.

Branching

Four conditional assembly branching instructions are available for use within macro definitions or open code.

AIF (Conditional Branch) Instruction

The AIF instruction is used to conditionally alter the sequence in which the assembler processes source program statements or macro definition statements. The assembler assigns a maximum count of 4096 (*DOS:* 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 4096 (*DOS:* 150) is set up for the inner macro definition. After processing the inner definition and returning to the higher definition, the assembler restores the saved count.

The format of this instruction is shown in Figure 5-13.

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

Figure 5-13. AIF Instruction

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. If the expression is false, the next sequential statement is processed.

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

If an AIF instruction is within 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 defintions, 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) Instruction

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 (*DOS:* 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 4096 (*DOS:* 150) is set up for the inner macro definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored.

The format of this instruction is shown in Figure 5-14.

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

Figure 5-14. AGO Instruction

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 within 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 determines if the type attribute of the first macro instruction operand is the letter F. If so, statement 3 is the next statement processed by the assembler. If not, statement 2 is the next statement processed.

Statement 2 indicates 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) Instruction

The ACTR instruction is used to assign a maximum count, different from the standard count of 4096 (*DOS:* 150), to the number of AGO and AIF branches executed within a macro definition or within the source program.

The format of this instruction is shown in Figure 5-15.

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

Figure 5-15. ACTR Instruction

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 not zero or negative, the counter 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 inside a macro definition, the assembler terminates the entire nest of macro definitions and continues processing with the next source statement.
2. If processing the source program, the assembler generates an END card.

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.

(*DOS/VS:* The assembler halves the ACTR counter value when it encounters serious syntax errors in conditional assembly instructions.)

ANOP (Assembly No Operation) Instruction

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 5-16.

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

Figure 5-16. ANOP Instruction

If you wish 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 wish 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 determines if the type attribute of the first macro instruction operand is the letter F. If so, statement 2 is the next statement processed. If not, 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 is statement 4, the statement following the ANOP instruction.

Conditional Assembly Elements

Figure 5-17 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 indicates the conditional assembly instructions in which a particular element can be used.

The intersection of a column and a row indicates whether an element can be used in an instruction, 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 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

N = name field
O = operand field

Figure 5-17. Elements of Conditional Assembly Instructions

This appendix summarizes the differences between the OS, DOS, OS/VS, and DOS/VS versions of the Communications Controller assembler. Features, options, and instructions that do not appear in the comparison are the same for all four versions.

Assembler Language Feature	*Version of Communications Controller Assembler*			
	DOS	DOS/VS	OS	OS/VS
Maximum number of continuation cards per statement	1	2	2	2
Maximum number of external symbol dictionary (ESD) entries	255	255	255	399
Maximum Location Counter Value	2^{18}-1	2^{18}-1	2^{18}-1	2^{18}-1
All control sections initiated by CSECT start at location counter value of 0	No	Yes	No	No
Maximum number of characters per symbol	8	8	8	8
SELF-DEFINING TERMS:				
Binary: maximum number of bits	18			
Decimal: maximum decimal value	262,143			
Hexadecimal: maximum hex value	3FFFF			
Character: maximum number of characters	2			
EXPRESSIONS (ABSOLUTE & RELOCATABLE):				
Maximum number of binary (+-*/) operators within expression	15	15	15	19
Unary operators allowed within expression	No	Yes	No	Yes
Maximum number of terms within expression	16	16	16	20
Maximum levels of parentheses	5	5	5	6
ASSEMBLER INSTRUCTIONS:				
ACTR—allowed within open code	No	Yes	No	Yes
AIF—maximum number of branches in source program or macro definition	150	4096	4096	4096
AGO—maximum number of branches in source program or macro definition	150	4096	4096	4096
CNOP—kind of name entry allowed	sequence symbol or blank	sequence symbol or blank	sequence symbol or blank	any symbol or blank
COPY—				
copying of macro definitions allowed	No	Yes	No	Yes
nesting depth allowed	none	3	none	5
CSECT—always starts at location counter value of 0	No	Yes	No	No
CXD—valid instruction	No	No	No	Yes

	DOS	DOS/VS	OS	OS/VS
DC–				
Expressions allowed as modifiers	Yes	Yes	Yes	Yes
Number of operands allowed	1	Multiple	Multiple	Multiple
Multiple nominal values allowed in constant	No	Yes	Yes	Yes
Length (incl. bit length) modifier allowed	No	Yes	No	Yes
Scaling modifier allowed (F and H constants only)	No	Yes	No	Yes
Exponent modifier allowed (F and H constants only)	No	Yes	No	Yes
Types allowed		C X B F H A Y V Q*		*OS/VS only
Types not allowed		E D L P Z S		
DROP–blank operand entry allowed	No	Yes	No	Yes
DS–				
Number of operands allowed	1	Multiple	Multiple	Multiple
Multiple nominal values allowed in constant	No	Yes	Yes	Yes
Length (excl. bit length) modifier allowed	Yes	Yes	Yes	Yes
Maximum length modifier allowed	65,535	65,535	65,535	65,535
Bit length modifier allowed	No	Yes	No	Yes
Scaling modifier allowed (F and H constants only)	No	Yes	No	Yes
Exponent modifier allowed (F and H constants only)	No	Yes	No	Yes
Types allowed		C X B F H A Y V Q*		*OS/VS only
Types not allowed		E D L P Z S		
DSECT–blank name entry allowed	No	Yes	No	Yes
END–generated or copied END statement allowed within macro definition	No	Yes	No	Yes
EQU–				
second operand (length attribute) allowed	No	No	No	Yes
third operand (type attribute) allowed	No	No	No	Yes
ICTL–end column required to be no further left than begin column +4	Yes	No	No	No
MNOTE–allowed in open code	No	No	No	Yes
OPSYN–valid instruction	No	No	No	Yes
ORG–kind of name entry allowed	sequence symbol or blank	sequence symbol or blank	sequence symbol or blank	any symbol or blank
POP–valid instruction	No	No	No	Yes
PRINT–allowed inside macro definition	No	Yes	No	Yes
PUSH–valid instruction	No	No	No	Yes
START–generation of START instruction within macro definition allowed	No	Yes	No	Yes
TITLE–maximum number of characters in name (if not a sequence symbol)	4	4	4	8
WXTRN–valid instruction	No	Yes	Yes	Yes

	DOS	DOS/VS	OS	OS/VS
ASSEMBLER PROGRAM OPTIONS:				
ALOGIC	No	No	No	Yes
BUFSIZE	No	No	No	Yes
CATAL	Yes	Yes	No	No
DECK	Yes	Yes	Yes	Yes
EDECK	No	Yes	No	No
ESD	No	No	No	Yes
FLAG	No	No	No	Yes
LIBMAC	No	No	No	Yes
LINECOUNT	No	No	Yes	Yes
LINK	Yes	Yes	No	No
LIST	Yes	Yes	Yes	Yes
LOAD	No	No	Yes	No
MCALL	No	No	No	Yes
MLOGIC	No	No	No	Yes
OBJ	No	No	No	Yes
RENT	No	No	Yes	Yes
RLD	No	No	No	Yes
SYSPARM	No	Yes	No	Yes
XREF	Yes	Yes	Yes	No
XREF (Full)	No	No	No	Yes
XREF (Short)	No	No	No	Yes
MACRO FACILITY FEATURES: (see also conditional assembly features below)				
Maximum number of operands or symbolic parameters allowed in macro instruction	100	200	200	No fixed Maximum
Maximum number of characters in operand	127	255	255	255
Positional and keyword operands can be mixed	No	No	No	Yes
System variable symbols available:				
Global				
&SYSDATE	No	No	Yes	Yes
&SYSPARM	No	Yes	No	Yes
&SYSTIME	No	No	Yes	Yes
Local				
&SYSECT	Yes	Yes	Yes	Yes
&SYSLIST	Yes	Yes	Yes	Yes
&SYSNDX	Yes	Yes	Yes	Yes
SET SYMBOLS:				
Type (T') and Count (K') attributes allowed in open code	No	No	No	Yes
SET Symbol Declaration–mixture of global and local allowed	No	No	No	Yes
Within macro definition, global and local declarations required to immediately follow macro prototype statement	Yes	Yes	Yes	No
Within open code, global and local declarations required to follow any source macro definitions and precede first control section	Yes	Yes	Yes	No
SETC instruction–duplication factor allowed in operand	No	No	No	Yes
Subscripted SET symbols–maximum array dimension	255	255	2500	32,767

	DOS	DOS/VS	OS	OS/VS
CONDITIONAL ASSEMBLY FEATURES: (see also Macro Facility Features above)				
Arithmetic expressions–				
Maximum number of unary and binary operators allowed	16	16	16	25
Maximum number of terms	15	15	15	25
Maximum levels of parentheses	5	5	5	11
Length attribute (L') allowed	No	Yes	No	Yes
Integer (I') attribute allowed	No	Yes	No	Yes
Scaling attribute (S') allowed	No	Yes	No	Yes

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 B-1. Instruction Formats (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*
A	An absolute or relocatable expression that specifies an address.
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 B-1. Instruction Formats (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 a length specification up 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 C-1. Summary of Constants

Appendix D: Assembler Instructions

OPERATION	*NAME ENTRY*	*OPERAND ENTRY*
ACTR	Omit	An arithmetic SETA expression
AGO	A sequence symbol or blank	A sequence symbol
AIF	A sequence symbol or blank	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP	A sequence symbol	Omit
CNOP	A sequence symbol or blank	Two absolute expressions, separated by a comma
COM	A sequence symbol or blank	Omit
COPY	Omit	A symbol
CSECT	Any symbol or blank	Omit
CW	Any symbol or blank	Four operands, separated by commas
CXD	Any symbol or blank	Omit
DC	Any symbol or blank	One or more operands, separated by commas
DROP	A sequence symbol or blank	One to sixteen absolute expressions, separated by commas (*OS/VS and DOS/VS:* can be blank)
DS	Any symbol or blank	One or more operands, separated by commas
DSECT	A variable symbol or an ordinary symbol (DOS/VS - can be blank)	Omit
DXD	Any symbol	One or more operands, separated by commas
EJECT	A sequence symbol or blank	Omit
END	A sequence symbol or blank	A relocatable expression or blank
ENTRY	A sequence symbol or blank	One or more relocatable symbols, separated by commas
EQU	A variable symbol or an ordinary symbol	An absolute or relocatable expression (OS, DOS, DOS/VS) One to three absolute or relocatable expressions (OS/VS)
EQUR	A variable symbol or an ordinary symbol	An expression grouping of the form R(N) or Q.
EXTRN	A sequence symbol or blank	One or more relocatable symbols, separated by commas
GBLA	Omit	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLB	Omit	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLC	Omit	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
ICTL	Omit	One to three decimal values, separated by commas
ISEQ	Omit	Two decimal values, separated by a comma
LCLA	Omit	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLB	Omit	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLC	Omit	One or more variable symbols separated by commas
MACRO[1]	Omit	Omit
MEND[1]	A sequence symbol or blank	Omit
MEXIT[1]	A sequence symbol or blank	Omit
MNOTE[1]	A sequence symbol, a variable symbol, or blank	A severity code, followed by a comma, followed by any combination of characters enclosed in apostrophes (*DOS/VS:* two apostrophes allowed)

OPERATION	*NAME ENTRY*	*OPERAND ENTRY*
OPSYN	Any symbol or operation code	
ORG	A sequence symbol or blank	A relocatable expression or blank
POP	A sequence symbol or blank	One of four options
PRINT	A sequence symbol or blank	One to three operands
PUNCH	A sequence symbol or blank	One to eighty characters, enclosed in apostrophes
PUSH	A sequence symbol or blank	One of four options
REPRO	A sequence symbol or blank	Omit
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 blank	A decimal self-defining term or blank
START	Any symbol or blank	A self-defining term or blank
TITLE[3]	A special symbol (0 to 4 characters [OS/VS: 0 to 8]), a sequence symbol, a variable symbol, or blank	One to 100 characters, enclosed in apostrophes *(DOS/VS:* two apostrophes allowed)
USING	A sequence symbol or blank	An absolute or relocatable expression followed by 1 to 16 absolute expressions, separated by commas
WXTRN	A sequence symbol or blank	One or more relocatable symbols, 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 for a description of the name entry.

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 equivalent to a symbol, or blank	Any combination of characters (including variable symbols)
Prototype Statement[1]	A symbolic parameter or blank	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 equivalent to a symbol,[2] or blank	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 equivalent to a symbol, or blank	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 instruction is performed.

Figure D-1. Assembler Statements

Figure E-1 through E-4 in this appendix summarize the macro language.

Figure E-1 indicates which macro language elements may be used in the name and operand entries of each statement.

Figure E-2 is a summary of the expressions that may be used in macro instruction statements.

Figure E-3 is a summary of the attributes that may be used in each expression.

Figure E-4 is a summary of the variable symbols that may be used in each expression.

	Variable Symbols														
	Global SET Symbols				Local SET Symbols			System Variable Symbols			Attributes				
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 E-1. 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 is:	-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 (127 characters for DOS); see chapter 5 under SETC - SET CHARACTER. If the two character expressions are of unequal size, then the smaller one will always compare less than the larger.

[3]Maximum of eight characters will be assigned.

Figure E-2. 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 ***Data Attributes.***

Figure E-3. Data 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] &SYSPARM[1,2]	The assembler	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions
&SYSDATE	The assembler	Current date	CPU Clock	
&SYSTIME	The assembler	Time of day	CPU Clock	

[1] May be used only in macro definitions

[2] DOS/VS only with a null character value-//option

Figure E-4. Variable Symbols

JOB CONTROL STATEMENTS—OS

Figure F-1 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)
 //SYSUT2       DD     DSNAME=&SYSUT2, UNIT=SYSSQ, SPACE=(1700, (400, 50))
 //SYSUT3       DD     DSNAME=SYSUT3, SPACE=(7200, (400, 50)), UNIT=(SYSSQ,
 //                    SEP=(SYSUT2, SYSUT1, SYSLIB))
[4]//SYSPRINT   DD     SYSOUT=A
[5]//SYSPUNCH   DD     SYSOUT=B
      •
      •
      •
 //SYSIN        DD     *
      •
      •
      •
 Program to be assembled
      •
      •
      •
 /*
```

[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] SYSUT1, SYSUT2, and SYSUT3 specify the assembler utility data sets. The device classname SYSSQ represents either a direct access device or a tape drive. The I/O units assigned to the classnames are specified by your installation during system generation. A unit name (for example, 2311) may be substituted for SYSSQ. The DSNAME parameters guarantee use of dedicated work data sets if this feature is supported by your installation.
The SEP= subparameter in statement 5 and the SPACE= parameter in statements 3, 4, and 5 are effective only if SYSSQ is a direct-access device. The space required depends on the source program. The publication *IBM System/360 Operating System: Job Control Language Reference* (GC28-6704) explains space allocation.
[4] This statement defines the standard system output class, SYSOUT=A, as the destination for the assembler listing.
[5] This statement describes the data set that will receive the punched object module.

Figure F-1. Job Control Statements for Assembly Under OS

You may catalog the procedure to simplify your assembly; see the IEBUPDTE Program, in the publication *OS Utilities*, GC28-6586.

OS Assembler Options

You may select the portions of the output desired by specifying them in the PARM= field of the EXEC statement. The options are as follows:

DECK—The object module is placed in the device specified in the SYSPUNCH DD statement.

LOAD—The object module is placed on the device specified in the SYSGO DD statement.

LIST—An assembler listing is produced.

XREF—The assembler produces a cross-reference table of symbols as part of the listing.

RENT—The assembler checks for a possible coding violation of program re-enterability.

LINECNT=nn—This parameter specifies the number of lines to be printed between headings in the listing. The permissible range is 01 to 99 lines.

Use the prefix NO (for example, NOLIST) with the above options to indicate which options are not wanted.

If an option is not specified, the assembler assumes the following default entry:

```
PARM='NOLOAD,DECK,LIST,XREF,LINECNT=55,NORENT'
```

JOB CONTROL STATEMENTS—DOS, DOS/VS

Figure F-2 lists the control cards necessary to assemble a Communications Controller program under DOS or DOS/VS. 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 *DOS System Control and Service* (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 detail see *DOS System Control and Service* (GC24-5036).

Card Group	*Card Arrangement*	*Comments*
Job Control	// JOB	First card in group; always required
	// ASSGN SYSSLB,[1] . .	Required for macros and copy code
	// ASSGN SYSIPT, . .	Source program input
	// ASSGN SYSLST, . .	Program listing
	// ASSGN SYS001, . . .	
	// ASSGN SYS002, . . .	Work files
	// ASSGN SYS003, . . .	
	// 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 or,	Required for DOS } use the
	// EXEC IFZASM	Required for DOS/VS } appropriate statement
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 F-2. Job Control Statements for Assembly Under DOS and DOS/VS

DOS and DOS/VS Assembler Options

You may select those portions of the output desired by specifying them on the option statement. The options are as follows:

DECK—The object module is placed in the device specified in the ASSGN SYSPCH statement.

LINK—The object module is placed on the device specified in the ASSGN SYSLNK statement.

LIST—An assembler listing is produced.

XREF—The assembler produces a cross-reference table of symbols as part of the listing.

DOS/VS only: EDECK—the source macros in the program are punched in edited format on the output device assigned to SYSPCH. This option is used to punch the macros for later cataloging into the F sublibrary of a source statement library.

Use the prefix NO (for example, NOLIST) with the above options to indicate which options are not wanted.

If no options are specified, the assembler assumes the following default entry:

```
// OPTION DECK, NOLINK, LIST, XREF (for DOS)
// OPTION NOEDECK, NOLINK, DECK, LIST, XREF
(for DOS/VS )
```

See the publication, *Guide to the DOS/VS Assembler*, GC33-4024, for more information concerning the assembler options.

JOB CONTROL STATEMENTS—OS/VS

Figure F-3 shows the control statements necessary to assemble a communications controller program under OS/VS.

```
[1]//jobname      JOB    accounting information, MSGLEVEL=1
[2]//stepname     EXEC   PGM=CWAX00,REGION=128K,PARM=(assembler options)
[3]//SYSLIB       DD     DSN=SYS1.MAC3705,DISP=SHR
[4]//SYSUT1       DD     DSN=&&SYSUT1,UNIT=SYSSQ,SPACE=(1700, (600, 100)),
 //                      SEP=(SYSLIB)
 //SYSUT2         DD     DSN=&&SYSUT2,UNIT=SYSSQ,SPACE=(1700, (300,50)),
 //                      SEP=(SYSLIB,SYSUT1)
 //SYSUT3         DD     DSN=&&SYSUT3,UNIT=SYSSQ,SPACE=(1700, (300,50))
[5]//SYSPRINT      DD     SYSOUT=A,DCB=BLKSIZE=1089
[6]//SYSPUNCH     DD     SYSOUT=B
 //SYSIN          DD     *
    •
    •
    •
Program to be assembled
    •
    •
    •
/*
```

[1]This statement names the job.
[2]This statement specifies that the program to be executed is CWAX00, which is the name of the assembler. The REGION parameter specifies the virtual storage region that gives best performance. It is possible to run the assembler in 64K, in which case you must change the region size parameter. You can also add COND and PARM parameters.
[3]This statement identifies the macro library data set. The data set name SYS1.MACLIB is an IBM designation.
[4]SYSUT1, SYSUT2, and SYSUT3 specify the assembler work data sets. The device classname SYSSQ represents either a direct access device or a tape drive. The I/O units assigned to the classnames are specified by your installation during system generation. Instead of a classname you can specify a unit name, such as 2314. The DSN parameters guarantee dedicated work data sets, if this is supported by your installation. The SEP and SPACE parameters are effective only if SYSSQ is a direct access device. The space required depends on the source program.
[5]This statement defines the standard system output class as the destination of the assembler listing. You can specify any blocksize that is a multiple of 121.
[6]This statement describes the data set that will receive the punched object module.

Figure F-3. Job Control Statements for Assembly under OS/VS

OS/VS Assembler Options

Assembler options are functions of the assembler that you, as an assembler language programmer, can select. For example, you can use assembler options to specify whether or not you want the assembler to produce an object deck; whether or not you want it to print certain items in the listing; and whether or not you want it to check your program for reenterability.

The assembler options can be divided into three categories:

- *Listing control options,* which determine the information to be included in the program listing.
- *Output control options,* which specify the device on which the assembler object module is to be written and the contents of the module.
- *Other assembler options,* which specify miscellaneous functions and values for the assembler.

Figure F-4 lists all the assembler options. The underlined values are the standard or default values. These values are used by the assembler for options that you do not specify.

The options fall into two format types:

- Simple pairs of keywords: a positive form (for example, DECK) that requests a function, and an alternative negative form (for example, NODECK) that rejects the function.
- Keywords that permit you to assign a value to a function (for example, LINECOUNT(40)).

How to Specify Assembler Options

You use the PARM field of the EXEC statement calling the assembler to specify the assembler options. Code PARM= followed by a list of options that you have selected. For example,

```
//STEPA  EXEC PGM=CWAX00,PARM='NODECK,FLAG(5),
NORLD'
```

CWAX00 is the name of the assembler; three options are specified for the execution of it. Default values are used for other options.

The PARM field is coded according to the following rules:

- Single quotes or parentheses must surround the entire PARM value if you specify two or more options.
- The options must be separated by commas. You may specify as many options as you wish, and in any order. However, *the length of the option list must not exceed 100 characters*, including separating commas.
- The BUFSIZE, FLAG, LINECOUNT, or SYSPARM options must appear within single quotes.
- If you need to continue the PARM field onto another card, the entire PARM field must be enclosed in parentheses. However, any part of the PARM field enclosed in quotes must not be continued on another card.

The following examples illustrate these rules:

,PARM=DECK	Only one option specified.
,PARM='LINECOUNT(40)'	LINECOUNT, BUFSIZE, FLAG, and SYSPARM must be surrounded by quotes.
,PARM=(DECK,NOOBJECT) or ,PARM='DECK,NOOBJECT'	More than one option specified. None of them requires quotes.
,PARM='DECK,NOLIST,SYSPARM(PARAM)' or ,PARM=(DECK,NOLIST,'SYSPARM(PARAM)') or ,PARM=(DECK,'NOLIST,SYSPARM(PARAM)')	More than one option specified. SYSPARM must appear within quotes.
,PARM= (DECK,NOLIST,'LINECOUNT (35)',NOALIGN, MCALL,'BUFSIZE(MIN)',NORLD)	The whole field must be enclosed by parentheses, because it is continued onto another card. The LINECOUNT and BUFSIZE options must be within quotes, and the portions of the field that are enclosed within quotes cannot be continued onto another card.

Listing Control Options	
ALOGIC	Conditional assembly statements processed in open code are listed.
NOALOGIC	The ALOGIC option is suppressed.
ESD	The external symbol dictionary (ESD) is listed.
NOESD	No ESD listing is printed.
FLAG { (nnn) / (0) }	Diagnostic messages and MNOTE messages below severity code *nnn* will not appear in the listing. Diagnostic messages can have severity codes of 4, 8, 12, 16, or 20 (20 is the most severe), and MNOTE severity codes can be between 0 and 255. For example, FLAG(8) suppresses diagnostic messages with a severity code of 4 and MNOTE messages with severity codes of 0 through 7.
LINECOUNT { (nn) / (55) }	nn specifies the number of lines to be listed per page.
LIST	An assembler listing is produced.
NOLIST	No assembler listing is produced. This option overrides ESD, RLD, and XREF.
MCALL	Inner macro instructions encountered during macro generation are listed following their respective outer macro instructions. The assembler assigns statement numbers to these instructions. The MCALL option is implied by the MLOGIC option; NOMCALL has no effect if MLOGIC is specified.
NOMCALL	The MCALL option is suppressed.
MLOGIC	All statements of a macro definition processed during macro generation are listed after the macro instruction. The assembler assigns statement numbers to them.
NOMLOGIC	The MLOGIC option is suppressed.
RLD	The assembler produces the relocation dictionary as part of the listing.
NORLD	The RLD is not printed.
LIBMAC	The macro definitions read from the macro libraries and any assembler statements following the logical END statement are listed after the logical END statement. The logical END statement is the first END statement processed during macro generation. It may appear in a macro or in open code; it may even be created by substitution. The assembler assigns statement numbers to the statements that follow the logical END statement.

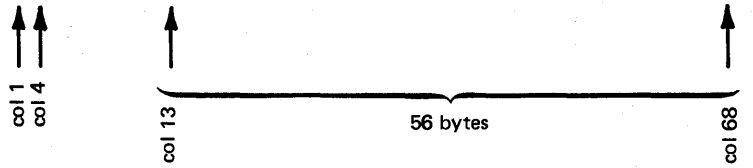
Figure F-4. The Assembler Options (OS/VS) (Part 1 of 2)

Listing Control Options (continued)	
NOLIBMAC	The LIBMAC option is suppressed.
XREF (FULL)	The assembler listing will contain a cross reference table of all symbols used in the assembly. This includes symbols that are defined but never referenced.
XREF (SHORT)	The assembler listing will contain a cross reference table of all symbols that are referenced in the assembly. Any symbols defined but not referenced are not included in the table.
NOXREF	No cross reference tables are printed.
Output Control Options	
DECK	The object module is written on the device specified in the SYSPUNCH DD statement.
NODECK	The DECK option is suppressed.
Other Assembler Options	
BUFSIZE (MIN)	The assembler uses the minimum buffer size (790 bytes) for each of the utility data sets (SYSUT1, SYSUT2, and SYSUT3). Storage normally used for buffers is allocated to work space. Because more work space is available, more complex programs can be assembled in a given region; but the speed of the assembly is substantially reduced.
BUFSIZE (STD)	The buffer size that gives optimum performance is chosen. The buffer size depends on the size of the region or partition. Of the assembler working storage in excess of minimum requirements, 37% is allocated to the utility data set buffers, and the rest to macro generation dictionaries. See *OS/VS Storage Requirements* later in this Appendix for a more complete description of the effects of BUFSIZE.
RENT	The assembler checks your program for a possible violation of program reenterability. Code that makes your program non-reentrant is identified by an error message.
NORENT	The RENT option is suppressed.
SYSPARM { (string) / () }	'string' is the value assigned to the system variable symbol &SYSPARM. Due to JCL restrictions, you cannot specify a SYSPARM value longer than 56 characters (as explained in Note 1 following this figure). Two quotes are needed to represent a single quote, and two ampersands to represent a single ampersand. For example, PARM='OBJECT,SYSPARM((&&AM,''BO).FY)' assigns the following value to &SYSPARM: (&AM,'BO).FY. Any parentheses inside the string must be paired. If you call the assembler from a problem program (dynamic invocation), SYSPARM can be up to 256 characters long.

Figure F-4. The Assembler Options (OS/VS) (Part 2 of 2)

Note 1: The restrictions imposed upon the PARM field limit the maximum length of the SYSPARM value to 56 characters. Consider the following example:

```
// EXEC ASMFC,PARM.ASM= (OBJECT,NODECK,
// 'SYSPARM (ABCD ..........................................)')
```



Since SYSPARM uses parentheses, it must be surrounded by quotes. Thus, it cannot be continued onto a continuation card. The leftmost column that can be used is column 4 on a continuation card. A quote and the keyword must appear on that line as well as the closing quotes. In addition, either a right parenthesis, indicating the end of the PARM field, or a comma, indicating that the PARM field is continued on the next card, must be coded before or in the last column of the statement field (column 71).

Note 2: Even though the formats of some of the options previously supported by the OS version of the controller assembler have been changed, you can use the old formats for the following options: ALGN (now ALIGN), NOALGN (NOALIGN), LINECNT=nn (LINECOUNT (nn)),LOAD (OBJECT), and NOLOAD (NOOBJECT). This support may not be continued indefinitely, so you should change to the new options as soon as possible.

Dynamic Invocation of the Assembler (OS/VS)

You can invoke the assembler from your problem program when it is executed, by using the CALL, LINK, XCTL, or ATTACH macro instruction. If you use the XCTL instruction, you cannot specify any assembler options. The assembler will use the standard or default options. If you use CALL, LINK, or ATTACH, you can specify both the assembler options and DD names of the data sets to be used by the assembler. The formats of these macros are:

Name	*Operation*	*Operand*
[symbol]	CALL	CWAX00, (optionlist [,ddnamelist]),VL
	{ LINK / ATTACH }	EP=CWAX00, PARAM=(optionlist [,ddnamelist]),VL=1

EP–specifies the symbolic name of the assembler (CWAX00).

PARAM–specifies, as a sublist, address parameters to be passed from the problem program to the assembler. The first word in the address parameter list contains the address of the option list. The second word contains the address of the ddname list.

optionlist–specifies the address of a variable length list containing the options. This address must be written even if no option list is provided.

The option list must begin on a halfword boundary. The first two bytes contain a count of the number of bytes in the remainder of the list. If no options are specified, the count must be zero. The option list is free form with each field separated from the next by a comma. No blanks or zeros should appear in the li[illegible]

ddnamelist–specifies the address of a variable length list containing alternate ddnames for the data sets used during assembler processing. If standard ddnames are used, this operand can be omitted.

The ddname list must begin on a halfword boundary. The first two bytes contain a count of the number of bytes in the remainder of the list. Each name of less than eight bytes must be left-justified and padded with blanks. If an alternate ddname is omitted, the standard name will be assumed. If the name is omitted within the list, the eight-byte entry must contain binary zeros. Names can be omitted from the end merely by shortening the list. The sequence of the eight-byte entries in the ddname list is as follows:

Entry	*Standard Name*
1	not applicable
2	not applicable
3	not applicable
4	SYSLIB
5	SYSIN
6	SYSPRINT
7	SYSPUNCH
8	SYSUT1
9	SYSUT2
10	SYSUT3

VL–specifies that the high-order bit is to be set to 1 in the last word of the list of address parameters in the macro expansion. The assembler checks this bit to find out if a ddname list is specified or not.

Note: If you invoke the assembler more than once from the same program, make sure that RECFM=S is not specified for the SYSPRINT data set.

ASSEMBLER STORAGE REQUIREMENTS

OS and OS/VS Storage Requirements

The primary storage requirement for the assembler under OS when operating in an MFT partition is a minimum of 48K bytes. The assembler under OS requires a minimum of 50K bytes when operating in an MVT region. The minimum virtual storage partition or region under OS/VS1 or OS/VS2 is 64K bytes. However, better performance is generally achieved if a partition or region of 128K bytes is used.

Assembler Data Set Characteristics and Buffer Sizes

If more storage is allocated to the assembler, the size of buffers and work space can be increased. The amount of storage allocated to buffers and work space determines assembler speed and capacity. Generally, as more storage is allocated to buffers, a given assembly will run faster; as more storage is allocated to work space, larger and more complex macro definitions can be handled.

You can control the buffer sizes of SYSIN, SYSLIB, SYSPRINT, SYSPUNCH by specifying the blocksize (BLKSIZE) and number of buffers (BUFNO) as shown in Figure F-5.

You can control the buffer sizes for the assembler utility data sets (SYSUT1, SYSUT2, and SYSUT3) and the size of the work space used during macro processing, by specifying the BUFSIZE assembler option. Of the storage given to the assembler, the assembler first allocates storage for the SYSIN and SYSLIB buffers according to the specifications in the DD statements or the labels of the data sets. It then allocates storage for the modules of the assembler. The remainder of the partition or region is allocated to utility data set buffers and macro generation dictionaries according to the BUFSIZE option specified:

BUFSIZE(STD): 37% is allocated to buffers, and 63% to work space. This is the default chosen, if you do not specify any BUFSIZE option.

BUFSIZE(MIN): Each utility data set is allocated a single 790-byte buffer. The remaining storage is allocated to work space. This allows relatively complex macro definitions to be processed in a given region or partition size, but the speed of the assembly is substantially reduced.

	SYSIN	SYSLIB	SYSPRINT	SYSPUNCH	SYSUT1 SYSUT2 SYSUT3
LRECL	Fixed at 80	Fixed at 80	Fixed at 121	Fixed at 80	Not applicable
RECFM (Note 1)	You must specify in LABEL or DD card: F,FS,FBS,FB, FBST,FBT	You must specify in LABEL or DD card: F,FS,FBS,FB, FBST,FBT	F and A set by assembler. B set by assembler except when F is specified and BLKSIZE is not specified. You may add S or T: FA,FAB,FAS,FAT, FABS,FABT	F set by assembler, you may specify B and/or T in label or DD card F,FB,FT,FBT	Set by assembler to U
BLKSIZE (Note 2)	You must specify in LABEL or DD card; must be a multiple of LRECL	You must specify in LABEL or DD card; must be a multiple of LRECL	Optional, but must be a multiple of LRECL; if omitted, BLKSIZE=LRECL	Optional, but must be a multiple of LRECL; if omitted, BLKSIZE=LRECL	If BUFSIZE (STD) in effect, a value between 790 and 8192 is chosen. If BUFSIZE (MIN) in effect, 790 is chosen
BUFNO	Optional; if omitted 2 is used	Set by assembler to 1	Optional; if omitted 2 is used	Optional; if omitted 3 is used for unit record and 2 for other devices	Set by assembler to either 1 or 2

Note 1: U = undefined, F = fixed length records, B= blocked records, S= standard blocks, T = track overflow, A = ASCII code carriage control

Note 2: Blocking is not allowed on unit record devices. Blocking on other direct access devices cannot exceed the track size unless T is specified on RECFM. If the BLKSIZE specified is not a multiple of LRECL, the assembler truncates it to a multiple. For example, if LRECL = 80, a BLKSIZE of 850 is truncated to 800.

Figure F-5. Assembler Data Set Characteristics–OS/VS

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 shown in Figure F-6.

Dictionary Capacities

The capacity of the general dictionary (global dictionay 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 storage depends upon storage availability. The minimum 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 storage for the assembler (the global and local dictionaries together can occupy up to 64K bytes).
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 storage to the assembler and reassemble your program.

The assembler can usually handle 400 ordinary symbols without overflow in its minimum main storage. The assembler can process one additional symbol for each 18 bytes above minimum storage.

DOS Storage Requirements

The primary storage requirement for the assembler under DOS is a minimum of 12K bytes.

Auxiliary Storage Requirements

The auxiliary storage requirements are as follows:

- Residence requirements:

Device Type	Blocks Needed
Core Image Library:	
2311	46
2314	23
Relocatable Library:	
2311	68
2314	40

- Work file requirements:

The number of bytes per statement is as follows:

SYSLNK: 15
SYS001: 150
SYS002: 150
SYS003: 36

To determine the approximate member of tracks required, multiply the number of statements by the values directly above, then divide 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.

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 that 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
4500/300 = 1.5 = 2 tracks

SYS003:

36(10) + 36(20) = 36(30) = 1080
1080/3000 = .36 = 1 track

DOS/VS Storage Requirements

The primary storage requirement for the assembler under DOS/VS is a minimum of 20K bytes.

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 F-6. Work Space for Assembly Under OS and OS/VS

Auxiliary Storage Requirements

The auxiliary storage requirements are as follows:

- Residence Requirements:

Device Type	Tracks Needed
Core Image Library:	
2314	20
2319	20
3330	11
Relocatable Library:	
2314	38
2319	38
3330	23

- Work File Requirements:

SYS001: MAX (60 x ITXT + 60, SM + 60 x LM)
SYS002: MAX (40 x ETXT, 60 x ITXT + 60 x SM)
SYS003: 60 x OTXT if option NOXREF
100 x OTXT if option XREF

where:
ITXT= Total number of statements on SYSIPT
OTXT= Total number of statements on SYSLST (with PRINT GEN)
SM= Number of source macro statements
ETXT= (OTXT–number of comments–SM)
LM= Number of statements in library macros used by the program.

Appendix G: Communications Controller Assembler Messages—OS and DOS

Component Name	IFK = OS IFT = DOS
Program Producing Message	IBM Communications Controller Assembler program during assembly of assembler instructions (under OS or DOS)
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; successful execution is possible 12 Serious error; successful execution is improbable 16 Critical error; successful execution is impossible 20 Critical error; further assembly impossible; assembler program terminated abnormally nnn Message serial number text Message text xx Message reply identification (absent, if operator reply not required)

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 *Defining Data* 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 and 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 Instruction* under *Addressing,* 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.

Severity 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 2.)

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 Instruction* under *Addressing* 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 Instruction* under *Defining Data* 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 *Controlling the Assembler Program* in Chapter 4.)

IFK031 UNKNOWN TYPE
IFT031

Explanation: Incorrect type designation is specified in a DC or DS instruction.

Severity Code: 8

Programmer Response: Ensure that the type designations specified in a DC or DS instruction 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 invalid 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 than 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 Instruction* under *Assigning Values to SET Symbols* 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 *The Conditional Assembly Language* 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 *The Macro Definition,* also in Chapter 5. Make sure source statements are 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 storage 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.

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 *The Conditional Assembly Language* 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 equestion 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 *Data Attributes* under *The Conditional Assembly Language* 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 Instruction* under *Branching* 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 statements are 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 Instruction* 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 Instruction,* in Chapter 5.)

IFK065 INVALID OR ILLEGAL TERM IN ARITHMETIC EXPRESSION
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 SETC 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 EXPRESSION 1 OF 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 Instruction,* in Chapter 5.)

IFK068 GENERATION TIME DICTIONARY AREA OVERFLOWED
IFT068

Explanation: Not enough storage allocated to the assembler; (or, 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 EXPRESSION 2 OF 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 OF 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 sequence-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 Instruction* under *Controlling the Assembler Program* 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 Instruction* under *Controlling the Assembler Program* 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.)

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 Instruction* 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 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 INSTRUCTION OR PROTOTYPE OPERAND EXCEEDS 255 CHARACTERS
IFT096 MACRO INSTRUCTION OR PROTOTYPE 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. Invalid "=".
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 prototype 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. Invalid (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 INSTRUCTION OPERAND, PROTOTYPE PARAMETER, EXTRA COMMA FOLLOWS 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.

IFK100 STATEMENT COMPLEXITY 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 OR SYSIPT

Explanation: EOD card was encountered 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 instruction 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 instruction is the first statement in the input deck and that the operands are in the proper range. (See *ICTL Instruction* 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 invalid 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 card was encountered before MEND card; MEND statement missing from macro definition; COPY code not found while editing a macro; macro definition is truncated; or EOF was encountered while reading a macro or copy code.

Severity Code: 12

Programmer Response: Probable user error. Make sure source program 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, DC instructions; 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 examples of what may be incorrect. (See *DC Instruction* 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 EXPRESSION VALUE TOO 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 NOLOAD 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 OR SYSxxx

Explanation: EOF (end of file) condition has occurred 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. Probable user error.

System Response: The job step is terminated.

Programmer Response: If the problem recurs, have the system log, printer output, and the job stream 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 OPTION 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. Probable 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 ASSGN statement 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 cuased 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. Probable 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) is 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. Probable 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 filed.

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.
2. 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.)
IFT116

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 multi-term expression.

Programmer Response: Replace the invalidly specified symbolic register expression with the appropriate absolute, relocatable or complex relocatable expression required for reassemble. See *EQUR Instruction* 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, printed 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, printed 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.

Appendix H: Communications Controller Assembler Messages: DOS/VS

Component Name	IFZ = DOS/VS
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, ***IFZnnn text (in SYSPRINT) xx IFZnnn 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; successful execution is possible 12 Serious error; successful execution is improbable 16 Critical error; successful execution is impossible 20 Critical error; further assembly impossible; assembler program terminated abnormally nnn Message serial number text Message text xx Message reply identification (absent, if operator reply not required)

IFZ001 END STATEMENT IN MACRO OR COPY CODE

Explanation: An END statement is found in a macro definition or in code that is inserted by means of the COPY instruction.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the END statement from the macro definition or the copy book. Make sure that an END statement is included at the end of your source module.

IFZ002 ICTL NOT FIRST STATEMENT

Explanation: The ICTL statement is used in a statement that is not the first statement in the source module.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the ICTL statement, or make it the first statement of the program.

IFZ003 STATEMENT INCORRECTLY PLACED, MUST BE IN MACRO DEFINITION

Explanation: A MEND, MEXIT, MNOTE, or internal macro comments (.*) statement appears in open code. These statements are allowed only in macro definitions.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the statement, or put it in a macro definition.

IFZ004 COMMENTS BETWEEN MACRO AND PROTOTYPE STATEMENTS

Explanation: The macro header (MACRO) instruction is followed by a comments statement (.* or *). The macro header must be immediately followed by a macro prototype statement.

Assembler Action: The comments statement is ignored. It is not generated when the macro is generated.

Programmer Response: Put the comments statement after the prototype statement.

IFZ005 STATEMENT INCORRECTLY PLACED

Explanation: One of the following errors has occurred:

- A macro header (MACRO) instruction appears too late in the program. It can only be used to identify the beginning

of a macro definition, and the macro definitions must all be placed at the beginning of the source module. The only instructions that can precede them are: ICTL, ISEQ, EJECT, PRINT, TITLE, SPACE, and comments statements.

- A GBLx or LCLx instruction in the macro definition does not immediately follow the macro prototype statement.
- A GBLx instruction is preceded by an LCLx instruction.
- A GBLx or LCLx instruction in open code does not precede the first control section.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure your MACRO, GBLx, and LCLx, instructions are placed according to the rules given in the explanation.

IFZ006 ILLEGAL NAME FIELD

Explanation: The name field is not a sequence symbol or blank, which is required by this instruction.

Assembler Action: The name field is ignored.

Programmer Response: Make sure the name field is either a sequence symbol or blank.

IFZ007 SOURCE RECORD OUT OF SEQUENCE

Explanation: The input sequence-checking specified by the ISEQ instruction has determined that this record is out of sequence. The sequence field of this record is not higher than the sequence field of the preceding record.

Assembler Action: The statement is flagged and assembled. The sequence of the rest of the statement is checked relative to the sequence of the statement before this statement.

Programmer Response: Put the record in the proper sequence.

IFZ008 UNPAIRED APOSTROPHE

Explanation: An ending apostrophe is missing in this statement, or an invalid attribute reference is found in the statement.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a terminating apostrophe or correct the attribute reference. An opening or ending apostrophe must be single, that is, it must be immediately followed or preceded by another single apostrophe. Double apostrophes are used to specify the character in a quoted string (between the opening and terminating apostrophes).

IFZ009 TOO MANY CONTINUATION LINES

Explanation: This statement occupies more than three records.

Assembler Action: The excessive continuation lines are treated as comments.

Programmer Response: Check for an unintentional continuation indicator in the column after the end column (usually in column 72). Do not use more than two continuation lines for a statement.

IFZ010 OP CODE MISSING

Explanation: The first or only record of a statement does not contain any operation code, followed by at least one blank.

Assembler Action: The statement is processed as comments.

Programmer Response: If this record is intended to be a comments statement, supply an asterisk in the begin column. If the record is intended to be an instruction, supply an op code followed by at least 1 blank in the first record of the statement.

IFZ011 INVALID OP CODE

Explanation: The specified operation code does not consist of 1-8 alphameric characters, the first of which is alphabetic.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the operation code is a valid ordinary symbol as described in the explanation.

IFZ012 MEND NOT PRECEDED BY MACRO IN THIS COPY BOOK

Explanation: In code inserted by means of the COPY instruction, a MEND instruction is encountered for which there is no corresponding MACRO instruction in this copy book.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure that a macro always starts and ends in the same copy book. If a MACRO statement is found in a copy book, the corresponding MEND statement must also be in that copy book.

IFZ013 CONTINUATION LINE MISSING

Explanation: End of file condition was encountered when the assembler was trying to read an expected continuation line.

Assembler Action: The statement is processed as if no continuation mark has been indicated in the continuation column.

Programmer Response: Add the missing continuation line(s), or remove the erroneous continuation mark, whichever is applicable.

IFZ014 SYMBOLIC PARAMETER 'xxxxxxxx' TOO LONG

Explanation: The specified symbolic parameter in a macro prototype statement is too long. It must not consist of more than eight characters. The first eight characters of the invalid symbolic parameter are identified in the message.

Assembler Action: The rest of the macro definition is checked for errors, but the macro is considered undefined.

Programmer Response: Make sure that all symbolic parameters consist of an ampersand followed by 1 to 7 alphameric characters, the first of which is alphabetic.

IFZ015 SYMBOLIC PARAMETER 'xxxxxxxx' DOES NOT START WITH AMPERSAND

Explanation: The specified symbolic parameter does not start with an ampersand (&).

Assembler Action: The rest of the macro definition is checked for errors, but the macro is considered undefined.

Programmer Response: Make sure that all symbolic parameters consist of an ampersand followed by 1-7 alphameric characters, the first of which is alphabetic.

IFZ016 SECOND CHARACTER OF SYMBOLIC PARAMETER 'xxxxxxxx' NOT A LETTER

Explanation: The second character of the specified symbolic parameter is not alphabetic.

Assembler Action: The rest of the macro definition is checked for errors, but the macro is considered undefined.

Programmer Response: Make sure that all symbolic parameters consist of an ampersand followed by 1-7 alphameric characters, the first of which is alphabetic.

IFZ017 SYMBOLIC PARAMETER 'xxxxxxxx' CONTAINS NON-ALPHAMERIC CHARACTER

Explanation: The specified symbolic parameter contains an invalid character. Only alphameric characters (A through Z, @, #, $, 0 through 9) are allowed in symbolic parameters.

Assembler Action: The rest of the macro definition is checked for errors, but the macro is considered undefined.

Programmer Response: Make sure that all symbolic parameters consist of an ampersand followed by 1-7 alphameric characters, the first of which is alphabetic.

IFZ018 INVALID OPCODE IN PROTOTYPE STATEMENT

Explanation: The mnemonic operation code of a prototype statement is (a) not a valid symbol, (b) is the same as the opcode of another macro definition in the source program, (c) is the same as the opcode of a machine instruction or assembler instruction.

Assembler Action: The macro definition will be checked for errors just as if the opcode were correct; but when the macro is called, it is treated as undefined.

Programmer Response: Make sure that the prototype opcode consists of 1-8 alphameric characters starting with an alphabetic character, and that the prototype opcode is different from other prototype, machine, and assembler opcodes.

IFZ019 KEYWORD OPERAND PRECEDES POSITIONAL OPERAND 'xxxxxxxx'

Explanation: In a macro prototype statement or a macro definition, a keyword operand has been placed before the positional operand identified in the message. All positional operands must appear before the keyword operands in the statement. If no operand is identified in the message, a comma indicating an omitted positional operand has been found after the first keyword operand.

Assembler Action: If the error is found in a prototype statement, all positional operands after the first keyword operand are considered undefined. The rest of the macro definition is then checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated.

Programmer Response: Make sure all positional operands in a macro prototype statement or macro instruction precede all keyword operands.

IFZ020 TOO MANY LEVELS OF PARENTHESES IN OPERAND 'xxxxxxxx'

Explanation: The operand expression identified in the message contains more than five levels of parentheses. The text inserted in the message is limited to eight characters.

Assembler Action: If the error is found in a prototype statement, the rest of the macro definition is checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated.

Programmer Response: Change the expression to delete one or more levels of parentheses.

IFZ021 UNPAIRED PARENTHESES IN OPERAND 'xxxxxxxx'

Explanation: The keyword parameter default value specified in a macro prototype or a macro instruction operand value contains an unpaired left or right parentheses not surrounded by apostrophes. Only the first eight characters of the operand value are inserted in the message.

Assembler Action: If the error is found in a prototype statement, the rest of the macro definition is checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated.

Programmer Response: If you want to specify an unpaired parentheses, make sure it appears with apostrophes. Otherwise make sure a left parenthesis is always followed by a right parenthesis with which it is paired.

IFZ022 INVALID SUBLIST 'xxxxxxxx' IN ALTERNATE STATEMENT FORMAT

Explanation: The termination of a macro prototype or macro instruction sublist written in the alternate statement format for sublists is invalid, either because the closing right parenthesis is missing, or because something other than a comma or a blank follows the closing right parenthesis; only the first eight characters of the sublist are inserted in the message list.

Assembler Action: If the error is found in a prototype statement, the rest of the macro is checked for errors, but the macro is considered undefined. If the error occurs in a macro instruction, the macro is not generated.

Programmer Response: If a sublist is intended, make sure that the sublist is terminated by a right parenthesis followed by a comma or a blank. If a character string is intended, use the normal statement format instead.

IFZ023 PARAMETER VALUE 'xxxxxxxx' EXCEEDS 255 CHARACTERS

Explanation: The specified value is too long. The parameter value specified in a macro prototype statement (as a keyword parameter default value) or a macro instruction is limited to 255 characters. The text inserted in the message contains only the first eight characters.

Assembler Action: If the error is found in a macro prototype statement, the rest of the macro definition is checked for errors, but the macro is considered undefined. If the error is found in the macro instruction, the macro is not generated.

Programmer Response: Limit the length of the parameter to 255 characters, or separate the value into two or more parameters.

IFZ024 UNPAIRED APOSTROPHE

Explanation: An unpaired apostrophe is found in a parameter value specified in a macro prototype statement (as a keyword parameter default value) or a macro instruction. Single apostrophes in parameter values must be specified with double apostrophes appearing inside paired apostrophes, unless they are used to specify attribute references in arithmetic expressions.

Assembler Action: If the error is found in a macro prototype statement, the rest of the macro is checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated.

Programmer Response: Make sure all apostrophes are paired or double, or belong to attribute references.

IFZ025 TOO MANY OPERANDS

Explanation: Too many operands found in a macro prototype statement or a macro instruction or too many sub-operands in a sublist. The maximum number allowed is 200.

Assembler Action: If the error is found in a macro prototype statement, the rest of the macro definition is checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated. Only the first eight characters of the default value are inserted in the message.

Programmer Response: Reduce the number of operands or include some of the operands in sublists or, if too many sub-operands, split the sublist into two or more.

IFZ026 INVALID NAME FIELD 'xxxxxxxx'

Explanation: The name field of a macro prototype statement or a macro instruction is invalid. The name field of a prototype statement must either be blank or contain a variable symbol specifying a name field parameter. The name field of a macro instruction must either be blank, or contain a sequence symbol, or a valid ordinary symbol, or one or more variable symbols that result in a valid ordinary symbol after substitution and concatention. Only the first eight characters of the default value are inserted in the message.

Assembler Action: If the error is found in a macro prototype statement, the rest of the macro definition is checked for errors, but the macro is considered undefined. If the error is found in a macro instruction, the macro is not generated.

Programmer Response: Supply a valid name field as described in the explanation.

IFZ027 NON-BLANK CHARACTER FOUND BEFORE CONTINUE COLUMN

Explanation: On a continuation record, that is, a record following after the first record of a statement occupying several records (lines), one or more characters have been encountered in the begin column or in the column between the begin column (usually column 1) and the continue column (usually column 16). These columns must be blank.

Assembler Action: The characters appearing before the continue column are ignored.

Programmer Response: If the record is intended as a continuation record, make sure the statement is continued in the correct column. If the record is not meant to be a continue record, check for an unintentional continuation indicator in the preceding record (usually in column 72).

IFZ028 INVALID KEYWORD PARAMETER DEFAULT VALUE 'xxxxxxxx'

Explanation: The default value specified for a keyword parameter in a macro prototype statement is invalid. The value must not contain variable symbols, and any ampersands must be double, that is, each sequence of consecutive ampersands must contain an even number of ampersands. Only the first eight characters of the default value are inserted in the message.

Assembler Action: The rest of the macro definition is checked for errors, but the macro is considered undefined.

Programmer Response: Delete variable symbols from the default value; make ampersands double.

IFZ029 INVALID KEYWORD IN MACRO INSTRUCTION, 'xxxxxxxx'

Explanation: A keyword of a macro instruction does not consist of 1-7 alphameric characters, the first of which is alphabetic, or a macro instruction operand contains an equal sign outside quotes or parentheses.

Assembler Action: The rest of the macro is checked for errors, but the macro is considered undefined.

Programmer Response: Make sure that all keywords consist of a letter followed by 0-6 alphameric characters.

IFZ031 NAME FIELD NOT BLANK

Explanation: The name field is not blank, which is required by this instruction.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the statement from the macro definition. Make sure all your macro definitions end with a MEND instruction.

IFZ032 STATEMENT INCORRECTLY PLACED, MUST NOT BE IN MACRO DEFINITION

Explanation: A statement has been found in a macro definition which is not allowed to appear in a macro definition.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the statement from the macro definition. Make sure all your macro definitions end with a MEND instruction.

IFZ033 INVALID ISEQ OR ICTL OPERAND

Explanation: One of the following errors has occurred:

- The operand field of an ISEQ instruction is invalid. It must either be a blank or consist of two decimal self-defining terms that do not fall between the begin and end columns, and the first value must not be greater than the second.

- The operand field of the ICTL statement is invalid. It must consist of one to three decimal self-defining terms, the first of which must be in the range 1-40, the second in the range 41-80, and the third must be in the range 2-40 and greater than the first.

Assembler Action: The statement is processed as comments.

Programmer Response: Correct the operand field according to the rules given in the explanation.

IFZ034 INVALID COPY OPERAND

Explanation: The operand of a COPY instruction is not an ordinary symbol.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid ordinary symbol that corresponds to the name of a book in the copy code library. Ordinary symbols consist of 1-8 alphameric characters, the first of which is alphabetic.

IFZ035 TOO MANY COPY NEST LEVELS

Explanation: More than three nesting levels of COPY instructions have been coded. Nesting occurs when a COPY instruction is coded in a book that is inserted by means of another COPY instruction.

Assembler Action: The last COPY instruction is processed as comments.

Programmer Response: Reduce the number of nesting levels by including some of the COPY books physically in the source module.

IFZ036 COPY BOOK NOT IN LIBRARY

Explanation: The ordinary symbol specified in the operand of this COPY instruction is not the name of a copy book in the source statement library that is assigned to this job.

Assembler Action: The statement is processed as comments.

Programmer Response: Check that the operand is correct, assign the proper source statement library, or catalog the missing copy book.

IFZ037 UNEXPECTED END-OF-FILE ON SYSSLB

Explanation: End-of-file was encountered in the source statement library before the end of a book has been reached. Since the end-of-file indicator is normally found only at the end of the COPY code library, the message indicates that the source statement library has been destroyed.

Assembler Action: Processing of the copy book is terminated. If the error occurs inside a source macro definition, a MEND instruction is generated.

Programmer Response: Reconstruct the source statement library.

IFZ038 MEND STATEMENT MISSING, HAS BEEN ADDED

Explanation: End-of-file occurred on SYSIPT during the processing of a macro definition, or a MEND instruction terminating a macro definition is missing.

Assembler Action: A MEND and an END instruction are inserted.

Programmer Response: Insert the missing MEND instruction or check for an unintentional end-of-file indicator in the source module.

IFZ039 END STATEMENT NOT IMMEDIATELY FOLLOWED BY END-OF-FILE

Explanation: The END statement identifying the end of the source module is not immediately followed by an end-of-data indicator statement (/*).

Assembler Action: The records appearing between the END statement and the end-of-data indicator are not processed by the assembler.

Programmer Response: Move the END statement, or make sure your JCL statements are properly placed.

IFZ040 END STATEMENT MISSING, HAS BEEN ADDED

Explanation: No END statement was found in the source module.

Assembler Action: An END statement is inserted at the end of the input.

Programmer Response: Supply an END statement at the end of your source module, or make sure that no end-of-data indicator (/*) has been placed inside your source module.

IFZ041 MEND STATEMENT MISSING IN COPY BOOK, HAS BEEN ADDED

Explanation: A source macro definition was coded in a copy book, but the macro trailer (MEND) statement to indicate the end of the macro definition was not found in the copy book. The whole macro definition must be coded within one copy book.

Assembler Action: The MEND instruction is inserted at the end of the copy book.

Programmer Response: Make sure that a macro always starts and ends in the same copy book. If a MACRO statement is found in a copy book, the corresponding MEND statement must also be in that copy book.

IFZ042 STATEMENT COMPLEXITY EXCEEDED

Explanation: A conditional assembly statement of a macro instruction operand has more than 50 variable symbols.

Assembler Action: A conditional assembly statement is treated as comments. A macro instruction operand in error will stop the generation of the macro.

Programmer Response: Do not use more than 50 variable symbol references in the same statement or a macro instruction operand.

IFZ043 OPERAND MISSING

Explanation: This statement requires an operand, but none is found.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid operand.

IFZ044 INVALID SYNTAX IN SET SYMBOL DECLARATION 'xxxxxxxx'

Explanation: In a SET symbol declaration, a variable symbol is invalid, a comma separating two symbols is missing, or a character other than a blank terminates the field. The text inserted in the message gives eight characters, starting with the character at which the error is found.

Assembler Action: The symbol in which the error is found and the rest of the statements are ignored.

Programmer Response: Make sure the operand field contains only valid variable symbols (possibly dimensioned), separated by commas.

IFZ045 INVALID DIMENSION 'xxxxxxxx'

Explanation: The dimension of a SET symbol is incorrectly specified. The dimension specification must follow immediately after the variable symbol and be an unsigned decimal value in the range 1-255 enclosed in parentheses.

Assembler Action: The symbol with the invalid dimension and the rest of the statements are ignored.

Programmer Response: Correct the subscript according to the rules given in the explanation.

IFZ046 DIMENSION TOO LARGE, 'xxxxxxxx'

Explanation: A SET symbol declaration specifies a dimension that is greater than 255. The string inserted in the message contains up to eight characters, starting with the dimension value.

Assembler Action: The symbol with the invalid dimension is ignored.

Programmer Response: Break up the SET symbol array into two or more arrays by using additional SET symbols.

IFZ047 VARIABLE SYMBOL DUPLICATES SYSTEM VARIABLE SYMBOL OR PREVIOUS DEFINITION, 'xxxxxxxx'

Explanation: The first or only variable symbol in the specified string is either:

- a symbolic parameter, which is identical to a system variable symbol or another symbolic parameter specified in the same macro prototype statement; or
- a SET symbol, which is identical to a system variable symbol, a symbolic parameter specified in the same macro definition, or another SET symbol declared in the same macro definition or open code.

Assembler Action: The flagged definition of the variable symbol is ignored, as well as any further operands in the statement. All references to the symbol are treated as references to the first definition of the variable.

Programmer Response: Make sure that all variable symbols within a macro definition or open code are unique within that scope. Do not define system variable symbols or symbolic parameters or SET symbols. The system variable symbols are:

&SYSECT	&SYSLIST
&SYSNDX	&SYSPARM

IFZ048 INVALID SYNTAX IN CONDITIONAL ASSEMBLY STATEMENT 'xxxxxxxx'

Explanation: A conditional assembly statement or a statement with variable symbol substitution contains a syntax error, for example:

- Invalid or misplaced characters in an expression.
- The statement is terminated before its logical end. This could be caused by an unintentional blank inside an expression.
- The sequence symbol in an AGO or AIF operand does not consist of a period, followed by a letter and 1-6 letters or digits; or both. The string in the message contains up to eight characters starting where an error is found.

Assembler Action: The statement is processed as comments.

Programmer Response: The first character of the string in the message tells you where the syntax error was found. Correct the error.

IFZ049 'xxxxxxxx' IS AN INVALID VARIABLE SYMBOL

Explanation: The specified variable symbol does not consist of ampersand followed by 1-7 alphameric characters the first of which is alphabetic.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid variable symbol.

IFZ050 INVALID ATTRIBUTE REFERENCE 'xxxxxxxx'

Explanation: The attribute reference is invalid for the type of attribute in this context; for example:

- A reference inside a macro definition refers to an ordinary symbol
- An attribute reference refers to a SET symbol
- A K or N attribute reference refers to an ordinary symbol.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the attribute reference is correct, that this type or attribute can refer to this type of symbol, that the reference is properly placed, etc.

IFZ051 INCORRECT VARIABLE SYMBOL IN NAME FIELD

Explanation:

- This symbol is declared to be of a type different from the type specified by the operation code in this statement; or
- a system variable symbol or symbolic parameter appears in the name field of the SETx instruction.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the declaration is correct, or change the operation code of this statement. Do not use system variable symbols in the name field of SETx instructions.

IFZ052 NAME FIELD MISSING

Explanation: This statement requires a name field, but none is found.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the proper symbol in the name field.

IFZ053 NAME FIELD NOT A SEQUENCE SYMBOL

Explanation: This statement requires a sequence symbol in the name field, but no valid sequence symbol is found.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid sequence symbol.

IFZ054 INVALID NAME FIELD, MUST NOT CONTAIN SEQUENCE SYMBOL OR BLANK

Explanation: The name field of this statement does not contain an ordinary symbol or one or more variable symbols that result in a valid ordinary symbol.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid ordinary symbol, or make sure that the result of variable symbol substitution and concatenation is a valid ordinary symbol.

IFZ055 UNPAIRED LEFT PARENTHESIS

Explanation: A left parenthesis in this statement does not have a corresponding right parenthesis.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the missing right parenthesis, or delete the superfluous left parenthesis.

IFZ056 TOO MANY LEVELS OF PARENTHESES

Explanation: This expression has more than five levels of parentheses.

Assembler Action: The statement is processed as comments.

Programmer Response: Reduce the number of levels of parentheses. Use additional SETA instructions, if necessary.

IFZ057 COUNT OR NUMBER ATTRIBUTE IN OPEN CODE

Explanation: A count (K') or number (N') attribute has been encountered in open code. These attributes can only appear in macro definitions.

Assembler Action: The statement is processed as comments.

Programmer Response: Do not use the count or number attribute in open code.

IFZ058 INVALID SUBSTRING NOTATION 'xxxxxxxx'

Explanation: The comma, or ending right parenthesis in a substring notation is missing. The string in the message consists of up to eight characters, starting where the error is found.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the substring notation consists of two arithmetic expressions, separated by commas and enclosed in parentheses.

IFZ059 ILLEGAL USE OF SYSTEM VARIABLE SYMBOL

Explanation: The specified system variable symbol is invalid in this context.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure that &SYSLIST, and &SYSNDX are not used in open code.

FZ060 SINGLE TERM IN LOGICAL EXPRESSION NOT SETB

Explanation: A single term in this logical expression is invalid. A logical term must be either an arithmetic relation, a character relation, or a SETB variable. Logical terms are combined into logical expressions by logical operators (AND, OR, and NOT).

Assembler Action: The statement is processed as comments.

Programmer Response: Check the logical expression for omitted relational terms (EQ, LT, etc.) or mispunched characters or terms.

IFZ061 INCOMPLETE LOGICAL EXPRESSION 'xxxxxxxx'

Explanation: An expression in this statement ended prematurely because of one of the following errors:

- Unpaired parenthesis; or
- Invalid character; or
- Invalid operator; or
- Operator not followed by a term

Assembler Action: The statement is treated as comments.

Programmer Response: Correct the logical expression.

IFZ062 INVALID SELF-DEFINING TERM, 'xxxxxxxx'

Explanation: A self-defining term is incorrectly specified. It must be:

- 1-6 decimal digits whose value is in the range 0-262,143; or
- 1-18 binary digits, enclosed by apostrophes and preceded by the character B; or
- 1-5 hexadecimal digits whose value is in the range X'00 - X'3FFFF', enclosed by apostrophes and preceded by the character X; or
- 1-2 characters, enclosed by apostrophes and preceded by the character C.

  The string in the message is up to eight characters starting with the invalid self-defining term.

Assembler Action: The statement is processed as comments.

Programmer Response: Correct the term according to the rules given in the explanation.

IFZ063 VALUE OF SELF-DEFINING TERM TOO LARGE, 'xxxxxxxx'

Explanation: The value of a decimal self-defining term in this statement is not in the range 0-262,143.

Assembler Action: The statement is processed as comments.

Programmer Response: Specify a value in the range specified in the explanation.

IFZ064 OPEN CODE ATTRIBUTE REFERENCE TO 'xxxxxxxx', WHICH IS NOT A VALID ORDINARY SYMBOL

Explanation: This attribute reference does not specify a valid ordinary symbol. Any attribute reference used outside macro definitions must specify an ordinary symbol.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid ordinary symbol that is defined in the program.

IFZ065 SET SYMBOL USE INCONSISTENT WITH ITS DECLARATION, 'xxxxxxxx'

Explanation: Either the declaration specifies this symbol as dimensioned, but in this statement the symbol is used as undimensioned, or the declaration specified this symbol as undimensioned, but in this statement the symbol is used as dimensioned. The string in the message consists of up to eight characters starting with the symbol in error.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure that your use of SET symbols is consistent with its declaration.

IFZ066 PREVIOUSLY DEFINED SEQUENCE SYMBOL

Explanation: The sequence symbol specified in the name field of this statement has already been defined within the macro definition or open code.

Assembler Action: The name field is ignored.

Programmer Response: Supply a sequence symbol that is unique within this macro definition or open code.

IFZ067 UNPAIRED RIGHT PARENTHESIS

Explanation: An expression in this statement contains a right parenthesis that is not matched by a preceding left parenthesis.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the missing left parenthesis, or delete the right parenthesis.

IFZ068 VARIABLE SYMBOL UNDEFINED, 'xxxxxxxx'

Explanation: The first or only variable symbol in the string inserted in the message has not been declared as a global or local SET symbol within this macro definition or open code, has not been defined as a symbolic parameter within this macro definition, and is not a valid system variable symbol.

Assembler Action: The statement is processed as comments.

Programmer Response: Define the symbol as a symbolic parameter or a SET symbol. Remember that any global variable symbols used in macro definitions must be declared within the definition.

IFZ069 SOURCE MACRO PREVIOUSLY DEFINED

Explanation: The operation code specified in the prototype statement is identical to the operation code of another source macro defined earlier in the program.

Assembler Action: The flagged macro definition has been checked for errors. It cannot be generated.

Programmer Response: Supply a unique operation code for this definition.

IFZ070 UNDEFINED SEQUENCE SYMBOL

Explanation: The sequence symbol used in this instruction is not defined within this macro definition or open code.

Assembler Action: No conditional assembly branch is taken.

Programmer Response: Define the symbol in the name field within the macro definition or open code (depending on where it is used), or use a sequence symbol that is already defined.

IFZ071 ILLEGAL LENGTH ATTRIBUTE REFERENCE

Explanation:

- The symbol specified in a length attribute reference (L) is not the name of a valid machine instruction, control section definition, CW instruction, DS instruction, or DC instruction.
- The symbol specified in a length attribute reference is the name of a DC or DS instruction containing variable symbols in the modifier field.

Assembler Action: The length attribute reference is set to one.

Programmer Response: Make sure the length attribute references a symbol for which length attribute references are valid.

IFZ072 ILLEGAL SCALE ATTRIBUTE REFERENCE

Explanation: The symbol referenced by a scale attribute reference (S) is not found in the name field of a valid fixed-point DC or DS instruction.

Assembler Action: The scale attribute reference is set to zero.

Programmer Response: Make sure the scale attribute references a symbol for which scale attribute references are valid.

IFZ073 ILLEGAL INTEGER ATTRIBUTE REFERENCE

Explanation: The symbol referenced by an integer attribute reference (I) is not found in the name field of a valid, fixed-point, DC, or DS instruction.

Assembler Action: The integer attribute reference is set to zero.

Programmer Response: Make sure the integer attribute references a symbol for which attribute references are valid.

IFZ074 OVERFLOW DURING ADDITION IN ARITHMETIC EXPRESSION

Explanation: During the evaluation of an arithmetic expression the addition of two terms produces a result that falls outside the range of -2^{31} through $2^{31}-1$.

Assembler Action: The result of the addition is set to zero.

Programmer Response: Make sure all the values in this expression are valid. Try to avoid overflow by adjusting the sequence in which the terms are placed in the expression. If necessary, separate the expression into two or more expressions (using SETA instructions), so that each of them is evaluated individually before they are combined.

IFZ075 OVERFLOW DURING SUBTRACTION IN ARITHMETIC EXPRESSION

Explanation: During the evaluation of an arithmetic expression, the subtraction of two terms produces a result that falls outside the range of -2^{31} through $2^{31}-1$.

Assembler Action: The result of the subtraction is set to zero.

Programmer Response: Make sure that all values in the expression are valid. Try to avoid overflow by adjusting the sequence in which the terms are placed in the expression. If necessary, separate the expression into two or more expressions (using SETA instructions), so that each of them is evaluated separately before they are combined.

IFZ076 OVERFLOW DURING MULTIPLICATION IN ARITHMETIC EXPRESSION

Explanation: During the evaluation of an arithmetic expression the multiplication of two terms produces a result that falls outside the range of -2^{31} through $2^{31}-1$.

Assembler Action: The result of the multiplication is set to zero.

Programmer Response: Make sure all the values in the expression are valid. Try to avoid overflow by adjusting the sequence in which the terms are placed in the expression. If necessary, separate the expression into two or more expressions (using SETA instructions), so that each of them is evaluated separately before they are combined.

IFZ077 CHARACTER STRING USED IN ARITHMETIC EXPRESSION TOO LONG

Explanation: The character string used as an arithmetic term is longer than eight characters.

Assembler Action: The value of the SETC variable is replaced by zero in the arithmetic expression.

Programmer Response: Make sure that any variable symbols used arithmetic expressions have a value of 1-8 characters.

IFZ078 CHARACTER STRING USED IN ARITHMETIC EXPRESSION CONTAINS NON-DECIMAL CHARACTER

Explanation: A non-decimal character is found in the value of a parameter or SETC symbol used in arithmetic term.

Assembler Action: The value of the parameter or SETC variable is replaced by zero in the arithmetic expression.

Programmer Response: Make sure that any parameter or SETC symbols used in arithmetic expressions have a value of 1-8 decimal characters.

IFZ079 NULL CHARACTER STRING USED IN ARITHMETIC EXPRESSION

Explanation: The value of a SETC symbol used as an arithmetic term is a null string.

Assembler Action: The values of the SETC symbols used in arithmetic expressions have a value of 1-8 decimal characters.

Programmer Response: Make sure that any SETC symbols used in arithmetic expressions have a value of 1-8 decimal characters.

IFZ080 PARAMETER SUBSCRIPT OUT OF RANGE

Explanation: A symbolic parameter subscript value is outside the range 1-200.

Assembler Action: The reference is treated as a reference to an omitted operand; that is, the value of a null string is assigned to it.

Programmer Response: Supply a subscript value in the range 1-200.

IFZ081 LENGTH OF CONCATENATED STRING EXCEEDS 255 CHARACTERS

Explanation: During the concatenation of strings, an intermediate string exceeding 255 characters is generated.

Assembler Action: The first 255 characters are used as the intermediate result.

Programmer Response: Make sure that the total length of two strings concatenated with each other does not exceed 255 characters. If needed, change the sequence of string evaluation by performing substring operation before concatenation.

IFZ082 SUBSCRIPT EXCEEDS DECLARED DIMENSION

Explanation: An arithmetic expression used to specify the subscript of a SET symbol has a value that exceeds the value specified in the declaration of the symbol.

Assembler Action: If the error is found in a conditional assembly statement, the statement is processed as comments. The error is found during substitution in one of the fields (name, operation, or operand) of a model statement. The whole field is replaced by a null value. If the error is found during substitution in a macro instruction operand, the operand is set to null value, but any other operands in the operand field are generated.

Programmer Response: Make sure the arithmetic expression has a value in the range of 1 through the declared dimension of the SET symbol.

IFZ083 SUBSCRIPT ZERO OR NEGATIVE

Explanation: An arithmetic expression used to specify the subscript of a SET symbol has a value that is zero or negative.

Assembler Action: If the error is found in a conditional assembly statement, the statement is processed as comments. If the error is found during substitution in one of the fields (name, operation, and operand) of a model statement, the whole field is replaced by a null value. If the error is found during substitution in a macro instruction operation, the operand is set to a null value, but any other operands in the operand field are generated.

Programmer Response: Make sure that the arithmetic expression has a value in the range of 1 through the declared dimension of the SET symbol.

IFZ084 ACTR LIMIT EXCEEDED

Explanation: The number of AIF and AGO branches within the macro definition or open code exceeds the value specified in the ACTR instruction or the conditional assembly loop counter default value.

Assembler Action: If a macro is being generated, its generation is terminated. If open code is being processed, all remaining statements are processed as comments.

Programmer Response: Correct the conditional assembly loop that caused the loop counter to be exceeded, or set the counter limit to be exceeded, or set the counter to a higher value.

IFZ085 FIRST SUBSTRING EXPRESSION ZERO OR NEGATIVE

Explanation: The arithmetic expression used to specify the starting character for a substring operation has a zero or negative value.

Assembler Action: The result of the substring operation is a null string.

Programmer Response: Make sure the arithmetic expression used to specify the starting character of the substring has a positive value not exceeding the length of the character string.

IFZ086 FIRST SUBSTRING EXPRESSION EXCEEDS STRING LENGTH

Explanation: The arithmetic expression used to specify the starting character for a substring operation has a value greater than the length of the string.

Assembler Action: The result of the substring operation is a null string.

Programmer Response: Make sure the arithmetic expression used to specify the starting character of the substring has a positive value not exceeding the length of the character string.

IFZ087 SECOND SUBSTRING EXPRESSION NEGATIVE

Explanation: The arithmetic expression used to specify the length of a substring has a negative value.

Assembler Action: The result of the substring operation is a null string.

Programmer Response: Make sure the arithmetic expression used to specify length has a zero positive value, and that the specified length does not extend beyond the end of the character string.

IFZ089 SETC OPERAND TOO LONG

Explanation: The character string value in a SETC operand contains more than eight characters.

Assembler Action: Only the first eight characters are assigned to the SETC symbol.

Programmer Response: Make sure that the value of the SETC expression contains no more than eight characters.

IFZ090 SYSLIST SUBSCRIPT NEGATIVE

Explanation: The arithmetic expression used to specify a &SYSLIST subscript has a negative value.

Assembler Action: The reference is treated as a reference to an omitted operand; that is, the value of a null string is assigned to it.

Programmer Response: Supply a non-negative value in the &SYSLIST subscript.

IFZ091 PARAMETER VALUE INVALID FOR LENGTH ATTRIBUTE REFERENCE

Explanation:

- A length attribute reference specified a symbolic parameter whose value is not the name of a machine instruction, control section definition, CW instruction, DS instruction, or DC instruction; or
- A length attribute reference specified a symbolic parameter whose value is the name of a DS or DC instruction containing variable symbols in the modifier field.

Assembler Action: The length attribute reference is set to one.

Programmer Response: Make sure that the referenced macro instruction operand is a symbol for which length attribute references are valid, or delete the length attribute reference from the macro definition.

IFZ092 PARAMETER VALUE INVALID FOR SCALE ATTRIBUTE REFERENCE

Explanation: A scale attribute reference specified a symbolic parameter whose value is not the name of a fixed-point DC or DS instruction.

Assembler Action: The scale attribute reference is set to zero.

Programmer Response: Make sure that the referenced macro instruction operand is a symbol for which scale attribute references are valid, or delete the scale attribute reference from the macro definition.

IFZ093 PARAMETER VALUE INVALID FOR INTEGER ATTRIBUTE REFERENCE

Explanation: An integer attribute reference specifies a symbolic parameter whose value is not the name of a fixed-point DC or DS instruction.

Assembler Action: The integer attribute reference is set to zero.

Programmer Response: Make sure that the referenced macro instruction operand is a symbol for which integer attribute references are valid, or delete the integer attribute reference from the macro definition.

IFZ094 PARAMETER VALUE INVALID IN ARITHMETIC EXPRESSION

Explanation: The value of a symbolic parameter used in an arithmetic expression is not a valid self-defining term, or 1-8 decimal characters created by variable symbol substitution in the macro instruction.

Assembler Action: The symbolic parameter is replaced by the value of zero in the arithmetic expression.

Programmer Response: Make sure the referenced macro instruction operand is a valid self-defining term or 1-8 decimal characters created by substitution, or remove the symbolic parameter from the arithmetic expression in the macro definition.

IFZ095 TOO MANY ERRORS IN THIS STATEMENT

Explanation: During the processing of a conditional assembly statement or a statement with variable symbol substitution, more than five errors are detected. Messages are issued only for the first five errors.

Assembler Action: If more errors are found, they will not be flagged.

Programmer Response: Correct the indicated errors, and check for further errors beyond the point indicated by the fifth error message. Any additional errors will be detected in the next assembly.

IFZ096 GENERATED STATEMENT TOO LONG

Explanation: The total length of the statement exceeds 248 characters after generation.

Assembler Action: The statement is processed as comments. If any part of the remarks field falls outside the space allowed for this statement, no part of the remarks field is listed.

Programmer Response: Make sure that the total length of a statement after generation does not exceed 248 characters.

IFZ097 UNDEFINED OP CODE, OR MACRO NOT FOUND

Explanation: The operation code of this statement does not correspond to any of the following:

- A machine instruction operation code
- An assembler instruction operation code
- The operation code of a valid library macro or a valid source macro

Assembler Action: The statement is processed as comments.

Programmer Response: Change the operation code to a valid machine, assembler, or macro operation code, or correct the corresponding macro definition. If the error occurred for a library macro, make sure the correct source statement library is assigned.

IFZ098 KEYWORD PARAMETER 'xxxxxxxx' DUPLICATED OR NOT DEFINED

Explanation: A keyword parameter appears more than once in a macro instruction, or a keyword parameter appears in a macro instruction in whose definition it is not defined as a keyword parameter. The message will also be given if an equal sign not enclosed in a quoted string or within parentheses appears in a positional parameter.

Assembler Action: If the keyword parameter is duplicated, the first parameter value is accepted. If the parameter is undefined, it is ignored.

Programmer Response: Delete the keyword from the macro instruction, define the parameter in the macro definition, or enclose the equal sign within apostrophes or parentheses.

IFZ099 TOO MANY MACROS CALLED

Explanation: The dictionary space available to the assembler is not large enough to generate all the different macros that are called in the source module.

Assembler Action: The whole assembly is processed as comments.

Programmer Response: Increase the size of the partition allocated to the assembly, or separate the module into smaller modules to be assembled separately.

IFZ100 TOO MANY MACROS CALLED OR TOO MANY VARIABLE SYMBOLS

Explanation: The dictionary space available to the assembler is not large enough to contain all the different macros, local variable symbols in open code, and global variable symbols that are used in this source module.

Assembler Action: The whole assembly is processed as comments.

Programmer Response: Increase the size of the partition allocated to assembly, or separate the module into smaller source modules to be assembled separately, making sure that references to a variable symbol all remain in the same module.

IFZ101 DICTIONARY SPACE FOR VARIABLE SYMBOLS EXHAUSTED 'xxxxxxxx'

Explanation: The dictionary space available to the assembler is not enough to contain all the different macros and global variable symbols of the entire assembly, plus all the local SET symbols and symbolic parameters used in the macro being generated.

Assembler Action: The generation of the macro is terminated.

Programmer Response: Increase the size of the partition allocated to assembly, reduce the number of local variable symbols, or separate the module into smaller source modules to be assembled separately.

IFZ102 SEQUENCE SYMBOL UNDEFINED

Explanation: A sequence symbol used in the operand of an AGO or AIF instruction is not defined in the name field of an instruction in the same macro definition or open code.

Assembler Action: The next sequential instruction is processed.

Programmer Response: Define the sequence symbol in the macro, or use a sequence symbol that is already defined.

IFZ103 REFERENCE TO GLOBAL VARIABLE SYMBOL WITH INCONSISTENT DECLARATION OF TYPE OR DIMENSION

Explanation: A global variable symbol is used whose declaration within this macro or open code is inconsistent with a previous declaration of the same global symbol. The inconsistency occurred in either the type or the dimension specification.

Assembler Action: The statement is processed as comments. If any part of the remarks field falls outside the space allowed for this statement, no part of the remarks field is listed.

Programmer Response: Make sure that all declarations of a global variable symbol are identical; for example, a global symbol cannot be declared as a SETB symbol in one macro and a SETA symbol in another; and it cannot be declared as dimensioned in one macro and undimensioned in another, or as having a dimension of 50 in one macro and 85 in another.

IFZ104 OP CODE 'xxxxxxxx' GENERATED

Explanation: One of the following assembler operation codes has been created by substitution: COPY, END, ICTL, ISEQ, PRINT, REPRO, MACRO, MEND, MEXIT, ANOP, SETA, SETB, SETC, AIF, AIFB, AGO, AGOB, GBLA, GBLB, GBLC, LCLA, LCLB, and LCLC. These operation codes are not allowed to be generated.

Assembler Action: The generated statement is processed as comments.

Programmer Response: Make sure that none of the operation codes listed in the explanation is created by substitution.

IFZ105 GENERATED OP CODE 'xxxxxxxx' UNDEFINED OR INVALID

Explanation: The operation code created by substitution is not a valid machine or assembler instruction operation code (macro instructions are not allowed to be generated).

Assembler Action: The generated statement is processed as comments.

Programmer Response: Make sure that the generation in the operation field results in a valid operation code.

IFZ106 GENERATED OP CODE IS BLANK

Explanation: The operation code created by substitution contains no characters or only blank characters.

Assembler Action: The generated statement is processed as comments.

Programmer Response: Make sure that substitution results in a valid machine or assembler operation code.

IFZ107 MACRO 'xxxxxxxx' NOT EXPANDABLE DUE TO ERROR IN DEFINITION

Explanation:

Source Macro: The prototype statement of the macro definition contains errors.

Library Macro: The library macro contains an error defined by another error message (of the type that has no statement number).

Assembler Action: The statement is processed as comments.

Programmer Response:

Source Macro: Correct the prototype statement.

Library Macro: Edit and catalog the macro again.

IFZ108 MACRO 'xxxxxxxx' NOT EXPANDABLE DUE TO ERROR IN MACRO INSTRUCTION

Explanation: An error has been found in a substitution expression in a macro instruction operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Check the other error messages on this macro instruction and correct the error(s).

IFZ109 INVALID OR ILLEGAL NAME FIELD

Explanation: Either the name field is not blank and does not contain a valid ordinary symbol, that is, one to eight alphameric characters the first of which is alphabetic, or there is an ordinary symbol in a name field that should only contain a sequence symbol or blank (CNOP, ORG, END, USING, DROP).

Assembler Action: The name field is ignored.

Programmer Response: Supply a valid ordinary symbol, delete the characters from the name field, or, if you want to write a comments statement, supply an asterisk in the begin column, or remove entries in name field.

IFZ110 NAME FIELD TOO LONG

Explanation: The symbol in the name field exceeds eight characters.

Assembler Action: The name field is ignored.

Programmer Response: Make sure the name field is blank or contains a valid ordinary symbol, that is 1-8 alphameric characters the first of which is alphabetic.

IFZ111 GENERATED SEQUENCE SYMBOL

Explanation: A period is found in the name field of a generated statement.

Assembler Action: The name field is ignored.

Programmer Response: Make sure the value generated in the name field is either a valid ordinary symbol or blank.

IFZ112 INVALID CHARACTER IN CONSTANT, 'xxxxxxxx'

Explanation: Subfield 4 (constant) of a DC or DS instruction contains characters that are invalid for the type of constant specified in subfield 2.

Assembler Action: The DS or DC instruction is processed as comments.

Programmer Response: Make sure the characters used to specify the value of the constant are valid for this type of constant. Change either the type or the value specification.

IFZ113 SYMBOL 'xxxxxxxx' TOO LONG

Explanation: The specified symbol contains more than eight characters. Only the first eight characters of the symbol are identified in the message.

Assembler Action: The operand is checked for further errors, but this operand and any further operands are ignored when the object code is generated.

Programmer Response: Supply a valid symbol.

IFZ114 RIGHT PARENTHESIS MISSING, 'xxxxxxxx'

Explanation: In the operand indicated in the message, a left parenthesis not matched by a right parenthesis has been found, or the parentheses have been incorrectly placed in the operand, for example, MVC (1, 2), 3(4) will cause this message to be issued. The first character in the string inserted in the message indicates where the right parenthesis was expected. If only a blank appears in the string the right parenthesis was expected at the end of the operand.

Assembler Action: The operand and the rest of the operand field is ignored.

Programmer Response: Make sure the parentheses are paired and correctly placed.

IFZ115 UNPAIRED APOSTROPHE

Explanation: No terminating apostrophe has been found to end the quoted string in this statement.

Assembler Action: The statement is ignored.

Programmer Response: Supply the missing apostrophe.

IFZ116 INVALID SELF-DEFINING TERM, 'xxxxxxxx'

Explanation: The first or only self-defining term specified in the text inserted in the message contains invalid characters or a null value (for example, B' 102', X'').

Assembler Action: The operand in which the term appears is ignored.

Programmer Response: Supply a valid decimal, binary, hexadecimal, or character self-defining term.

IFZ117 VALUE OF SELF-DEFINING TERM 'xxxxxxxx' TOO LARGE

Explanation: The value of the specified self-defining term is either too long or too large. Valid symbols are:

- 1–6 decimal digits whose value is in the range 0–262,143 or
- 1–18 binary digits, enclosed by apostrophes and preceded by the character B; or
- 1–5 hexadecimal digits whose value is in the range X'00'–X'3FFFF', enclosed by apostrophes and preceded by the character X; or
- 1–2 characters, enclosed by apostrophes and preceded by the character C.

Assembler Action: The operand in which the term appears is ignored.

Programmer Response: Correct the term according to the rules given in the explanation.

IFZ118 ILLEGAL ATTRIBUTE REFERENCE, 'xxxxxxxx'

Explanation: The length attribute reference does not specify a valid ordinary symbol or location counter reference (*). The first character of the inserted string indicates the point where the invalid attribute reference was found.

Assembler Action: The operand is ignored.

Programmer Response: Supply a valid ordinary symbol or location counter reference.

IFZ119 TOO MANY OPERATORS, 'xxxxxxxx'

Explanation: More than 15 operators have been found in the operand specified by the message. The first character of the inserted string indicates the point where too many operators have been encountered.

Assembler Action: The operand is ignored.

Programmer Response: Limit the number of operators. If necessary, use EQU instructions to break up the expression into smaller expressions.

IFZ120 TOO MANY LEVELS OF PARENTHESES, 'xxxxxxxx'

Explanation: More than five levels of parentheses are used in an expression in the operand specified by the message. The first character in the string indicates the point where too many levels of parentheses have been encountered.

Assembler Action: The operand is ignored.

Programmer Response: Limit the number of levels of parentheses. If necessary, use EQU instructions to break up the expression into smaller expressions, each of which is evaluated separately.

IFZ121 ILLEGAL CHARACTER IN EXPRESSION, 'xxxxxxxx'

Explanation: In an expression an invalid character has been found in or instead of a term. The first character of the text inserted in the message identifies the invalid character.

Assembler Action: The operand in error and the rest of the statement are ignored.

Programmer Response: Supply a valid term.

IFZ122 INVALID DELIMITER, 'xxxxxxxx'

Explanation: An operand or sub-operand is not delimited by a comma, a left or right parenthesis, or a blank. The first character of the text inserted in the message identifies the invalid delimiter.

Assembler Action: The operand in error and the rest ot the statement are ignored.

Programmer Response: Make sure all the delimiters in the statement are correct.

IFZ123 INVALID TYPE SPECIFICATION, 'xxxxxxxx'

Explanation: The type specified in subfield 2 of a DC or DS instruction is invalid or missing. The text inserted in the message consists of up to eight characters, starting at the point where a valid type specification is expected.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid type specification (A, B, C, F, H, R, V, X, or Y).

IFZ124 INVALID SYMBOL IN ENTRY, ENTRN, OR WXTRN STATEMENT

Explanation: An ENTRY, EXTRN, or WXTRN instruction contains an invalid symbol. The operand field of these statements must consist of one or more ordinary symbols, separated by commas. Any ordinary symbols defined by the ENTRY instruction must also appear in the name field of an instruction in this source module.

Assembler Action: The operand in error and the rest of the statement are ignored.

Programmer Response: Make sure that the operand field follows the rules given in the explanation.

IFZ126 EXPONENT MODIFIER USED ILLEGALLY, 'xxxxxxxx'

Explanation: An exponent modifier is specified for a DC or DS instruction operand that is not a fixed-point type constant. The character string inserted in the message consists of up to eight characters starting with the invalid exponent modifier.

Assembler Action: The statement is processed as comments.

Programmer Response: Delete the exponent modifier or change the type specification.

IFZ127 SCALE MODIFIER USED ILLEGALLY 'xxxxxxxx'

Explanation: The scale modifier is specified for a DC or DS instruction operand which is not a fixed-point type constant.

Assembler Action: The statement is processed as comments.

Programmer Response: Delete the scale modifier or change the type specification.

IFZ128 CONSTANT FIELD MISSING OR PRECEDED BY INVALID FIELD, 'xxxxxxxx'

Explanation: In a DC instruction operand, either invalid characters are found between the modifier and constant subfield, or the constant subfield does not contain any nominal value. The first character in the string indicates the point where the invalid constant value was found.

Assembler Action: The statement is processed as comments.

Programmer Response: Delete the invalid characters, or supply a nominal value, or change the operation code to DS if you wish only to specify a data area.

IFZ129 INVALID DUPLICATION FACTOR OR MODIFIER, 'xxxxxxxx'

Explanation: A syntax error was found in the duplication factor subfield (subfield 1) or the modifier subfield (subfield 3) of this DC or DS instruction operand. The first character in the string indicates the point where the invalid constant value was found.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the syntax in the expression is correct.

IFZ133 TOO MANY SYMBOLS IN STATEMENT

Explanation: More than 50 symbols have been specified in an ENTRY, EXTRN, or WXTRN instruction.

Assembler Action: The first 50 symbols are processed.

Programmer Response: Place the excessive operands in additional ENTRY, EXTRN, or WXTRN instructions.

IFZ134 STATEMENT COMPLEXITY EXCEEDED, 'xxxxxxxx'

Explanation:

- More than one operand has been specified in a DC or DS instruction; or
- The constant subfield of a DC or DS operand contains too many symbols or terms or both. The maximum number of symbols and terms that can be handled by the assembler is around 30.

  The first character in the string indicates the point where the invalid character was found.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure only one period is coded, or break up the constant subfield into two or more statements.

IFZ135 SYMBOL 'xxxxxxxx' PREVIOUSLY DEFINED

Explanation: The ordinary symbol defined in this instruction either by appearing in the name field, or by appearing in the operand field of an EXTRN or WXTRN instruction, has already been defined within the source module.

Assembler Action: The second definition is ignored.

Programmer Response: Supply a name that does not conflict with any other symbols in the program.

IFZ136 ARITHMETIC OVERFLOW (IN OPERAND n)

Explanation: During the evaluation of an expression, a value has been reached which is outside the range of -2^{31} through $2^{31}-1$.

Assembler Action: The expression is ignored.

Programmer Response: Rearrange the terms or expression to avoid overflow. If necessary, use EQU statements to separate the expression into smaller expressions that can be evaluated separately and then combined.

IFZ137 EXPRESSION COMPLEXLY RELOCATABLE (IN OPERAND n)

Explanation: A complexly relocatable expression is used in the operand field of an EQU, CNOP, or ORG instruction or in the modifier subfield of a DC or DS instruction operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Correct the expression so that it is simply relocatable (EQU, ORG) or absolute (EQU, CNOP, modifiers).

IFZ138 TOO FEW OPERANDS

Explanation: This statement requires more operands than are supplied.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the missing operand(s).

IFZ139 INVALID DUPLICATION FACTOR (IN OPERAND n)

Explanation: The duplication factor is relocatable.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply an absolute value.

IFZ140 INVALID LENGTH MODIFIER (IN OPERAND n)

Explanation: The length modifier is either too large, zero, or relocatable. The maximum value allowed for the length modifier varies with the type specified for this operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the length modifier is an absolute value in the range allowed for this type of constant.

IFZ141 INVALID SCALE MODIFIER (IN OPERAND n)

Explanation: The expression used to specify the scale modifier is relocatable.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Make sure the scale modifier expression specifies an absolute value, and that any symbols used in it have been previously defined.

IFZ142 INVALID EXPONENT MODIFIER (IN OPERAND n)

Explanation: The expression used to specify the exponent modifier is relocatable.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Make sure the exponent modifier expression specified an absolute value, and that any symbols used in it have been previously defined.

IFZ143 INVALID CNOP OPERAND

Explanation: One or both of the operands in this CNOP instruction are invalid. Only the following combinations are allowed: 0 and 4, 2 and 4, 0 and 8, 2 and 8, 4 and 8, and 6 and 8.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply one of the combinations listed in the explanation.

IFZ144 INVALID END OPERAND

Explanation: The operand of the END instruction is invalid. It must be a simply relocatable expression whose value represents an address within an ordinary control section (that is, not a dummy or common control section) in this source module, or an external reference.

Assembler Action: The operand field is ignored.

Programmer Response: Supply a valid operand as described in the explanation.

IFZ145 RELOCATABLE TERM IN DIVIDE OR MULTIPLY OPERATION (IN OPERAND n)

Explanation: A relocatable term is used in a multiply or divide operation in an expression in this statement.

Assembler Action: The statement is processed as comments.

Programmer Response: Use only absolute terms in divide or multiply operations.

IFZ146 NAME MISSING

Explanation: The name is missing in this EQU or EQUR instruction.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid name field.

IFZ147 INVALID START STATEMENT

Explanation:

- The operand field is not blank or an absolute value; or
- The START instruction does not identify the beginning of the first control section in this source module; it was preceded by another START instruction, a CSECT instruction, or a statement that causes an unnamed control section (private code) to be initiated.

Assembler Action: The statement is processed as a CSECT statement.

Programmer Response: Make sure any operand specified is an absolute value, and that the START instruction initiates the first control section in the source module.

IFZ148 ILLEGAL SYMBOL 'XXXXXXXX' IN IN ENTRY STATEMENT

Explanation: The specified symbol is not defined as a relocatable symbol within an ordinary control section (not a dummy or common control section,) or is defined as an EXTRN symbol or has already appeared as an entry statement.

Assembler Action: The symbol is ignored.

Programmer Response: Make sure the ENTRY operand is a valid external name as defined in the explanation.

IFZ149 SYMBOL 'xxxxxxxx' NOT PREVIOUSLY DEFINED

Explanation: The specified symbol appears in an EQU, ORG, or CNOP operand or in the modifier subfield of a DC or DS instruction, but has not been defined prior to this use. These fields require that any symbols used in them are previously defined.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the definition of this symbol precedes this statement.

IFZ150 VALUE OF ORG OPERAND LESS THAN CONTROL SECTION STARTING ADDRESS

Explanation: The operand of an ORG instruction results in a value less than the starting address of the control section.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the operand of the ORG instruction is a positive relocatable expression, greater than the starting address in the control section.

IFZ151 LOCATION COUNTER OVERFLOW

Explanation: The location counter value is greater than or equal to X' 3FFFF'.

Assembler Action: The location counter is carried in three bytes. When overflow occurs, the location counter will not be updated and every statement that causes overflow will be flagged.

Programmer Response: The probable cause of the error is a high ORG instruction value or a high START instruction value. Correct the value or divide the control section.

IFZ152 ORG OPERAND VALUE NOT WITHIN THIS CONTROL SECTION

Explanation: The operand of an ORG instruction is not a simply relocatable expression whose value falls within the current control section.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the resulting value of the expression in the operand field falls within the control section where the ORG is coded. Any relocatable symbols defined in other control sections must be paired (that is, each such term must be matched by another term from the same control section with the opposite sign).

IFZ153 ILLEGAL USE OF LOCATION COUNTER REFERENCE

Explanation: A location counter reference (*) is used in the modifier subfield of a DC or DS instruction or literals or in the operand of a CNOP instruction.

Assembler Action: The statement is processed as comments.

Programmer Response: Remove the invalid location counter reference.

IFZ154 TOO MANY ENTRY SYMBOLS

Explanation: The number of ENTRY operands specified in this source module exceeds 10.

Assembler Action: ENTRY operands encountered in the rest of the assembly are ignored.

Programmer Response: Reduce the number of ENTRY operands, or separate the module into two or more modules.

IFZ155 TOO MANY EXTERNAL SYMBOLS

Explanation: Too many entries have been made in the external symbol dictionary. Only 255 entries can be made for the following: control sections, dummy sections, common control sections, and external references (EXTRN, WXTRN, V-type constants). ENTRY operands are not counted towards this maximum, but the number of entry operands must not exceed 100.

Assembler Action: No more symbols are entered in the external symbol dictionary. The rest of the source module is assembled as part of the control section currently being processed.

Programmer Response: Reduce the number of ESD items, or separate the source module into two or more modules.

IFZ156 SYMBOL 'xxxxxxxx' UNDEFINED

Explanation: The specified symbol has not been defined within the module; that is, it has not appeared in the name field of an instruction or in the operand field of an EXTRN or WXTRN instruction.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Make sure the symbol is defined, or use a symbol that has already been defined.

IFZ158 TOO MANY OPERANDS

Explanation: Too many operands have been coded for this statement.

Assembler Action: If the statement is an assembler instruction, the excessive operands are ignored.

If the statement is a machine instruction, zeros are generated in the object module.

Programmer Response: Delete the excessive operands, make sure that the format of the operand field is correct.

IFZ159 TOO FEW OPERANDS

Explanation: This instruction requires more operands than are specified.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply the right number of operands; make sure that the format of the operand field is correct.

IFZ160 COMPLEXLY RELOCATABLE EXPRESSION IN OPERAND n

Explanation: A complexly relocatable expression has been used in an operand where a simply relocatable or absolute expression is required.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a valid simply relocatable or absolute expression.

IFZ161 OPERAND n NOT A CURRENT BASE REGISTER

Explanation: The register specified in the operand of this DROP instruction is not a current base register, either because it has not been specified as a base register by a previous USING instruction, or because it has been encountered in a previous DROP instruction.

Assembler Action: The operand is ignored.

Programmer Response: Make sure the operand is currently being used as a base register.

IFZ162 ADDRESS OPERAND OUT OF RANGE

Explanation: The address operand in a RA type instruction must be an absolute or relocatable expression in the range 0 to $2^{18}-1$.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a valid address operand.

IFZ163 ADDRESSABILITY ERROR IN OPERAND n

Explanation: An address specified in the operand of this statement is not covered by any base register, that is, it does not appear in the range of a USING instruction, or relocatability of transfer address not the same as that of the instruction which makes reference.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Make sure the address of the symbol in the operand falls within the ranges of a address instruction; it must be within the first 127 bytes of the address specified in the USING instruction.

IFZ164 INVALID USE OF SYMBOLIC REGISTER IN OPERAND n

Explanation: A symbolic register expression is specified when absolute relocatable, or complexly relocatable expression is required, or a symbolic register expression appears in a multiterm expression.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply an expression that is not a symbolic register expression.

IFZ165 REGISTER VALUE IN OPERAND n NOT ODD

Explanation: The indicated operand does not specify an odd register value.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Specify an odd-numbered register in the range 1-7.

IFZ166 REGISTER VALUE IN OPERAND n OUT OF RANGE

Explanation: The register number specified in this operand is not in the range required by this instruction.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Specify a register value in the range range 0-7.

IFZ167 REGISTER VALUE IN OPERAND n NOT ABSOLUTE

Explanation: The register number specified in this operand is not an absolute value.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Specify an absolute value.

IFZ168 EXTERNAL REGISTER VALUE IN OPERAND n OUT OF RANGE

Explanation: External register specification is not a value from 0 to 127.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a value in the valid range.

IFZ169 EXTERNAL REGISTER VALUE IN OPERAND n NOT ABSOLUTE

Explanation: External register specification is not an absolute expression.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply an absolute value.

IFZ170 IMMEDIATE VALUE IN OPERAND n OUT OF RANGE

Explanation: The value specified as an immediate value is negative or too high. The allowable range is 0-255.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Supply an absolute value in the range described in the explanation.

IFZ171 IMMEDIATE VALUE IN OPERAND n NOT ABSOLUTE

Explanation: The immediate value specified as an absolute value is relocatable.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Supply an absolute value in the range 0-255.

IFZ172 DISPLACEMENT VALUE IN OPERAND n OUT OF RANGE

Explanation: The displacement value in the specified operand is not in the range 0-127.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Make sure the displacement is specified as an absolute value in the range 0-127.

IFZ173 DISPLACEMENT VALUE IN OPERAND n NOT ABSOLUTE

Explanation: The displacement value in the specified operand is relocatable.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Make sure the displacement is specified as an absolute value in the range 0-127.

IFZ174 BIT SELECTION VALUE IN OPERAND n OUT OF RANGE

Explanation: Bit specification is not an expression with a value from 0 to 7.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a value in the valid range.

IFZ175 BIT SELECTION VALUE IN OPERAND n NOT ABSOLUTE

Explanation: Bit specification is not an absolute expression.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply an absolute value from 0 to 7.

IFZ176 BASE REGISTER VALUE IN OPERAND n OUT OF RANGE

Explanation: The value specified in the base register subfield of this operand is not in the range 0-7.

Assembler Action: Zeros are generated instead of this instruction in the object module.

Programmer Response: Make sure the base register is specified as an absolute value in the range 0-7.

IFZ177 BASE REGISTER VALUE IN OPERAND n NOT ABSOLUTE

Explanation: The value specified in the base register subfield of this operand is not absolute.

Assembler Action: Zeros are generated instead of the instruction in the object module.

Programmer Response: Make sure the base register is specified as an absolute value in the range 0-7.

IFZ178 BYTE SELECTION VALUE IN OPERAND n OUT OF RANGE

Explanation: Byte specification is not a value of 0 or 1. In the case of the extended mnemonic BBE the value should be in the range 0 to 15.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a value in the valid range.

IFZ179 BYTE SELECTION IN OPERAND n NOT ABSOLUTE

Explanation: Byte specification is not an absolute expression.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply an absolute value.

IFZ180 SUBFIELD MISSING IN OPERAND n

Explanation: An operand that should contain two subfields has only one, or an operand that should contain three subfields has only one or two. This error can be caused by specifying an expression that is not a symbolic register in a position where a symbolic register was intended.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a valid operand.

IFZ181 REGISTER VALUE 0 SPECIFIED AS BASE REGISTER

Explanation: Register 0 has been specified as a base register in a USING or DROP instruction. Only register values from 1 to 7 are permitted.

Assembler Action: The operand is ignored.

Programmer Response: Supply a valid register value.

IFZ182 ALIGNMENT ERROR IN OPERAND n

Explanation: This operand refers to a storage location that is not on the boundary required by this instruction.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Correct the address specification.

IFZ183 SUBFIELD SPECIFIED ILLEGALLY IN OPERAND n

Explanation: Three subfields have been specified in an operand where only two are allowed, or three or two specified where only one is allowed.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a correct operand.

IFZ184 EXPONENT MODIFIER OUT OF RANGE IN CONSTANT n (OPERAND m)

Explanation: The value of the exponent modifier is too large or too small. The sum of the exponent modifier and the exponent specification in the constant must be in the range -85 through +75.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Make sure the total value of the exponent in the constant subfield and the exponent modifier in the modifier subfield is in the range of -85 through +75.

IFZ185 SCALE MODIFIER OUT OF RANGE IN CONSTANT n (OPERAND m)

Explanation: The scale modifier is either too large or too small. For a fixed-point constant, the allowed range is -187 through +346.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Make sure the scale modifier value falls in the range described in the explanation.

IFZ186 ABSOLUTE TRANSFER ADDRESS IN OPERAND n

Explanation: The transfer address specified is an absolute expression.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a relocatable expression for transfer address.

IFZ187 TRANSFER ADDRESS IN OPERAND n OUT OF RANGE

Explanation: The transfer address specified is out of the available displacement range which is +1024 to -1022 halfwords for BCL, BZL, B and +64 to -62 halfwords for BB and BCT, as counted from the instruction location.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a transfer address within the valid range.

IFZ188 INVALID SYNTAX IN DATA FIELD OF CONSTANT n (OPERAND m)

Explanation: The syntax is invalid in the present constant subfield of this operand. For instance, an E is present to designate an exponent, but no exponent is found.

Assembler Action: Zeros are generated in the source module.

Programmer Response: Correct the syntax of the statement.

IFZ189 DATA ITEM TOO LARGE IN CONSTANT n (OPERAND m)

Explanation: The constant specified in the constant subfield of this DC or DS instruction operand is too large for the data type or for the length specified explicitly in the length modifier.

Assembler Action: The value is truncated on the left.

Programmer Response: Change the type specification or the length modifier.

IFZ190 LENGTH MODIFIER ILLEGAL WITH CONSTANT n (OPERAND m)

Explanation: An A-, R-, or Y-type address constant has been specified with an explicit length which is correct for absolute, but not for relocatable expressions.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Change length modifier to allow expression to be relocatable or make the expression absolute.

IFZ191 ILLEGAL EXPRESSION IN ADDRESS CONSTANT n (OPERAND m)

Explanation: Only a simple expression is allowed in an address constant (no subfields).

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a simple expression.

IFZ192 ILLEGAL EXPRESSION IN OPERAND n

Explanation: Only a simple expression is allowed as a CW operand (no subfield allowed).

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply a simple expression.

IFZ193 ALIGNMENT ERROR IN OPERAND 4

Explanation: Operand 4 of a CW instruction does not contain an even address.

Assembler Action: The operand is accepted as it is specified.

Programmer Response: Supply an operand 4 value that is even.

IFZ194 TOO FEW OPERANDS

Explanation: Fewer than four operands found in a CW instruction.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Supply the missing operand(s).

IFZ195 OPERAND n NOT ABSOLUTE

Explanation: The value specified in operands 1, 2, or 3 is not an absolute value.

Assembler Action: Zeros are generated instead of the CW in the object module.

Programmer Response: Make sure the values of the expressions in operands 1, 2, and 3 are absolute.

IFZ196 TOO MANY OPERANDS

Explanation: More than four operands have been found in CW instruction.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Delete the excessive operand(s).

IFZ197 VALUE OF OPERAND n OUT OF RANGE

Explanation: The value specified for the operand identified in the message is too high or negative. The value of operand 1 must be in the range 0-3, the value of operand 2 must be in the range 0-3, and the value of operand 3 must be in the range 0-1023.

Assembler Action: Zeros are generated instead of the CW in the object module.

Programmer Response: Make sure the operand specifies an absolute value in the range described in the explanation.

IFZ198 SYMBOL IN ADDRESS OPERAND DEFINED IN DUMMY SECTION

Explanation: A symbol in a CW address operand or in a RA machine instruction is defined in a dummy section. If a symbol in an expression in the address operand is defined in a dummy section, the symbol must be paired with another symbol with the opposite sign defined in the same dummy section.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Delete any symbols in a CW address operand defined in dummy sections, or make sure they are paired with other symbols defined in the same dummy section.

IFZ199 DUMMY SECTION SYMBOL USED ILLEGALLY IN CONSTANT n (OPERAND m)

Explanation: A dummy section symbol appearing in the constant subfield of this address constant is defined in a dummy section. If a symbol in an expression in the constant subfield is defined in the dummy section, the symbol must be paired with another symbol with the opposite sign defined in the same dummy section.

Assembler Action: Zeros are generated in the object module.

Programmer Response: Delete any dummy section symbols, or make sure they are paired with other symbols defined in the same dummy section.

IFZ200 NAME FIELD TOO LONG

Explanation: The length of the symbol in the name field exceeds eight characters.

Assembler Action: The name field is ignored.

Programmer Response: Make sure the name field is not longer than eight characters.

IFZ201 NAME FIELD NOT SEQUENCE SYMBOL OR BLANK

Explanation: The name field contains something other than a valid sequence symbol or blank. The following instructions must have a blank or a sequence symbol in the name field: EJECT, PRINT, SPACE, MNOTE, PUNCH, REPRO, and TITLE (except the first TITLE statement in the module).

Assembler Action: The name field is ignored.

Programmer Response: Supply a valid sequence symbol, or leave the name field blank.

IFZ202 TITLE NAME TOO LONG

Explanation: The name field of the first TITLE instruction in the program contains more than four characters that are used to specify a valid sequence symbol.

Assembled Action: The name field is ignored.

Programmer Response: Supply up to four alphameric characters in the name field, or leave the name field blank.

IFZ203 TITLE NAME CONTAINS NON-ALPHAMERIC CHARACTER

Explanation: A non-alphameric character was encountered in the name field of the first TITLE statement in the program.

Assembler Action: The name field is ignored.

Programmer Response: Supply one to four alphameric characters, or leave the name field blank.

IFZ204 OPERAND MISSING

Explanation: The operand field of a PRINT, PUNCH, or TITLE statement is blank.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid operand field.

IFZ205 FIRST APOSTROPHE MISSING

Explanation: The first apostrophe in the operand of an MNOTE, PUNCH, or TITLE instruction is missing.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure the operand is a character combination enclosed in apostrophes.

IFZ206 SINGLE AMPERSAND IN OPERAND

Explanation: A single ampersand that is not part of a variable symbol appears in the MNOTE, PUNCH, or TITLE operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure that an ampersand that is meant to be part of the operand rather than of a variable symbol in the operand is coded as a double ampersand.

IFZ207 LAST APOSTROPHE MISSING

Explanation: The operand of an MNOTE, PUNCH, or TITLE instruction does not end with a single apostrophe.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the closing apostrophe.

IFZ208 TITLE OR PUNCH OPERAND TOO LONG

Explanation: The operand of a TITLE or PUNCH instruction is too long. The maximum length of the TITLE operand is 100 characters, excluding the enclosing apostrophes, and the maximum length of the PUNCH operand is 80 characters, excluding the enclosing apostrophes.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply an operand that does not exceed the length described in the explanation.

IFZ209 OPERAND FIELD ILLEGALLY TERMINATED

Explanation: The closing apostrophe of an MNOTE, PUNCH, or TITLE operand is not immediately followed by a blank. This message can be caused by a single apostrophe coded or generated inside the enclosing apostrophes or by a missing blank between the operand field and the remarks field.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure all apostrophes inside the enclosing apostrophes are coded as double apostrophes, or supply the missing blank between the operand and the remarks field.

IFZ211 NON-DECIMAL CHARACTER IN OPERAND

Explanation: The operand of a SPACE instruction contains non-decimal characters or the severity code operand of an MNOTE instruction contains characters that are not decimal or an asterisk.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a decimal value or (for MNOTE only) an asterisk.

IFZ212 INVALID PRINT OPERAND

Explanation: The operand of a PRINT instruction does not specify one or more of the following values: ON, OFF, GEN, NOGEN, DATA, NODATA.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply from one to three operands that do not conflict with each other. The operands are listed in the explanation.

IFZ213 CONFLICTING PRINT OPERANDS

Explanation: Conflicting operands have been specified in a PRINT statement. Only one value from each of the following three pairs can be specified: ON/OFF, GEN/NOGEN, and DATA/NODATA.

Assembler Action: The statement is processed as comments.

Programmer Response: Delete conflicting values.

IFZ214 'x' IS AN INVALID DELIMITER

Explanation: An operand in a PRINT statement is not immediately followed by a comma or a blank.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply the correct delimiter.

IFZ215 OPERAND FIELD INCOMPLETE

Explanation: A PRINT instruction ends with a comma followed by a blank, or an MNOTE instruction contains a severity code operand but no message operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Delete the comma or supply the additional operand.

IFZ216 MNOTE GENERATED

Explanation: An MNOTE statement specified with a severity code, or an explicitly omitted (by means of a comma) severity code has been encountered.

Assembler Action: Processing continues.

Programmer Response: Determine the cause of the message by referring to the source statements section of the listing. The MNOTE message is written at the statement number supplied with the message.

IFZ217 MNOTE SEVERITY VALUE TOO HIGH

Explanation: The severity code specified in the first operand of an MNOTE instruction is greater than 255.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a severity code in the range 0-255, or omit the first operand.

IFZ218 NULL STRING IN PUNCH OPERAND

Explanation: The operand field of a PUNCH statement contains only two apostrophes placed immediately after each other.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply 1-80 characters inside the apostrophes.

IFZ220 TOO MANY SUBFIELDS SPECIFIED IN EQUR OPERAND

Explanation: Operand in EQUR statements can only be of two kinds: R(N) and Q.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid operand as described in the explanation.

IFZ221 REGISTER SPECIFIED IN EQUR OPERAND IS NOT 1, 3, 5 OR 7.

Explanation: An EQUR operand can be of two kinds: R(N), or Q. The R must be an odd register: 1, 3, 5 or 7.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid odd register.

IFZ222 RELOCATABLE EXPRESSION IN EQUR OPERAND

Explanation: Only absolute expressions are allowed in EQUR operands.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply an absolute expression.

IFZ223 BYTE SELECTION SUBFIELD MISSING

Explanation: Operand in EQUR statements can only be of two kinds: R(N) and Q.

Assembler Action: Statement processed as comments.

Programmer Response: Supply a valid operand as described in explanation.

IFZ224 INVALID USE OF SYMBOLIC REGISTER

Explanation: A symbolic register expression is specified where an absolute, relocatable, or complexly relocatable expression is required, or a symbolic register expression appears in a multi-term expression.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply an expression which is not a symbolic register expression.

IFZ225 TOO MANY OPERANDS IN EQUR INSTRUCTION

Explanation: EQUR can only have one operand.

Assembler Action: The statement is processed as comments.

Programmer Response: Make sure only one operand is specified.

IFZ226 INVALID BYTE SELECTION IN EQUR OPERAND

Explanation: Byte specification is not an expression of the value 0 or 1.

Assembler Action: The statement is processed as comments.

Programmer Response: Supply a valid byte specification.

IFZ227 INVALID ORIGIN FOR RELOCATABLE R-TYPE CONSTANT

Explanation: A relocatable R-type constant is assembled at location 0.

Assembler Action: No RLD is produced for this constant.

Programmer Response: Move the constant to a location other than zero.

IFZ230 PERMANENT I/O ERROR ON SYS00x

Explanation: An unrecoverable I/O error occurred on the device to which this file is assigned.

Assembler Action: The assembly is terminated. No listing is produced.

Programmer Response: Re-assemble the program.

Operator Response: Rerun the job using a different device for the file indicated in the message.

IFZ231 INVALID DEVICE FOR SYS00x

Explanation: The device assigned for this file cannot be used as a work file by the assembler.

Assembler Action: The assembly is terminated. No listing is produced.

Programmer Response: If you have supplied an ASSGN statement for this work file, correct the ASSGN statement so that it specifies a direct-access device that can be used by the assembler. If you have not supplied any ASSGN statement, rerun the job, making sure that the work files are assigned to direct-access storage devices.

Operator Response: Use the ASSGN command to assign the indicated file to a direct-access storage device, and rerun the job.

IFZ232 SYSxxx NOT ASSIGNED

Explanation: This file is required by the assembler, either because it is a work file, or because it is required by an option specified in the OPTION statement, but the file is not assigned or the IGNORE option is specified for the file. The IGN option is valid only for SYSPCH and SYSLST.

Assembler Action: The assembly is terminated. No listing is produced.

Programmer Response: Rerun the job, making sure that the indicated file is assigned, or change the corresponding option on the OPTION statement. *Or*: Execute the LISTIO command and verify the assignments. Submit an ASSGN command for the file indicated in the message, and rerun the job.

IFZ233 ASSEMBLER PARTITION TOO SMALL/DE-EDITOR PARTITION TOO SMALL

Explanation: The number of bytes allocated for the assembler are not enough. The assembler must not be loaded into less than 20K bytes. Note that in a foreground partition the assembler is always loaded immediately after the save area.

Assembler Action: The assembly is terminated. No listing is produced.

Programmer Response: Specify a larger partition for the job and rerun it.

Operator Response: Use the ALLOC command to increase the size of the partition and rerun the job.

IFZ234 END OF EXTENT FOR SYS00x

Explanation: The direct access storage extent assigned for this file is not large enough. Note that multiple extents are not used for an assembler work file .

Assembler Action: The assembly is terminated. No listing is produced.

Programmer Response: If you have supplied DLBL and EXTENT statements for the file in your job, increase the extent specified in the EXTENT statement and rerun the job. If not, check the LSERV output to make sure that the standard assignment for this file specifies an extent that is large enough. If you do not wish to change the EXTENT size, separate the program into two or more source modules and assemble each module separately. *Or*: If the standard assignment for the file indicated in the message was used by this job, execute LSERV, and return the output to the programmer.

IFZ236 ASSEMBLER CANNOT CONTINUE/ DE-EDITOR CANNOT CONTINUE

Explanation:

- If this message is preceded by other messages, the preceding message explains the reason why the assembler cannot continue.
- If the message is not accompanied by other messages, an error in the logic of the assembler has been encountered.

Assembler Action: Assembly is terminated. No listing is produced. If the message is caused by an error in the assembler, a main storage dump of the assembler area is given.

Programmer Response: If the message is caused by another error message, perform the actions indicated in the description of that message. Otherwise, save your job stream, SYSLOG listing and SYSLST listing to aid in problem determination, before calling IBM.

Operator Response: If the message is preceded by another error message, ignore this message, and perform the actions indicated for this message. If this message appears alone, consider the preceding job as terminated.

IFZ240 TOO MANY MACROS

Explanation: The capacity of the assembler is exceeded.

Assembler Action: All statements will be treated as comments.

Programmer Response: Separate the source module into smaller modules, and assemble each module separately.

IFZ241 TOO MANY GLOBAL VARIABLE SYMBOLS

Explanations: The partition allocated to the assembler is not large enough to process the source module because too many global symbols have been used.

Assembler Action: All statements will be treated as comments.

Programmer Response: Increase the size of the partition, or reduce the number of global symbols by grouping them together in SET symbol arrays (subscripted SET symbols).

IFZ242 INCONSISTENT TYPE OF GLOBAL VARIABLE SYMBOL 'xxxxxxxx' IN 'yyyyyyyy'

Explanation: The type of variable symbol specified in the declaration is inconsistent with the type specified in another macro or in open code. For example, if a global symbol is declared as a SETA symbol in one macro definition, it must be declared as a SETA symbol in all macro definitions where it is used.

Assembler Action: All declarations inconsistent with the first declaration are considered invalid. The macro definitions are processed in the order in which they appear in the source deck, with all outer macros first, followed by the inner macros of the first level, inner macros of the second level, etc. Open code is processed last.

Programmer Response: Make sure all global declarations are consistent.

IFZ243 INCONSISTENT DIMENSION OF GLOBAL VARIABLE SYMBOL 'xxxxxxxx' IN 'yyyyyyyy'

Explanation: Either the dimensions specified in declaration of global variable symbols are different in different macros and/or open code, or a global symbol is declared as dimensional in one macro definition and undimensional in another.

Assembler Action: All declarations inconsistent with the first declaration encountered are ignored. The macro definitions are processed in the order in which they appear in the source, with all outer macros first, followed by all inner macros of the first level, all inner macros of the second level, etc. Open code is processed last.

Programmer Response: Make sure all global declarations are consistent.

IFZ244 SYSSLB RECORD 'nnn' IN MACRO 'xxxxxxxx' NOT IN SEQUENCE

Explanation: This library macro definition was not in the proper order when it was cataloged; the specified record was out of sequence.

Assembler Action: The macro is not generated.

Programmer Response: Catalog the macro definition again, making sure that all the records are in the right sequence.

IFZ245 MACRO 'xxxxxxxx' CATALOGED UNDER DIFFERENT NAME 'yyyyyyyy'

Explanation: This library macro definition was not cataloged under the right name; the name under which a macro is cataloged must always be identical to the operation code of the macro as it is specified in the macro prototype statement.

Assembler Action: The macro is not generated.

Programmer Response: Catalog the macro under its own name (operation code), or change the operation to match the book name.

IFZ246 UNEXPECTED END-OF-FILE ON SYSSLB AT RECORD 'nnn' IN MACRO 'xxxxxxxx'

Explanation: End-of-file was encountered in the source statement library before the end of a book had been reached, or record length is greater than 80 bytes. Since the end-of-file indicator is normally only found at the end of the sublibrary, the message indicates that the source statement library has been destroyed.

Assembler Action: The macro is not generated.

Programmer Response: Reconstruct the source statement library.

IFZ247 UNEXPECTED END OF BOOK AT RECORD 'nnn' IN LIBRARY MACRO 'xxxxxxxx'

Explanation: Some cards at the end of this definition were missing in this macro definition when it was cataloged.

Assembler Action: The macro is not generated.

Programmer Response: Catalog a complete version of the macro definition.

IFZ248 'xxxxxxxx' NOT AN EDITED MACRO

Explanation: The macro library book that corresponds to the specified operations code is not recognized as an edited macro.

Assembler Action: The macro is not generated.

Programmer Response: Catalog an edited version of the macro definition.

IFZ250 ERRORS FOUND IN MACRO 'xxxxxxxx' EDECK NOT PUNCHED

Explanation: Since errors were found in this macro definition, no edited macro is punched, even though that is requested by means of the EDECK option.

Programmer Response: Correct the errors in the macro definition and assemble again.

Appendix I: Communications Controller Assembler Messages—OS/VS

Component Name	CWA = OS/VS
Program Producing Message	IBM Communications Controller Assembler program during assembly of assembler instructions (under OS/VS)
Audience and Where Produced	For programmer: Assembler listing in SYSPRINT data set For operator: Console
Message Format	ss,***CWAnnn text (in SYSPRINT) xx CWAnnn text (on console) ss Severity code indicating effect on execution of program being assembled: * Informational message; no effect on execution 0 Informational message: normal execution is expected 4 Warning message; normal execution is probable 8 Error; successful execution is possible 12 Serious error; successful execution is improbable 16 Critical error; successful execution is impossible 20 Critical error; further assembly impossible; assembler program execution terminated abnormally nnn Message serial number text Message text xx Message reply identification (absent, if operator reply not required)

CWA000 UNDEFINED ERROR CODE IFOxxx

Explanation: An error code has been generated by the assembler for which no message has been defined. This is caused by a logical error in the assembler.

Assembler Action: Assembly continues.

Programmer Response: Perform the actions described under "Recurring Errors" above before calling IBM.

Severity Code: 16

Module Originating Message: CWAX1A

CWA001 SYSTEM VARIABLE SYMBOL xxxxxxxx USED AS SYMBOLIC PARAMETER IN MACRO PROTOTYPE

Explanation: A variable symbol used as a symbolic parameter on a macro prototype statement has the same characters as a system variable symbol. The system variable symbols are:

&SYSECT	&SYSPARM
&SYSLIST	&SYSTIME
&SYSNDX	&SYSDATE

Assembler Action: Editing of the macro definition is terminated. All statements in the macro definition are processed as comments.

Programmer Response: Redefine the parameter with a variable symbol other than &SYSPARM, &SYSDATE, &SYSTIME, &SYSLIST, &SYSECT, or &SYSNDX.

Severity Code: 8

Module Originating Message: CWAX1J

CWA002 SYMBOLIC PARAMETER xxxxxxxx IS DUPLICATED IN SAME MACRO PROTOTYPE

Explanation: Two identical symbolic parameters have been specified in the same macro prototype statement.

Assembler Action: Editing of the macro definition is terminated. All statements in the macro definition are processed as comments.

Programmer Response: Redefine one of the symbolic parameters with a variable symbol that is unique to that particular macro definition.

Severity Code: 8

Module Originating Message: CWAX1J

CWA003 SYSTEM VARIABLE SYMBOL xxxxxxxx USED IN OPERAND OF GLOBAL OR LOCAL DECLARATION

Explantion: A system variable symbol has been used in the operand of a global or local declaration. The system variable symbols are:

&SYSECT	&SYSPARM
&SYSLIST	&SYSTIME
&SYSNDX	&SYSDATE

Assembler Action: The declaration conflicting with the system variable symbol is ignored. All subsequent references to the variable symbol in error are treated as references to the system variable symbol.

Programmer Response: Redefine the variable symbol using character combinations other than those listed above in the explanation.

Severity Code: 8

Module Originating Message: CWAX1J

CWA0004 GLOBAL OR LOCAL VARIABLE xxxxxxxx DUPLICATES A SYMBOLIC PARAMETER IN SAME MACRO DEFINITION

Explanation: A variable symbol that appears in the operand field of a global or local declaration is identical to a symbolic parameter defined on the macro prototype earlier in the macro defintion.

Assembler Action: The declaration conflicting with the symbolic parameter is ignored. All subsequent references to it are treated as references to the symbolic parameter that it duplicates.

Programmer Response: Redefine the global or local variable with a variable symbol that is unique to the macro definition.

Severity Code: 8

Module Originating Message: CWAX1J

CWA005 GLOBAL OR LOCAL VARIABLE SYMBOL xxxxxxxx DUPLICATES PREVIOUS DEFINTION

Explanation: A global or local variable symbol was declared twice in the same macro definition or in open code.

Assembler Action: The second declaration of the variable symbol is ignored. All subsequent references to it are treated as references to the first declaration.

Programmer Response: If the second declaration is LCLx, redeclare it using a variable symbol unique to the macro definition or to open code. If the second declaration is GBLx, redeclare it as for LCLx, but be sure that all declarations of that global variable elsewhere in the program are identical.

Severity Code: 8

Module Originating Message: CWAX1J

CWA006 UNDEFINED VARIABLE SYMBOL xxxxxxx

Explanation: A variable symbol has been referenced in this statement that is not a system variable symbol; has not been defined within the macro definition as a symbolic parameter, a local variable, or a global variable; or has not been defined in open code as a local or global variable.

Assembler Action: The statement is processed as a comment, unless the error has occurred in a macro instruction parameter. If the macro instruction parameter contains an undefined variable symbol, the parameter is assigned the value of a null string.

Programmer Response: Define the variable symbol as a symbolic parameter, a local variable, or a global variable; or, if desired, reference a previously-defined variable symbol of the appropriate type. This message may be issued if an ampersand erroneously appears as the first character of an ordinary symbol, and thus creates an unintended variable symbol.

Severity Code: 8

Module Originating Message: CWAX1J

CWA007 USAGE OF xxxxxxxx IS INCONSISTENT WITH ITS DECLARATION

Explanation: A global or local variable symbol was defined as dimensioned but was used without a subscript, or a global or local variable symbol was defined as undimensioned but was used with a subscript.

Assembler Action: Editing of the statement that contains the inconsistent usage is terminated, and the statement is processed as a comment.

Programmer Response: Make the usage of the SET symbol consistent with its global or local declaration, or make the declaration of the SET symbol consistent with its usage.

Severity Code: 8

Module Originating Message: CWAX1J

CWA008 CIRCULAR OPSYN STATEMENTS

Explanation: The assignment of a synonym in the operand field of an OPSYN statement to the established mnemonic in the name field results in the mnemonic being its own synonym. For example:

```
ADD     OPSYN A
PLUS    OPSYN ADD
XYZ     OPSYN PLUS
ADD     OPSYN XYZ
```

The final OPSYN statement in the above sequence is flagged.

Assembler Action: The flagged OPSYN statement is processed as a comment.

Programmer Response: Remove any OPSYN statement that results in a circular definition, or alter such an OPSYN statement by respecifying the synonym or the mnemonic.

Severity Code: 8

Module Originating Message: CWAX1J

CWA009 EDIT DICTIONARY SPACE EXHAUSTED

Explanation: The work space available is not sufficient to contain the dictionaries that are required to edit the macro definition or open code.

Assembler Action: If a macro definition is being edited, the remaining statements up to the MEND statement are processed as comments, and editing resumes. If open code is being edited, the remaining statements up to the end-of-file are processed as comments.

Programmer Response: Increase the size of the region or partition that is allocated to assembly, or allocate more dictionary space via the BUFSIZE assembler option.

Severity Code: 12

Module Originating Message: CWAX1J

CWA010 SOURCE MACRO xxxxxxxx HAS BEEN PREVIOUSLY DEFINED

Explanation: The mnemonic in the macro instruction prototype of a source macro duplicates a mnemonic already defined as a source macro.

Assembler Action: All statements in this macro definition are processed as comments. All subsequent references to the mnemonic are treated as references to the first definition associated with that op code.

Programmer Response: Provide a unique mnemonic op code for the flagged macro prototype.

Severity Code: 8

Module Originating Message: CWAX1A

CWA012 ICTL OR OPSYN STATEMENT APPEARS TOO LATE IN THE PROGRAM

Explanation:

- The ICTL statement does not precede all other statements in the source module; or
- The OPSYN statement does not appear before source macro definitions and open code statements. The only statements that can precede an OPSYN statement are: ICTL, ISEQ, TITLE, PRINT, EJECT, SPACE, OPSYN, COPY (unless the member copied contains any other than the statements listed here), and comments statements.

Assembler Action: The ICTL or OPSYN statement is processed as a comment.

Programmer Response: Place the ICTL or OPSYN statement at the beginning of your program as described in the explanation above.

Severity Code: 8

Module Originating Message: CWAX1A

CWA013 OPSYN NAME FIELD NOT ORDINARY SYMBOL, OR OPSYN OPERAND FIELD NOT ORDINARY SYMBOL OR BLANK

Explanation: The name or operand field of an OPSYN instruction contains more than 8 alphanumeric characters or does not begin with an alphabetic character.

Assembler Action: The OPSYN statement is processed as a comment.

Programmer Response: Correct the invalid name field or operand field.

Severity Code: 8

Module Originating Message: CWAX1A

CWA014 INVALID OPCODE IN OPSYN OPERAND OR NAME FIELD

Explanation:

- The name field of an OPSYN instruction with a blank operand field does not specify a machine instruction operation code, an extended machine instruction operation code, or an assembler operation code; or
- The operand field of an OPSYN instruction does not specify a machine instruction operation code, an extended machine instruction operation code, or an assembler operation code.

Assembler Action: The OPSYN statement is treated as a comment.

Programmer Response: Make sure that the name field contains a valid operation code, or supply a valid operation code in the operand.

Severity Code: 8

Module Originating Message: CWAX1J

CWA015 EXPRESSION VALUE EXCEEDS 262143 NEAR OPERAND COLUMN nn

Explanation:

- An expression has been detected whose value exceeds 262143, near column nn.

Assembler Action: The statement is set to zero.

Programmer Response: Correct the expression so that its value will be equal to or less than 262,143.

Severity Code: 8

Module Originating Message: CXAX5V

CWA016 ILLEGAL OR INVALID NAME FIELD

Explanation: One of the following errors was detected.

- No name was found where one is required.
- A name was supplied where none is allowed.
- An invalid character was found in the name field.

Assembler Action: The statement is processed as a comment, unless the error has occurred in the name field of a macro instruction. If the macro name field parameter contains an error, the parameter is assigned the value of a null string.

Programmer Response: Supply a name if one is required, omit the name if one is not allowed, or correct the invalid character.

Severity Code: 12

Modules Originating Message: CWAX1A, CWAX3A

CWA017 *COMMENT STATEMENT IS ILLEGAL OUTSIDE MACRO DEFINITION

Explanation: An internal macro comments statement (.*) appears outside macro definitions (in open code).

Assembler Action: The statement is printed.

Programmer Response: Remove the .* comments statement. If you want a comment, put an * in the begin column and follow it by the comment.

Severity Code: 4

Module Originating Message: CWAX1A

CWA018 MORE THAN 5 ERRORS ON THIS STATEMENT, ERROR ANALYSIS OF THIS STATEMENT IS TERMINATED

Explanation: The maximum number of error messages issued during editing to each statement is 5. The sixth error causes this message.

Assembler Action: Error analysis for this statement is terminated.

Programmer Response: Correct the indicated errors and reassemble. Any additional errors on this statement will be detected in the next assembly.

Severity Code: 4

Module Originating Message: CWAX1A

CWA019 INVALID OPERAND ON ICTL OR ISEQ STATEMENT

Explanation:

1. The value of one or more operands in an ICTL statement is incorrect. The begin column must be within columns 1 to 40; the end column must be within columns 41 to 80 and at least 5 columns away from the begin column; and the continue column must be within columns 2 to 40.
2. One of the following errors has occurred in an ISEQ statement:
   - The operand has an illegal range; the operand value cannot fall between the begin and end columns, and the second operand must not be less than the first.
   - The operand field is invalid. The operand field must contain two valid decimal self-defining terms, separated by a comma or be blank.

Assembler Action: If a program contains an ICTL error, the whole program is processed as comments. If one of the ISEQ errors has occurred, no sequence checking is performed.

Programmer Response: Supply valid operand(s).

Severity Code: 8

Module Originating Message: CWAX1A

CWA020 INVALID BRANCH ADDRESS—DISPLACEMENT EXCEEDS 2046

Explanation: A machine instruction with a 'T field' has a displacement which is greater than 2046 or less than -2046.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Make sure that the displacement (in respect to the address of the next sequential instruction after the branch instruction) is less than or equal to 2046 or greater than or equal to -2046.

Severity code: 8

Module Originating Message: CWAX5M

CWA021 INVALID TERM IN OPERAND

Explanation: An invalid term has been used in an expression of the operand.

Assembler Action: The statement is processed as a comment.

Programmer Response: Make sure the operand is a character relation, an arithmetic relation, a logical relation, a SETx symbol, a symbolic parameter, or a decimal self-defining term.

Severity Code: 8

Module Originating Message: CWAX1A

CWA022 ICTL STATEMENT IS ILLEGAL IN COPY CODE

Explanation: An ICTL statement appears in code that is inserted in the program by a COPY instruction.

Assembler Action: The ICTL statement is processed as a comment.

Programmer Response: Make sure the ICTL instruction is not in code inserted by the COPY instruction. If used, the ICTL instruction must always be the first instruction in your source module.

Severity Code: 8

Module Originating Message: CWAX1A

CWA023 ILLEGAL MACRO, MEND, OR MEXIT STATEMENT—MAY APPEAR ONLY WITHIN MACRO DEFINITIONS

Explanation: MACRO, MEND, or MEXIT statements are not allowed in open code. They can be used only in macro definitions. This message will be issued if an instruction other than ICTL, ISEQ, OPSYN, TITLE, PRINT, EJECT, SPACE, or COPY appears before any macro definitions in your program. Of course, any such COPY instruction cannot copy any other statements than ISEQ, OPSYN, TITLE, PRINT, EJECT, or SPACE. This message will also be issued, if an undefined operation code appears before your macro definitions.

Assembler Action: The invalid MACRO, MEND, or MEXIT statement is processed as a comment.

Programmer Response: Remove the statement from open code or place it within a macro definition. Make sure that all your macro definitions are placed at the beginning, before open code.

Severity Code: 8

Module Originating Message: CWAX1A

CWA024 UNPAIRED PARENS, OR BLANK FOUND INSIDE PAIRED PARENS

Explanation:

- Unpaired parentheses appear in the operand field; or
- A blank appears inside paired parentheses in the operand field of a macro instruction. This may be an error in sublist structure; or
- A blank appears inside parentheses of an arithmetic expression; or
- A term is missing in a logical expression.

Assembler Action: The operand in error is ignored.

Programmer Response: If unpaired parentheses appear, be sure that there is a right parenthesis for every left parenthesis. Remove illegal blanks inside paired parentheses.

Severity Code: 8

Module Originating Message: CWAX1A

CWA025 STATEMENT OUT OF SEQUENCE

Explanation: The input sequence checking specified by the ISEQ instruction has determined that the flagged statement is out of sequence.

Assembler Action: The statement is flagged and assembled, however, the sequence number of the following statements will be checked relative to this statement and not relative to the sequence of previous statements.

Programmer Response: Put the statement in the proper sequence.

Severity Code: 4

Module Originating Message: CWAX1A

CWA026 CHARACTERS APPEAR BETWEEN THE BEGIN AND CONTINUE COLUMNS ON CONTINUATION CARD

Explanation: On a continuation card, the begin column and all columns between the begin column and the continue column (usually column 16) must be blank.

Assembler Action: Characters that appear between the begin column and the continue column are ignored.

Programmer Response: Determine whether the operand started in the wrong continue column or whether the preceding card contained an erroneous continue punch in column 72.

Severity Code: 4

Module Originating Message: CWAX1A

CWA027 ICTL, ISEQ, MACRO, OR OPSYN STATEMENT APPEARS IN MACRO DEFINITION

Explanation: One of the specified operations is used within a macro definition, which is invalid.

Assembler Action: The illegal operation is ignored and the statement is processed as a comment.

Programmer Response: Remove all ICTL, ISEQ, MACRO, and OPSYN statements from within macro definitions. Make sure your ICTL and OPSYN instructions precede your macro definitions, and that each macro definition ends with a MEND statement.

Severity Code: 8

Module Originating Message: CWAX1A

CWA028 ILLEGAL PROTOTYPE KEYWORD PARAMETER DEFAULT VALUE

Explanation: A variable symbol is used as the default value of a keyword parameter.

Assembler Action: The statement is ignored.

Programmer Response: Supply a valid default value for the keyword parameter.

Severity Code: 8

Module Originating Message: CWAX1A

CWA029 xxxxxxxx IS AN ILLEGAL OPERAND IN A GLOBAL OR LOCAL DECLARATION

Explanation: In a global (GBLx) or local (LCLx) SET symbol declaration, the indicated operand does not consist of one or more variable symbols that are separated by commas and terminated with a blank.

Assembler Action: The attempted global or local SET symbol declaration is processed as a comment. Recovery is made in certain circumstances and some valid variable symbols in the declaration are recognized and defined correctly.

Programmer Response: Supply the operand with valid variable symbols and delimiters. Check all global and local declarations.

Severity Code: 8

Module Originating Message: CWAX1A

CWA030 DECLARED DIMENSION OF xxxxxxxx IS ILLEGAL

Explanation: The declared dimension, which appears in the error message, must be a nonzero, unsigned decimal integer, not greater than 32,767, and enclosed in parentheses.

Assembler Action: If the declared dimension was a decimal self-defining term greater than 32,767, a default dimension of 32,767 is assigned to the variable symbol. In all other cases, the variable symbol declaration is ignored.

Programmer Response: Supply a valid dimension.

Severity Code: 8

Module Originating Message: CWAX1A

CWA031 SET STATEMENT NAME NOT A VARIABLE SYMBOL, OR SET STATEMENT NAME INCONSISTENT WITH DECLARED TYPE

Explanation:

1. The name field of a SET statement does not consist of an ampersand followed by from 1 to 7 alphameric characters, the first of which is alphabetic.
2. The symbol does not match its previously declared type. For instance, the symbol might have been previously defined as LCLA, but the flagged statement may have tried to assign a SETC character string to it.
3. A system variable symbol appears in the name field of a SETx instruction. The system variable symbols are &SYSECT, &SYSLIST, &SYSNDX, &SYSPARM, &SYSDATE, and &SYSTIME.

Assembler Action: The flagged statement is processed as a comment.

Programmer Response: Assign a valid variable symbol to the name field of the SET statement (the symbol must be previously defined as a global or local variable), or be sure that the usage of the symbol corresponds to its previously declared type.

Severity Code: 8

Module Originating Message: CWAX1A

CWA032 xxxxxxxx APPEARS IMPROPERLY IN THE OPERAND OF THIS STATEMENT

Explanation: The specified operand part is invalid.

Assembler Action: The statement is processed as a comment.

Programmer Response: Check the syntax required for the operand field of this statement, and supply a valid operand.

Severity Code: 8

Module Originating Message: CWAX1A

CWA033 xxxxxxxx IS AN INVALID LOGICAL OPERATOR

Explanation: The specified character string was found where a logical operator (AND or OR) was expected.

Assembler Action: The statement is processed as a comment.

Programmer Response: Use either AND or OR, as appropriate, for the logical operator.

Severity Code: 8

Module Originating Message: CWAX1A

CWA034 INVALID BRANCH ADDRESS—DISPLACEMENT EXCEEDS 126

Explanation: A machine instruction with a 'T field' has a displacement which is greater than 126 or less than -126.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Make sure that the displacement (in respect to the address of the next sequential instruction after the branch instruction) is less than or equal to 126 or greater than or equal to -126.

Severity Code: 8

Module Originating Message: CWAX5M

CWA035 QUOTES NOT PAIRED, OR ILLEGAL TERMINATION OF QUOTED STRING

Explanation: The quotes in the operand field of this statement are unpaired, or the string is invalidly terminated.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply any missing quotes.

Severity Code: 8

Modules Originating Message: CWAX1A, CWAX5D

CWA036 ATTRIBUTE REFERENCE FOR xxxxxxxx IS INVALID

Explanation: The flagged statement has attempted to reference a symbol that is not a valid ordinary or variable symbol. The attributes referenced were one or more of the following: type (T'), length (L'), scaling (S'), integer (I'), count (K'), and number (N').

Assembler Action: The attribute referenced is ignored, and/or the statement is ignored, and/or default values for type, length, and scaling attributes are supplied.

Programmer Response: Determine if a clerical error was made in coding either the reference or the definition of the symbol that appears in the message text; or supply a valid ordinary or variable symbol where necessary.

Severity Code: 8

Module Originating Message: CWAX1A

CWA037 xxxxxxxx IS AN ILLEGAL SUBSCRIPT

Explanation: The subscript that appears in the message text either is not enclosed by paired parentheses, or is an invalid subscript.

Assembler Action: The statement that contains the invalid subscript is processed as a comment.

Programmer Response: Be sure the parentheses are paired, and that a valid subscript appears inside them.

Severity Code: 8

Module Originating Message: CWAX1A

CWA038 xxxxxxxx IS AN INVALID SELF-DEFINING TERM

Explanation: The characters specified in the message are invalid in the operand field of a binary (type B), character (type C), decimal, or hexadecimal (type X) self-defining term.

Assembler Action: The statement that contains the invalid self-defining term is processed as a comment.

Programmer Response: Make sure that the characters used for a self-defining term are consistent with the type of term.

Severity Code: 8

Module Originating Message: CWAX1A

CWA039 xxxxxxxx IS AN INVALID VARIABLE SYMBOL

Explanation: The specified symbol does not consist of an ampersand followed by from 1 to 7 alphameric characters, the first of which is alphabetic.

Assembler Action: The statement that contains the invalid variable symbol is processed as a comment. If the statement is a macro prototype statement, all statements in the macro definition are treated as comments.

Programmer Response: Supply a valid variable symbol, or check that a single ampersand is not used where a double ampersand is needed.

Severity Code: 8

Module Originating Message: CWAX1A

CWA040 INVALID M-FIELD NEAR OPERAND COLUMN nn

Explanation: The binary value of the three-bit n-field in the Branch-on-Bit instruction (BB) did not specify bits 0-7.

Assembler Action: The instruction is set to zero.

Programmer Response: Correct the m-field designator to indicate bits 0 to 7 only.

Severity Code: 8

Module Originating Message: CWAX5M

CWA041 M-FIELD GREATER THAN 7 NEAR OPERAND COLUMN nn

Explanation: The binary value of the three-bit n-field in the Branch-on-Bit instruction (BB) is greater than 7.

Assembler Action: The statement is set to zero.

Programmer Response: Correct the m-field designator to indicate bits 0 to 7 only.

Severity Code: 8

Module Originating Message: CWAX5M

CWA042 PARAMETER IN MACRO PROTOTYPE OR MACRO INSTRUCTION EXCEEDS 255 CHARACTERS

Explanation: A parameter value that appears in the operand fields of either a macro prototype or a macro instruction exceeds 255 characters in length.

Assembler Action: The first 255 characters of the parameter are deleted. The remaining characters are used as the parameter value.

Programmer Response: Limit the parameter to 255 characters or separate it into two or more parameters.

Severity Code: 8

Module Originating Message: CWAX1A

CWA043 MACRO INSTRUCTION PROTOTYPE STATEMENT HAS INVALID OP CODE

Explanation:

- The operation code of a macro prototype statement is previously defined as the operation code of a machine, assembler, or macro instruction; or
- The operation code of a macro prototype statement is not a valid ordinary symbol; that is, it does not consist of a letter, followed by 0 to 7 letters or digits or both.

Assembler Action: The entire macro definition is processed as comments.

Programmer Response: Supply a valid ordinary symbol that does not conflict with any machine, assembler, or macro instruction operation code.

Severity Code: 8

Module Originating Message: CWAX1A

CWA046 STATEMENT COMPLEXITY EXCEEDED

Explanation: The expression evaluation work area has overflowed because the expression is too complex. The complexity of an expression is determined by the number of nested operators and levels of parentheses. Up to 35 operators and levels of parentheses are allowed. For logical expressions, this total allows 18 unary and binary operators, and 17 levels of parentheses. For arithmetic expressions in conditional assembly, the total allows 24 unary and binary operators, and 11 levels of parentheses.

Assembler Action: The statement is processed as a comment.

Programmer Response: Simplify the expression to the limits described in the explanation.

Severity Code: 8

Module Originating Message: CWAX1A

CWA047 UNEXPECTED END OF FILE ON SYSTEM INPUT (SYSIN)

Explanation:

- A continuation record was expected when an end-of-file occurred on SYSIN (the source program ended); or
- End-of-file immediately follows a REPRO statement; or
- End-of-file occurs before an END card has been read.

Assembler Action: An END statement is generated and assembly continues.

Programmer Response: Determine if any statements were omitted from the program.

Severity Code: 4

Module Originating Message: CWAX1A

CWA048 ICTL STATEMENT HAS NO OPERAND

Explanation: The ICTL statement requires an operand, but none is present.

Assembler Action: The entire source module is processed as comments.

Programmer Response: Supply from 1 to 3 decimal self-defining terms to indicate respectively the begin, end, and continue columns. If the ICTL statement is omitted, columns 1, 71, and 16, respectively, are the default values.

Severity Code: 8

Module Originating Message: CWAX1A

CWA049 COPY STATEMENT OPERAND NOT A VALID ORDINARY SYMBOL

Explanation: The operand of a COPY statement is not a symbol of 1 to 8 alphameric characters, the first of which is alphabetic.

Assembler Action: The COPY request is processed as a comment.

Programmer Response: Supply a valid ordinary symbol in the operand field.

Severity Code: 8

Module Originating Message: CWAX1A

CWA050 COPY STATEMENT DOES NOT HAVE AN OPERAND

Explanation: No operand found on this COPY statement.

Assembler Action: The statement is processed as a comment.

Programmer Response: Place the name of a member to be copied in the operand field, or remove the COPY statement.

Severity Code: 8

Module Originating Message: CWAX1A

CWA051 UNEXPECTED END OF DATA ON SYSTEM LIBRARY (SYSLIB)

Explanation: An end-of-file occurred on the input from a system library before a MEND statement terminating a macro definition was encountered.

Assembler Action: The missing MEND statement is generated.

Programmer Response: Determine if the MEND statement was omitted from the library macro, or if the library contains an otherwise incomplete macro definition, or if a macro call has been made to a non-macro definition.

Severity Code: 4

Module Originating Message: CWAX1A

CWA052 UNARY OPERATOR NOT A PLUS OR MINUS SIGN

Explanation: An operator other than a plus or minus sign appears as a unary operator. Except for unary operators, which are limited to plus and minus signs, only one operator can appear between two terms.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply the missing term or a correct operator.

Severity Code: 8

Module Originating Message: CWAX1A

CWA053 OP CODE NOT FOUND ON FIRST OR ONLY CARD

Explanation: The complete statement name (if one is used) and the operation code, each followed by a blank, do not appear before the continuation indicator column on the first card of a continued statement.

Assembler Action: The entire statement is processed as a comment.

Programmer Response: Make sure that both the name and operation code of the statement appear on the first card. Check for syntactic errors.

Severity Code: 8

Module Originating Message: CWAX1A

CWA054 INVALID OPERATION CODE

Explanation:

- The operation code specified is not a valid ordinary symbol; or
- A variable symbol in the operation field is invalid; or
- The resulting operation code after substitution with or without concatenation is not a valid ordinary symbol.

Assembler Action: The statement is processed as a comment.

Programmer Response: Make sure that ordinary or variable symbols used in the operation field are valid. If you use variable symbols with or without concatenation, make sure the resulting symbol is a valid ordinary symbol.

Severity Code: 8

Module Originating Message: CWAX1A

CWA055 MEND STATEMENT GENERATED

Explanation: An end-of-file occurred on the input from the system input device (SYSIN) or the system library (SYSLIB) before a MEND statement terminating a macro definition was encountered.

Assembler Action: A MEND statement is generated.

Programmer Response: Supply a MEND statement to terminate the macro definition.

Severity Code: 8

Module Originating Message: CWAX1A

CWA057 DUPLICATION FACTOR xxxxxxxx IN SETC EXPRESSION NOT TERMINATED BY A RIGHT PARENTHESIS

Explanation: A SETC operand begins with a left parenthesis, but a comma, a period, or a blank appears before the closing right parenthesis.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply a right parenthesis.

Severity Code: 8

Module Originating Message: CWAX1A

CWA058 NO ENDING QUOTE ON SETC EXPRESSION

Explantion: The character expression in the operand field of a SETC statement must be enclosed in quotes. The statement ends before a delimiting quote.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply any missing quotes.

Severity Code: 8

Module Originating Message: CWAX1A

CWA059 INVALID TERM IN LOGICAL EXPRESSION

Explanation: One of the terms in the logical expression is invalid in the context.

Assembler Action: The statement is processed as a comment.

Programmer Response: Make sure that the terms in the logical expression are valid.

Severity Code: 8

Module Originating Message: CWAX1A

CWA060 END STATEMENT GENERATED

Explanation: One of two errors occurred.

1. End-of-file occurred on the system input device (SYSIN) before an END card was read.
2. The ACTR limit was exceeded in open code.

Assembler Action: An END statement is generated.

Programmer Response:

1. Supply a valid END statement; or
2. Either correct the conditional assembly loop in open code so that the ACTR limit is not exceeded, or set the ACTR limit in open code to a higher value.

Severity Code: 4

Modules Originating Message: CWAX1A, CWAX3A, CWAX5A

CWA061 COPY NEST GREATER THAN FIVE

Explanation: The maximum limit of five nested levels of COPY statements is exceeded.

Assembler Action: COPY processing terminates.

Programmer Response: Eliminate excessive levels of COPY statements.

Severity Code: 8

Module Originating Message: CWAXIA

CWA062 REQUIRED OPERAND FIELD MISSING

Explanation: This statement requires an operand in the operand field and none is present.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply the missing operand.

Severity Code: 8

Modules Originating Message: CWAX1A, CWAX5D

CWA063 VALUE OF EQO MUST BE POSITIVE—VALUE SET TO ZERO

Explanation: The expression to the EQU has the high-order bit turned on. The value of the EQU can only be 0–262,143 (2^{18}-1).

Assembler Action: The statement is processed as an EQU 0.

Programmer Response: Make the absolute value or the relocatable value of the expression positive.

Severity Code: 8

Module Originating Message: CWAX5A

CWA064 INTERLUDE DICTIONARY SPACE EXHAUSTED

Explanation: The work space available is not sufficient to contain the dictionaries required to build either:

1. The skeleton dictionary for a macro definition or all of open code, or
2. The ordinary symbol attribute reference dictionary.

This message is always logged against statement number 0.

Assembler Action: If a macro is being processed, building of the skeleton dictionary for that macro definition is terminated and the macro will not be expanded. If open code is being processed, the building of the open code skeleton dictionary is terminated and the program is processed as comments. If space for the ordinary symbol attribute reference dictionary is exhausted, the building of it is abandoned.

Programmer Response: Within the partition, increase the size of the region that is allocated to assembly, or allocate more of the partition to dictionary space via the BUFSIZE assembler option.

Severity Code: 12

Module Originating Message: CWAX2A

CWA065 EXPRESSION 2 OF EQU SYMBOL xxxxxxxx NOT IN RANGE 0-65535

Explanation: The value of the expression specified in the second operand of the EQU instruction where this symbol is defined is not in the range 0-65535.

This message is always logged against statement number 0.

Assembler Action: The length attribute of the symbol is set to 1.

Programmer Response: Make sure the value of the second operand of the EQU instruction is in the range 0-65535, or delete the second operand.

Severity Code: 8

Module Originating Message: CWAX2A

CWA066 EXPRESSION 3 OF EQU SYMBOL xxxxxxxx NOT IN RANGE 0-255

Explanation: The value of the expression specified in the third operand of the EQU instruction where this symbol is defined is not in the range 0-255.

This message is always logged against statement number 0.

Assembler Action: The type attribute of the symbol is set to U.

Programmer Response: Make sure the value of the third operand of the EQU instruction is in the range 0-255, or delete the third operand.

Severity Code: 8

Module Originating Message: CWAX2A

CWA067 DECLARED DIMENSION FOR GLOBAL VARIABLE xxxxxxxx IN xxxxxxxx xxxxxxxx IS INCONSISTENT

Explanation: The declared dimension of a global variable defined in a macro definition or in open code is not consistent with the declared dimension of the same global variable in another macro definition or in open code.

This message is always logged against statement number 0. The message text identifies the macro (or open code) where the error is found.

Assembler Action: All references to the global variable in the macro definition or in open code where the inconsistency was detected result in a null (zero) value.

Programmer Response: Be sure that all definitions of a given global variable have the same declared dimension.

Severity Code: 4

Module Originating Message: CWAX2A

CWA068 COPY MEMBER xxxxxxxx NOT FOUND IN LIBRARY

Explantion: The COPY member shown in the message text was not found in the library.

Assembler Action: The COPY statement is processed as a comment.

Programmer Response: Determine whether the library member name is misspelled or whether an incorrect member name was referenced. Make sure the proper macro library is assigned in your JCL statements.

Severity Code: 8

Module Originating Message: CWAX1A

CWA069 TOO MANY CONTINUATION CARDS, TWO ALLOWED

Explanation: Only two continuation cards are allowed for each statement, except for macro definition prototype and macro call statements.

Assembler Action: Excess continuation cards are processed as comments.

Programmer Response: Restructure the statement so that it can be contained on a total of three cards. Extensive remarks may be recorded as comment statements by coding an asterisk in column 1 and eliminating the continuation indicators.

Severity Code: 4

Module Originating Message: CWAX1A

CWA070 SUBSTRING NOTATION IS NOT DELIMITED BY COMMA OR RIGHT PARENTHESIS

Explanation: Two SETA expressions used in substring notation are not separated by a comma or enclosed in parentheses.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply the missing delimiter, or check for other syntax errors that make this appear as substring notation.

Severity Code: 8

Module Originating Message: CWAX1A

CWA073 AGO OR AIF OPERAND NOT A SEQUENCE SYMBOL

Explanation: The symbol in the operand field of an AIF or AGO statement is not a period (.) followed by from 1 to 7 alphameric characters, the first of which is alphabetic.

Asssembler Action: The statement is processed as a comment.

Programmer Response: Supply a valid sequence symbol.

Severity Code: 8

Module Originating Message: CWAX1A

CWA074 SEQUENCE SYMBOL xxxxxxxx IS MULTIPLY DEFINED IN xxxxxxxx xxxxxxxx

Explanation: The sequence symbol in the name field has been used in the name field of a previous statement within the same macro definition or open code.

This message is always logged against statement number 0. The message text identifies the macro (or open code) where the error is found.

Assembler Action: All definitions of the sequence symbol after the first one are ignored. All references to the sequence symbol are treated as references to the first definition.

Programmer Response: Provide unique sequence symbols for the macro definition or open code.

Severity Code: 4

Module Originating Message: CWAX2A

CWA076 SEQUENCE SYMBOL xxxxxxxx IS UNDEFINED IN xxxxxxxx xxxxxxxx

Explanation: A sequence symbol appears in the operand of an AIF or AGO statement, but does not appear in the name field of another statement in the same macro definition or open code.

This message is always logged against statement number 0. The message text identifies the macro (or open code) where the error is found.

Assembler Action: All statements which reference the undefined sequence symbol are processed as comments.

Programmer Response: Define the sequence symbol at the appropriate point, or reference a sequence symbol that is already defined.

Severity Code: 4

Module Originating Message: CWAX2A

CWA078 UNDEFINED OP CODE

Explanation: The mnemonic operation code of this statement does not correspond to any of the following:

- a machine instruction operation code
- an extended machine instruction operation code
- an assembler instruction operation code
- a macro instruction operation code
- an operation code that has been defined by an OPSYN instruction.

This message is also issued for operation codes that have been deleted by OPSYN instructions.

Assembler Action: The statement is treated as a comment. If the statement appears before open code, all statements following it are considered to belong to open code. This means that any macro definitions following the error are treated as errors.

Programmer Response: Either make sure you use a valid mnemonic operation code, or make sure that the proper OPSYN instructions are included in your program.

Severity Code: 12

Module Originating Message: CWAX3A

CWA080 ATTRIBUTE REFERENCE TO UNDEFINED SYMBOL

Explanation: The symbol specified in a length (L'), scaling (S'), or integer (I') attribute reference is either an undefined symbol or a symbolic parameter (or a &SYSLIST specification) representing an undefined symbol.

Assembler Action:

- The length attribute, if specified, is set to 1.
- The integer or scaling attribute, if specified, is set to 0.

Programmer Response: Make sure the symbol is defined.

Severity Code: 4

Module Originating Message: CWAX3A

CWA081 DECLARED TYPE FOR GLOBAL VARIABLE xxxxxxxx IN xxxxxxxx xxxxxxxx IS INCONSISTENT

Explanation: The type (GBLA, GBLB, or GBLC) of a global variable declared in a macro definition or in open code is not consistent with the type of the same global variable declared in another macro definition or in open code.

This message is always logged against statement number 0. The message text identifies the macro (or open code) where the error is found.

Assembler Action: All references to the global variable in the macro definition or in open code where the inconsistency was detected result in a null (zero) value.

Programmer Response: Make all declarations of the same global variable consistent.

Severity Code: 4

Module Originating Message: CWAX2A

CWA085 MACRO HEADER MISSING, MACRO NOT EXPANDABLE

Explanation: The first statement of a libarary macro definition was not a MACRO statement, and the search for the macro definition is terminated.

Assembler Action: The macro call is processed as a comment.

Programmer Response: Be sure that the libarary macro definition begins with a MACRO statement.

Severity Code: 8

Module Originating Message: CWAX3A

CWA087 INVALID MACRO DEFINITION PROTOTYPE, MACRO NOT EXPANDABLE

Explanation: A comment statement appears immediately after a macro header (MACRO statement).

Assembler Action: All the statements of the macro definition are processed as comments.

Programmer Response: Make sure that the statement immediately following the macro header is a macro prototype statement. No comments or any other statements are permitted between the macro header and the prototype of a macro definition.

Severity Code: 8

Modules Originating Message: CWAX1A, CWAX3A

CWA088 LIBRARY MACRO PROTOTYPE DOES NOT MATCH MEMBER NAME—MACRO NOT EXPANDABLE.

Explanation: The mnemonic operation code in the macro prototype in a library macro definition does not match the entry in the macro library.

Assembler Action: The macro instruction is processed as a comment.

Programmer Response: Enter the macro definition in the library under the same name as the mnemonic op code that appears on the macro prototype.

Severity Code: 8

Module Originating Message: CWAX3A

CWA089 GENERATION-TIME DICTIONARY SPACE EXHAUSTED

Explanation: The workspace available is not sufficient to contain the dictionaries required to expand the macro, to extend a SETC variable, or to contain the basic global dictionaries.

Assembler Action: If the global dictionary workspace is insufficient, the text is processed as comments. If there is insufficient space to extend the SETC variable, expansion of the macro that contains the variable is terminated. If the space for macro definition dictionaries is insufficient, calls to those macros are not expanded.

Programmer Response: Within the partition, increase the size of the region that is allocated to assembly, or allocated more of the partition to dictionary space via the BUFSIZE assembler option.

Severity Code: 12

Module Originating Message: CWAX3N

CWA090 UNDEFINED SEQUENCE SYMBOL ENCOUNTERED DURING CONDITIONAL ASSEMBLY

Explanation: A sequence symbol referenced in the operand field of this statement is undefined in the macro definition or open code. This statement has been encountered during conditional assembly.

Assembler Action: The statement is processed as a comment.

Programmer Response: Define the sequence symbol at an appropriate point, or reference a sequence symbol that is already defined.

Severity Code: 8

Module Originating Message: CWAX3A

CWA091 KEYWORD PARAMETER xxxxxxxx IS DUPLICATED ON SAME MACRO CALL

Explanation: A keyword parameter has appeared more than once on the same macro instruction.

Assembler Action: The last value assigned to the parameter is used, the other value(s) are ignored.

Programmer Response: Define only one value for each parameter.

Severity Code: 8

Module Originating Message: CWAX3N

CWA092 KEYWORD PARAMETER xxxxxxxx UNDEFINED IN MACRO DEFINITION

Explanation: A keyword parameter has been used in the macro instruction that is not a keyword parameter in the macro prototype, or an equal sign not surrounded by quotes is found in a positional parameter.

Assembler Action: The extra keyboard parameter in the macro instruction is ignored.

Programmer Response:

1. Delete the keyword parameter and its value from the macro instruction; or
2. make the keyword parameter in the macro call correspond to one of the keyword parameters in the macro prototype; or
3. define the keyword parameter in the operand field of the macro prototype; or
4. if you wish to include an equal sign in a positional parameter, enclose the parameter within single quotes.

Severity Code: 8

Module Originating Message: CWAX3N

CWA100 DICTIONARY SPACE EXHAUSTED, NO SKELETON DICTIONARY BUILT

Explanation:

- If the message is given for a macro definition or for open code: no available space is left to build the skeleton dictionary after space has been used for the definition of global symbols.
- If the message is given for a macro instruction; dictionary space was exhausted during the editing of a library macro.

Assembler Action: The macro is not considered defined, and any calls to it are processed as comments. If the error occurs in open code, the entire assembly is processed as comments.

Programmer Response: Within the partition, increase the size of the region that is allocated to assembly, or allocate more of the partition to dictionary space via the BUFSIZE assembler option.

Severity Code: 8

Module Originating Message: CWAX3A

CWA101 GENERATED OP CODE INVALID OR UNDEFINED

Explanation: The operation code created by substitution is not a valid ordinary symbol or is not a valid machine, assembler, or macro instruction, or is not defined by an OPSYN instruction.

Assembler Action: The generated statement is treated as a comment.

Programmer Response: Be sure that substitution results in a valid ordinary symbol that consists of from1 to 8 alphameric characters, the first of which is alphabetic, and that the resulting symbol is a defined operation code.

Severity Code: 8

Module Originating Message: CWAX3A

CWA102 GENERATED OP CODE IS BLANK

Explanation: The op code created by substitution contains no characters, or from 1 to 8 blank characters.

Assembler Action: The generated statement is processed as a comment.

Programmer Response: Be sure that substitution results in a valid ordinary symbol that consists of from 1 to 8 alphameric characters, the first of which is alphabetic.

Severity Code: 8

Module Originating Message: CWAX3A

CWA104 MORE THAN ONE TITLE STATEMENT NAMED

Explanation: This is at least the second TITLE statement that contains something other than a sequence symbol or blanks in the name field.

Assembler Action: The name field is ignored.

Programmer Response: Be sure that the name fields of all but one TITLE statement contain only sequence symbols or blanks.

Severity Code: 4

Module Originating Message: CWAX3A

CWA105 GENERATED FIELD EXCEEDS 255 CHARACTERS

Explanation: As a result of substitution, a character string that is longer than 255 characters has been generated.

Assembler Action: The first 255 characters are used.

Programmer Response: Limit the generation of any character string to 255 characters, minus the number of non-substituted characters. (Limit substitution in the name and operation fields to 8 characters, in the operand field to 255 characters.)

Severity Code: 8

Module Originating Message: CWAX3A

CWA107 CHARACTER STRING USED AS AN ARITHMETIC TERM EXCEEDS 10 CHARACTERS

Explanation: A character string used in a SETA expression or in an arithmetic relation in a SETB expression is longer than 10 characters. Ten is the maximum number of characters permitted in a decimal self-defining term.

Assembler Action: The character string is replaced by an arithmetic value of zero.

Programmer Response: Be sure that all character strings used as described in the explanation are from 1 to 10 decimal digits with a value in a range of 0 to 2,147,483,647. Also be sure that the values of all variables that contribute to the generation of the character string are valid for their type.

Severity Code: 8

Module Originating Message: CWAX3A

CWA108 CHARACTER STRING USED AS AN ARITHMETIC TERM CONTAINS NON-DECIMAL CHARACTERS

Explanation: A character string used in a SETA expression or in an arithmetic relation in a SETB expression contains characters other than 0 through 9.

Assembler Action: The character string is replaced by an arithmetic value of zero.

Programmer Response: Be sure that all character strings used in a SETA expression or as an arithmetic relation in a SETB expression contain from 1 to 10 *decimal* digits with a value in the range of 0 to 2,147,483,647. Also be sure that the values of all variables that contribute to the generation of the character string are valid for their type.

Severity Code: 8

Module Originating Message: CWAX3A

CWA109 CHARACTER STRING USED AS AN ARITHMETIC TERM IS A NULL STRING

Explanation: A character string used in a SETA expression or in an arithmetic relation in a SETB expression is zero characters in length.

Assembler Action: The character string is replaced by an arithmetic value of zero.

Programmer Response: Be sure that all character strings used in an arithmetic context are from 1 to 10 decimal digits with a value in a range of 0 to 2,147,483,647. Also make sure that the values of all variables that contribute to the generation of the character string are valid.

Severity Code: 8

Module Originating Message: CWAX3A

CWA110 ARITHMETIC OVERFLOW IN INTERMEDIATE RESULT OF SETA EXPRESSION

Explanation: During the evaluation of a SETA expression, an intermediate value was produced that was outside the range of -2^{31} to $2^{31}-1$.

Assembler Action: The intermediate result is replaced by an arithmetic value of zero.

Programmer Response: Be sure that the values of all variables that contribute to the intermediate result are valid. No expression should ever attempt a value outside the range of -2^{31} to $2^{31}-1$. Overflow may be avoided if you adjust the sequence of expression evaluation, or if you separate components of the expression and evaluate them individually (perhaps by additional SET statements) before combining them.

Severity Code: 8

Module Originating Message: CWAX3A

CWA111 SUBSCRIPT EXPRESSION HAS A ZERO OR NEGATIVE VALUE

Explanation: A term or a SETA expression used as the subscript on a dimensioned global or local variable symbol results in a zero or negative value.

Assembler Action: Any such reference to the dimensioned variable results in a null (zero) value.

Programmer Response: Be sure that the values of all the variables that contribute to the subscript are valid. Expressions that are used as subscripts must have a value in the range of 1 through the declared dimension of the global or local variable. A zero subscript is allowed only on the system variable &SYSLIST.

Severity Code: 8

Module Originating Message: CWAX3A

CWA112 SUBSCRIPT EXPRESSION EXCEEDS DECLARED DIMENSION

Explanation: A term or a SETA expression used as the subscript on a dimensioned global or local variable results in a value greater than the declared dimension of the variable.

Assembler Action: Any such reference results in a null (zero) value.

Programmer Response: Be sure that all terms and variables that contribute to the subscript have valid values. Be sure that a term or a SETA expression used as a subscript has a value in the range of 1 through the declared dimension of the global or local variable.

Severity Code: 8

Module Originating Message: CWAX3A

CWA113 ILLEGAL REFERENCE MADE TO A PARAMETER THAT IS A SUBLIST

Explanation: A reference has been made in a SETA or SETB expression (that is, in an arithmetic context) to a parameter that is a sublist.

Assembler Action: The reference to the parameter results in an arithmetic value of zero.

Programmer Response: Check to see that the proper parameter is being referenced. Be sure that an appropriate value is assigned to a parameter that is referenced in a SETA or SETB expression. Check for a missing subscript.

Severity Code: 8

Module Originating Message: CWAX3A

CWA114 NEGATIVE DUPLICATION FACTOR IN CHARACTER STRING

Explanation: A term or a SETA expression that is used as the duplication factor in a SETC operand results in a negative value.

Assembler Action: The duplication factor is set to an arithmetic value of zero.

Programmer Response: Be sure that any term or expression used as a duplication factor has a positive value, and that the values of all variables that contribute to the duplication factor are valid.

Severity Code: 8

Module Originating Message: CWAX3A

CWA115 FIRST EXPRESSION IN SUBSTRING NOTATION HAS ZERO OR NEGATIVE VALUE

Explanation: A term or SETA expression that is used to specify the starting character for a substring operation has a zero or negative value.

Assembler Action: The assembler assigns the value of null to the substring.

Programmer Response: A term, a SETA expression, or a combination of variables used to produce the first expression in a substring notation must result in a positive, nonzero value, not exceeding the length of the character string.

Severity Code: 8

Module Originating Message: CWAX3A

CWA116 SECOND EXPRESSION IN SUBSTRING NOTATION HAS NEGATIVE VALUE

Explanation: A term or SETA expression that is used to specify the number of characters affected by a substring operation has a negative value.

Assembler Action: The value of the second expression of the substring notation is set to 0, that is, the assembler assigns a value of null to the substring.

Programmer Response: A term, a SETA expression, or a combination of variables used to produce the second expression in a substring notation must result in a non-negative value.

Severity Code: 4

Module Originating Message: CWAX3A

CWA117 FIRST EXPRESSION IN SUBSTRING NOTATION EXCEEDS THE LENGTH OF THE STRING

Explanation: A term or SETA expression that specifies the starting character for a substring operation specifies a character beyond the end of the string.

Assembler Action: The assembler assigns the value of null to the substring.

Programmer Response: Make sure the term, SETA expression, or combination of variables used to produce the first expression in a substring notation results in a value in the range of 1 through the length of the character string.

Severity Code: 8

Module Originating Message: CWAX3A

CWA118 ACTR LIMIT HAS BEEN EXCEEDED

Explanation: The number of AIF and AGO branches within the text segment exceeds the value specified in the ACTR instruction or the conditional assembly loop counter default value.

Assembler Action: If a macro is being expanded, the expansion is terminated. If open code is processed, all remaining statements are processed as comments.

Programmer Response: Correct the conditional assembly loop that caused the ACTR limit to be exceeded, or set the ACTR value to a higher number.

Severity Code: 8

Module Originating Message: CWAX3A

CWA119 ILLEGAL TYPE ATTRIBUTE REFERENCE

Explanation: A type attribute reference is made to a symbol defined by an EQU instruction with an invalid third operand.

Assembler Action: The type attribute value is set to U.

Programmer Response: Correct the third operand on the EQU instruction. It must be a self-defining term in the range 0-255.

Severity Code: 4

Module Originating Message: CWAX3A

CWA120 ILLEGAL LENGTH ATTRIBUTE REFERENCE

Explanation:

- A length attribute reference specifies a SETx symbol; or
- A length attribute reference specifies a symbolic parameter (or a &SYSLIST representation) that does not represent an ordinary symbol; or
- The ordinary symbol referenced by a length or integer attribute reference is defined by an EQU instruction, and the value of the second operand of that instruction is not in the range 0-65535; or
- The ordinary symbol referenced by a length or integer attribute reference is defined in a DC or DS instruction, and the instruction contains a length modifier that is not a self-defining term.

Assembler Action: The length attribute is set to 1.

Programmer Response: Review the use of the length attribute and recode.

Severity Code: 4

Modules Originating Message: CWAX3A, CWAX5U

CWA121 ILLEGAL COUNT (K') ATTRIBUTE REFERENCE

Explanation:

- The count attribute references a non-macro instruction operand; or
- The count attribute reference is not used in an arithmetic expression.

Assembler Action: The count attribute is set to zero.

Programmer Response: Review the use of the count attribute and recode.

Severity Code: 4

Module Originating Message: CWAX3A

CWA122 ILLEGAL NUMBER (N') ATTRIBUTE REFERENCE

Explanation:

- The number attribute references a non-macro instruction operand; or
- The number attribute is not used in an arithmetic expression.

Assembler Action: The number attribute is set to zero.

Programmer Response: Review the use of the number attribute and recode.

Severity Code: 4

Module Originating Message: CWAX3A

CWA123 ILLEGAL SCALE ATTRIBUTE REFERENCE

Explanation:

- A scaling attribute reference specifies a SETx symbol; or
- A scaling attribute reference specifies a symbolic parameter (or a &SYSLIST representation) that does not represent an ordinary symbol; or
- A scaling attribute reference is made to an ordinary symbol whose type attribute is not H, F, G, E, D, L, K, P, or Z; or
- The ordinary symbol referenced by a scaling or integer attribute reference is defined in a DC or DS instruction containing a scaling modifier that is not a self-defining term.

Assembler Action: The scale attribute is set to 0.

Programmer Response: Review the use of the scale attribute and recode.

Severity Code: 4

Module Originating Message: CWAX3A

CWA124 ILLEGAL INTEGER ATTRIBUTE REFERENCE

Explanation:

- An integer attribute reference specifies a SETx symbol; or
- An integer attribute reference specifies a symbolic parameter (or a &SYSLIST representation) that does not represent an ordinary symbol; or
- An integer attribute reference is made to an ordinary symbol whose type attribute is not H, F, G, or K.

Assembler Action: The integer attribute is set to 0.

Programmer Response: Review the use of the integer attribute and recode.

Severity Code: 4

Module Originating Message: CWAX3A

CWA125 INVALID NAME—ILLEGAL EMBEDDED CHARACTER OR NON-ALPHABETIC FIRST CHARACTER

Explanation:

- The symbol generated in the name field does not begin with an alphabetic character or it contains a special character or an embedded blank after substitution; or
- for the TITLE instruction: the name field contains a special character.

Assembler Action: The name field is ignored.

Programmer Response: Be sure that the symbol generated in the name field conforms to the rules for forming valid ordinary symbols, or is a valid TITLE name field entry. Also check to make sure that the values of all variables that contribute to the generation of the symbol in the name field are valid.

Severity Code: 8

Module Originating Message: CWAX3A

CWA126 MORE THAN 5 ERRORS IN THIS STATEMENT, PROCESSING OF THE STATEMENT IS TERMINATED

Explanation: Six or more errors were detected in processing this statement. The maximum number of error messages issued by the processor to each statement is five.

Assembler Action: The sixth error causes this message to be issued, and messages are not issued for any further errors in this statement.

Programmer Response: Correct the indicated errors and check carefully for errors beyond the point indicated by the fifth error message. Assemble again. Any additional errors will be located in the next assembly.

Severity Code: 8

Module Originating Message: CWAX3A

CWA127 VALUE OF CHARACTER STRING USED IN ARITHMETIC CONTEXT EXCEEDS 2, 147, 483, 647

Explanation: A character string used in a SETA expression or in an arithmetic relation in a SETB expression exceeds a value of 2, 147, 483, 647, which is the maximum value allowed for a decimal self-defining term.

Assembler Action: The character string is replaced by an arithmetic value of zero.

Programmer Response: Be sure that all character strings used in an arithmetic context are from 1 to 10 decimal digits and have a value in the range of 0 to 2, 147, 483, 647. Be sure that the values of all variables that contribute to the generation of the character string are valid.

Severity Code: 8

Module Originating Message: CWAX3A

CWA128 GENERATED OP CODE EXCEEDS 8 CHARACTERS

Explanation: The syntax for mnemonic operation codes must follow the same rules as ordinary symbols; that is, they must be from 1 to 8 alphanumeric characters long and the first character must be alphabetic.

Assembler Action: The statement that contains the illegal op code is processed as a comment. Only the first 8 characters of the generated op code appear in the printed statement.

Programmer Response: Be sure that the values of all variables that contribute to the generation of the op code are valid, and be sure that no attempt is made to generate an op code of more than 8 characters.

Severity Code: 8

Module Originating Message: CWAX3A

CWA129 GENERATED SYMBOL IN NAME FIELD EXCEEDS 8 CHARACTERS

Explanation: A generated symbol that appears in the name field exceeds 8 characters. It should be from 1 to 8 alphameric characters in length, and the first character should be alphabetic.

Assembler Action: The name field is ignored. Only the first eight characters of the generated symbol appear in the printed statement.

Programmer Response: Be sure that the values of all variables that contribute to the generation of the symbol in the name field are valid. Be sure that no attempt is made to generate a symbol of more than 8 characters.

Severity Code: 8

Module Originating Message: CWAX3A

CWA130 FIRST SUBSCRIPT OF &SYSLIST REFERENCE IS NEGATIVE

Explanation: A term or an arithmetic (SETA) expression that is used as the first subscript of a &SYSLIST reference has resulted in a negative value.

Assembler Action: The parameter reference is treated as a reference to an omitted operand.

Programmer Response: Be sure that the values of all variables that contribute to the generation of the first subscript are valid.

Severity Code: 8

Module Originating Message: CWAX3A

CWA131 INCONSISTENT GLOBAL VARIABLE DECLARATION, SETx INSTRUCTION IGNORED

Explanation: Global variable declaration is inconsistent with a previous definition of the variable in another macro definition or in open code.

Assembler Action: The value of the global variable remains the same and the SETx instruction is ignored.

Programmer Response: Correct all inconsistencies between global variable declarations regarding dimension and type.

Severity Code: 8

Module Originating Message: CWAX3N

CWA132 REFERENCE TO INCONSISTENTLY DECLARED GLOBAL VARIABLE RESULTS IN ZERO VALUE

Explanation: An attempt to obtain a value from a global variable has been ignored because the declaration of the global variable was inconsistent with a previous declaration of the same variable in another macro definition or in open code. Either the dimension or the type does not agree.

Assembler Action: The reference to the global variable is replaced by a null or zero value.

Programmer Response: Correct all inconsistencies among declarations of the same global variable.

Severity Code: 8

Module Originating Message: CWAX3N

CWA133 NO WORK SPACE FOR OPEN CODE SKELETON DICTIONARY

Explanation: The allotted dictionary work space is insufficient to build the skeleton dictionary for open code. Since the generation process requires the open code dictionary, generation is not attempted.

Assembler Action: The entire assembly is processed as comments.

Programmer Response: Within the partition, increase the size of the region that is allocated to assembly, or allocate more of the partition to dictionary space via the BUFSIZE assembler option.

Severity Code: 12

Module Originating Message: CWAX3N

CWA134 BRANCH ADDRESS LESS THAN 0

Explanation:

- The branch address of an RT machine instruction is outside the current CSECT; or
- The branch address of an RA machine instruction is negative.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the RT- or RA-type machine instructions and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA135 IMMEDIATE FIELD NEGATIVE NEAR OPERAND COLUMN nn

Explanation: The immediate field "I" of the RI machine instruction is negative.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the RI-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA136 ABSOLUTE BRANCH ADDRESS NEAR OPERAND COLUMN nn

Explanation: The "T" field of an RT machine instruction is not a relocatable expression.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the RT-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA137 BRANCH ADDRESS OUTSIDE CURRENT CSECT

Explanation: The branch address of an RT machine instruction is outside of the current CSECT.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the RT-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA138 M-FIELD FOR BBE GREATER THAN 15 NEAR OPERAND COLUMN nn

Explanation: The mask field "M" in the BBE machine instruction is greater than 15 (the field is only 4 bits long).

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the "M" field in the BBE extended machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA149 DISPLACEMENT GREATER THAN 127 NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction (IC, STC) the displacement was greater than 127 bytes.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the IC or STC machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA150 DISPLACEMENT GREATER THAN 126 NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction (LH, STH, STHZ) the displacement was greater than 126 bytes.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the LH, STH or STHZ machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA151 DISPLACEMENT GREATER THAN 124 NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction (L, ST, BND, STZ) the displacement was greater than 124 bytes.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the L, ST, BND or STZ machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA152 DISPLACEMENT IS NEGATIVE NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction the displacement was found to be negative.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the failing RS-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA153 DISPLACEMENT IS NOT A MULTIPLE OF 2 NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction (LH, STH, STHZ) the displacement did not reference a halfword boundary.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the failing RS-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA154 DISPLACEMENT IS NOT A MULTIPLE OF 4 NEAR OPERAND COLUMN nn

Explanation: In an RS machine instruction (L, ST, BND, STZ) the displacement did not reference a fullword boundary.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the failing RS-type machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA155 BYTE DESIGNATOR IS NOT AN ABSOLUTE VALUE NEAR OPERAND COLUMN nn

Explanation: In a machine instruction which requires at least one of its operands to have the format R(N), the byte designator "N" was not an absolute value.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the failing machine instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA157 DC OPERAND VALUE TOO LONG

Explanation: The specified value of an operand in a DC instruction is too long. The maximum length of a DC operand is 16,777,215 bytes.

Assembler Action: The specified value is ignored.

Programmer Response: Make the constant shorter, or break it up into two constants.

Severity Code: 8

Module Originating Message: CWAX5D

CWA158 NAME OF STATEMENT IN DSECT USED IN RELOCATABLE ADDRESS CONSTANT

Explanation: A non-paired relocatable term used in an A-type, R-type, or Y-type address constant is defined in a dummy section.

Assembler Action: The constant is ignored.

Programmer Response:

- Make sure the relocatable term is not defined in a dummy section; or
- Make sure the term defined in the dummy section is paired with another term (with the opposite sign) from the same dummy section.

Severity Code: 8

Module Originating Message: CWAX5A

CWA161 INVALID EQUR SYMBOLIC REGISTER EXPRESSION

Explanation: The EQUR assembler instruction has:

- An invalid format; or
- An invalid register "R" value; or
- An invalid byte "N" value.

Assembler Action: The value of the EQUR instruction is set to zero.

Programmer Response: Correct the format and/or insert correct values for register and byte and recode.

Severity Code: 8

Module Originating Message: CWAX2A

CWA163 MISSING OR INVALID SYMBOL IN NAME FIELD

Explanation: One of two errors has occurred:

- A symbol is missing in the name field where one is required.
- The symbol in the name field in invalid.

Assembler Action: The statement is processed as a comment.

Programmer Response: Supply a valid name.

Severity Code: 4

Module Originating Message: CWAX5A

CWA164 INVALID OR ILLEGAL START STATEMENT

Explanation: The START statement did not start the first control section in the assembly, or the operand on the START statement was not an absolute value.

Assembler Action: The START statement is treated as a CSECT statement.

Programmer Response: Be sure that the START statement has an absolute operand and that it begins the first control section in the assembly.

Severity Code: 4

Module Originating Message: CWAX5A

CWA165 NULL PUNCH OPERAND OR PUNCH OPERAND EXCEEDS 80 CHARACTERS

Explanation: The operand of a PUNCH instruction either specifies only a null string surrounded by quotes, or is more than 80 characters long.

Assembler Action: The PUNCH statement is processed as a comment.

Programmer Response: Be sure that the operand of a PUNCH statement consists of from 1 to 80 characters surrounded by quotes.

Severity Code: 4

Module Originating Message: CWAX5A

CWA167 SYMBOL FILE OUT OF STEP

Explanation: The symbol file, which is an internal data file, has got out of step with the rest of the assembly process because of error recovery on a user error.

Assembler Action: A soft recovery is attempted by continuing the assembly. Assembly results subsequent to the point of error may not be valid.

Programmer Response: This message will always be accompanied by user errors. Correct them and reassemble the program.
If the problem recurs, do the following before calling IBM:

- Have your source program, macro definitions, and associated listings available.
- If a COPY statement was use, execute the IEBPTPCH utility to obtain a copy of the partitioned data set member specified in the COPY statement.
- Make sure that MSGLEVEL=(1,1) was specified in the JOB statement.

Severity Code: 16

Module Originating Message: CWAX5C

CWA168 AN ARITHMETIC EXPRESSION NOT USED IN CONDITIONAL ASSEMBLY CONTAINS MORE THAN 20 TERMS

Explanation: An arithmetic expression used in a macro definition or in open code, but not in a conditional assembly statement, contains more than 19 unary and binary operators and 6 levels of parentheses. The maximum number of terms this combination allows is 20.

Assembler Action: The value of the expression is set to 0.

Programmer Response: Be sure that this arithmetic expression does not contain more than 19 operators (unary and binary) and 6 levels of parentheses. If greater complexity is necessary, use EQU statements to evaluate intermediate results.

Severity Code: 8

Module Originating Message: CWAX5V

CWA169 INVALID SELF-DEFINING TERM NEAR OPERAND COLUMN nn

Explanation: A self-defining term was invalidly specified.

Assembler Action: The value of the term is set to zero.

Programmer Response: Check the syntax and correct the error.

Severity Code: 8

Module Originating Message: CWAX5V

CWA170 TWO ADJACENT BINARY OPERATORS, OR BINARY OPERATOR EXPECTED BUT NOT FOUND NEAR OPERAND COLUMN nn

Explanation: One of two errors has occurred.

1. Two binary operators appear consecutively near the column specified in the message text. This applies only to "*" (multiply) and "/" (divide).
2. A binary operator was expected near the column specified in the message text, but none was found. A single binary operator must occur between all terms of an expression.

Assembler Action: The expression that contains the absent or illegal operator is set to zero.

Programmer Response:

1. Eliminate one of the binary operators.
2. Provide a binary operator.

Severity Code: 8

Module Originating Message: CWAX5V

CWA171 TITLE STATEMENT OPERAND EXCEEDS 100 CHARACTERS

Explantion: The operand of a TITLE instruction contains more than 100 characters.

Assembler Action: The character string in the operand is truncated to 100 characters.

Programmer Response: Be sure that the length of the character string in the operand of a TITLE statement does not exceed 100 characters.

Severity Code: 4

Module Originating Message: CWAX5A

CWA172 VALUE OF ORG OPERAND IS LESS THAN THE CONTROL SECTION STARTING ADDRESS

Explanation: The operand of an ORG statement results in a value less than the starting address of the control section.

Assembler Action: The ORG statement is processed as a comment and has no effect on the value of the location counter.

Programmer Response: Be sure that the operand of the ORG statement is a positive relocatable expression, greater than the starting address of the control section, or blank.

Severity Code: 8

Module Originating Message: CWAX5A

CWA173 ONE OR MORE SYMBOLS IN AN ORG OPERAND DO NOT BELONG TO THE CURRENT CSECT, DSECT, OR COM

Explanation: One or more of the symbols used in the operand of an ORG statement are not defined in the current control section (dummy, common or ordinary).

Assembler Action: The ORG statement is processed as a comment and the value of the location counter remains unchanged.

Programmer Response: Be sure that all symbols used in the operand field of an ORG statement belong to (are defined by appearing in the name field of a statement within) the current control section.

Severity Code: 8

Module Originating Message: CWAX5A

CWA174 ORG OPERAND IS ABSOLUTE. MUST BE RELOCATABLE

Explanation: An absolute term or expression used in the operand of an ORG statement must be a relocatable term, a relocatable expression, or a blank.

Assembler Action: The ORG instruction is processed as a comment and the value of the location counter remains unchanged.

Programmer Response: Be sure that the operand of an ORG statement is a relocatable term, a relocatable expression, or a blank. An ORG to an absolute address is not possible because the assembler assumes that all location references are relocatable. A common error is an ORG to 0. Since the start of the program is not absolute machine location 0 but relocatable 0, replace the 0 with a symbol or expression that makes reference to the labeled program start.

Severity Code: 8

Module Originating Message: CWAX5A

CWA175 OPERAND SHOULD BEGIN WITH A QUOTE.

Explanation: A quote was expected to begin a character string in the operand field, but was not found.

Assembler Action: The invalid character string is ignored.

Programmer Response: Supply the missing leading quote in the character string of the operand.

Severity Code: 8

Module Originating Message: CWAX5A

CWA176 UNPAIRED AMPERSAND NEAR OPERAND COLUMN nn

Explanation: A single ampersand followed by a blank was found in a quoted character string. If an ampersand is desired as a character in a quoted character string, two ampersands must be coded. Ampersands must be either paired or part of a valid variable symbol.

Assembler Action: The character string that contains the illegal ampersand is ignored.

Programmer Response: Determine whether the ampersand is desired as a character in a quoted character string or whether the ampersand is intended as the beginning of a valid variable symbol, and correct the error.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5D

CWA177 MISSING OPERAND

Explanation: This statement requires an operand, but none is found.

Assembler Action: The statement which lacks the operand is processed as a comment.

Programmer Reponse: Supply a valid operand.

Severity Code: 12

Module Originating Message: CWAX5A

CWA178 SYNTAX ERROR NEAR OPERAND COLUMN nn

Explanation: A syntax error has occurred in the operand of this statement.

Assembler Action: The statement which contains the invalid operand is processed as a comment.

Programmer Response: Correct the syntax of the operand. There are a large number of syntactic errors that can produce this diagnostic. All of them require careful checking of the syntax of the specific type of statement being processed. The error is logged at the point where the syntax becomes ambiguous or unrecognizable, not necessarily at the point where the actual error occurs.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5D, CWAX5M

CWA179 OPERAND SUBFIELD NEAR OPERAND COLUMN nn MUST BE ABSOLUTE

Explanation: All terms and expressions used in the operand field of this statement must result in an absolute value.

Assembler Action: The operand is processed as a comment.

Programmer Response: Be sure that each term or expression used in the operand field of this statement has an absolute value. No relocatable expressions are allowed.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5D, CWAX5M

CWA180 OPERAND 2 OF CNOP MUST BE EITHER 4 OR 8

Explanation: The second operand of a CNOP statement must be either 4 or 8.

Assembler Action: The CNOP statement is processed as a comment and no alignment is performed.

Programmer Response: Be sure that the second operand of a CNOP statement is either a 4 or an 8.

Severity Code: 12

Module Originating Message: CWAX5A

CWA181 OPERAND 1 OF CNOP MUST BE 0, 2, 4, OR 6

Explanation: The first operand of a CNOP statement must be 0, 2, 4, or 6.

Assembler Action: The CNOP statement is ignored and no alignment is performed.

Programmer Response: Be sure that the first operand of a CNOP statement is a 0, 2, 4, or 6.

Severity Code: 12

Module Originating Message: CWAX5A

CWA 182 OPERAND 1 OF CNOP IS NOT LESS THAN OPERAND 2

Explanation: The value of the first operand of a CNOP statement must be less than the value of the second operand.

Assembler Action: The CNOP statement is processed as a comment and no alignment is performed.

Programmer Response: Check the validity of each operand of the CNOP statement to be sure that the value of the second operand is greater than the value of the first operand.

Severity Code: 12

Module Originating Message: CWAX5A

CWA183 MNOTE OPERAND EXCEEDS 255

Explanation: The value of an operand used as an MNOTE severity exceeds 255.

Assembler Action: The MNOTE is processed as a comment.

Programmer Response: Check the validity of the operand.

Severity Code: 12

Module Originating Message: CWAX5A

CWA 184 INVALID COUNT ON CW NEAR OPERAND COLUMN nn, 1023 IS MAXIMUM VALUE

Explanation: The value of the third operand of a define control word has exceeded X'3FF' (1023).

Assembler Action: Space is allocated for the CW, but the value of the flagged operand is set to 0.

Programmer Response: Check the validity of the third operand of the define control word.

Severity Code: 12

Module Originating Message: CWAX5A

CWA185 BLANK EXPECTED AS A DELIMITER NEAR OPERAND COLUMN nn

Explanation: A blank was expected as a delimiter but none was found. Subsequent characters have no syntactic meaning, and the statement is ambiguous.

Assembler Action: The statement that contains the invalid delimiter is processed as a comment.

Programmer Response: Supply a blank delimiter.

Severity Code: 8

Module Originating Message: CWAX5A, CWAX5M

CWA186 INVALID SYMBOL NEAR OPERAND COLUMN nn OF ENTRY, EXTRN, OR WXTRN

Explanation: An improperly constructed symbol was found in the operand field of an ENTRY, EXTRN, or WXTRN statement.

Assembler Action: The statement that contains the invalid symbol is processed as a comment.

Programmer Response: Be sure that the symbol in the operand field of EXTRN, WXTRN, or ENTRY statements contain from 1 to 8 alphanumeric characters, the first of which is alphabetic.

Severity Code: 8

Module Originating Message: CWAX5A

CWA187 SYMBOL LONGER THAN 8 CHARACTERS NEAR OPERAND COLUMN nn

Explanation: A symbol that is more than 8 characters in length has appeared in the operand field of this statement.

Assembler Action: The invalid symbol in the operand field is replaced by a zero.

Programmer Response: Be sure that symbols do not exceed 8 characters in length. A missing or misplaced delimiter or operator may cause a symbol to appear longer than intended.

Severity Code: 8

Module Originating Message: CWAX5A, CWAX5D, CWAX5V

CWA188 xxxxxxxx IS AN UNDEFINED SYMBOL

Explanation: The symbol that appears in the message text has not appeared in the name field of another statement, or as an operand of an EXTRN or WXTRN statement.

Assembler Action: Reference to the undefined symbol results in a zero value.

Programmer Response: Define the symbol in the program.

Severity Code: 8

Module Originating Message: CWAX5V

CWA189 INVALID ENTRY OPERAND, LINKAGE CANNOT BE PERFORMED

Explanation: The symbol in the operand field of an ENTRY statement is invalid because it is either undefined or improperly defined.

Assembler Action: The invalid symbol in the operand field is processed as a comment, and no linkage is provided if another program references it.

Programmer Response: Define the symbol at an appropriate place in this program, or correct it. A valid symbol consists of from 1 to 8 alphameric characters, the first of which is blank.

Severity Code: 8

Module Originating Message: CWAX5A

CWA190 OPERAND OF PUSH STATEMENT IS NOT USING OR PRINT NEAR OPERAND COLUMN nn

Explanation: The only symbols allowed in the operand field of a PUSH or POP statement are PRINT and USING, in any order, separated by commas.

Assembler Action: The PUSH instruction is processed as a comment.

Programmer Response: Be sure the operand of the PUSH statement is either PRINT or USING or both.

Severity Code: 4

Module Originating Message: CWAX5A

CWA191 PUSH LEVELS EXCEEDS 4 NEAR OPERAND COLUMN nn

Explanation: More than 4 levels of PUSH and POP statements were attempted for either PRINT or USING.

Assembler Action: The PUSH instruction is processed as a comment.

Programmer Response: Rework the program logic to require no more than 4 levels of PUSH and POP for USING and 4 for PRINT.

Severity Code: 8

Module Originating Message: CWAX5A

CWA192 OPERAND OF POP STATEMENT IS NOT USING OR PRINT NEAR OPERAND COLUMN nn

Explanation: The only symbols allowed in the operand of a PUSH or POP statement are USING and PRINT, in any order, separated by commas.

Assembler Action: The POP instruction is processed as a comment.

Programmer Response: Be sure the operand of the POP statement is either PRINT or USING or both.

Severity Code: 4

Module Originating Message: CWAX5A

CWA193 POP REQUEST NOT BALANCED BY PREVIOUS PUSH

Explanation: No PUSH request was issued prior to this POP request, or more POP statements have been issued than PUSH statements. A POP statement restores the USING or PRINT status saved by the most recent PUSH statement, on a one for one basis.

Assembler Action: The POP instruction is processed as a comment.

Programmer Response: Check for errors in balancing PUSH and POP statements, or rework the program logic to request balanced PUSH and POP statements. Repetition of a given operand (i.e., USING or PRINT) on a single PUSH or POP statement is treated as multiple statements, and could cause unbalanced PUSH and POP statements.

Severity Code: 8

Module Originating Message: CWAX5A

CWA194 INVALID OPTION IN PRINT STATEMENT NEAR OPERAND COLUMN nn

Explanation: An option appears in the operand field of a PRINT statement that is not one of the following: ON, OFF, GEN, NOGEN, DATA, and NODATA.

Assembler Action: The invalid operand is ignored.

Programmer Response: Be sure that only the options listed in the explanation above appear in the operand field of a PRINT statement.

Severity Code: 4

Module Originating Message: CWAX5A

CWA195 INVALID USING OR DROP STATEMENT NEAR OPERAND COLUMN nn

Explanation: One of three errors has occurred:

1. register 0 is specified for other than the second operand of a USING statement; or
2. a register number outside the range of 0 to 7 has been used; or
3. a DROP statement has been issued for a register that was never assigned for use by a USING statement.

Assembler Action: The invalid register specification is set to zero.

Programmer Response: The second and following operands of a USING or DROP instruction must be decimal terms 0 to 7. Register 0 may only be specified as the second operand of a USING statement.

Severity Code: 12

Module Originating Message: CWAX5A

CWA196 xxxxxxxx HAS BEEN PREVIOUSLY DEFINED

Explanation: The specified symbol has previously appeared in the name field of a statement or in the operand field of an EXTRN or WXTRN instruction.

Assembler Action: All references to the symbol are interpreted as references to the first definition of the symbol.

Programmer Response: A given symbol must be defined only once. Determine which occurrence of the symbol you want to use, and change all others.

Severity Code: 8

Module Originating Message: CWAX5A, CWAX5C

CWA197 ***MNOTE***

Explanation: An MNOTE statement has been encountered during the generation of a macro or open code. The text of the MNOTE message appears in-line in the listing at the point where it is encountered. (Refer to *OS/VS–DOS/VS–VM/370 Assembler Language* (GC33-4010) for a description of the MNOTE instruction.)

Assembler Action: None.

Programmer Response: Investigate the reason for the MNOTE. Errors flagged by MNOTE will often cause unsuccessful execution of the program, depending upon the severity code.

Severity Code: An MNOTE is assigned a severity code of 0 to 255 by the writer of the MNOTE statement.

Module Originating Message: CWAX5A

CWA198 INVALID TYPE DECLARED ON DC/DS/DXD CONSTANT NEAR OPERAND COLUMN nn

Explanation: Operand subfield 2 is not a valid type for a DC, DS, or DXD statement. Valid types are the following: A, B, C, F, H, Q, R, V, X, and Y.

Assembler Action: The statement that contains the invalid type declaration is processed as a comment.

Programmer Response: Supply a valid type in operand subfield 2.

Severity Code: 8

Module Originating Message: CWAX5D

CWA199 INVALID LENGTH MODIFIER NEAR OPERAND COLUMN nn

Explanation: The length modifier in operand subfield 3 of this statement is invalid. The length attribute of a symbol is not allowed as a term in the length modifier expression for the first operand of the DC, DS, or DXD statement in which the symbol is defined. For example, SYM DC CL(L'SYM) 'AA' is invalid.

Assembler Action: The statement that contains the invalid length modifier is processed as a comment.

Programmer Response: Supply a valid length modifier, or eliminate the explicit length modifier.

Severity Code: 8

Module Originating Message: CWAX5D

CWA200 INVALID SCALE MODIFIER NEAR OPERAND COLUMN nn

Explanation: The scale modifier in operand subfield 3 of a DC, DS, or DXD statement is invalid. The scale modifier should be either a decimal value or an absolute expression enclosed in parentheses.

Assembler Action: The statement that contains the invalid scale modifier is processed as a comment.

Programmer Response: Supply a valid scale modifier for the type of constant used.

Severity Code: 8

Module Originating Message: CWAX5D

CWA201 ILLEGAL OR INVALID EXPONENT MODIFIER IN DC/DS/DXD CONSTANT NEAR OPERAND COLUMN nn

Explanation: An exponent modifier used in a DC, DS, or DXD constant is not a decimal self-defining term, an absolute expression enclosed in parentheses, or produces a value outside the range allowed for that constant type.

Assembler Action: The invalid or illegal operand is ignored.

Programmer Response: Be sure that the exponent modifier used conforms to the rules for exponent modifiers for each type of DC, DS, or DXD constant.

Severity Code: 8

Module Originating Message: CWAX5D

CWA202 ILLEGAL USE OF SYMBOLIC REGISTER IN EXPRESSION NEAR OPERAND COLUMN nn

Explanation: An expression contained an invalid value which was a Symbolic Register notation R(n).

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the expression causing the error and recode.

Severity Code: 12

Modules Originating Message: CWAX5A, CWAX5D, CWAX5M, CWAX5V

CWA203 F-, H-, R-, OR Y-TYPE CONSTANT TRUNCATED, HIGH ORDER DIGITS LOST NEAR OPERAND COLUMN nn

Explanation: The high order digits of an F-, H-, R-, or Y-type constant were lost because the designated field was too small to contain the whole constant.

Assembler Action: Processing continues using the truncated constant.

Programmer Response: Modify the explicit or implicit length of the constant, so that the value may be contained within the area designated for it.

Severity Code: 4

Module Originating Message: CWAX5D

CWA204 RELOCATABLE EXPRESSION IN A-, R-, OR Y-TYPE ADDRESS CONSTANT WITH BIT LENGTH SPECIFICATION NOT ALLOWED

Explanation: A relocatable expression in used to specify a constant for which bit length specification is used. This is not allowed.

Assembler Action: The value of the operand is set to 0 and no entry for this constant is made in the relocation dictionary.

Programmer Response: Convert the operand to an absolute expression, or use a length of 3 or 4 bytes for A-type or 2 bytes for Y-type and R-type constants.

Severity Code: 8

Module Originating Message: CWAX5D

CWA206 DUPLICATION FACTOR ERROR

Explanation: The duplication factor in a DC, DS, or DXD statement is negative.

Assembler Action: No storage is reserved for the operand, but alignment is performed as required by the type of constant used.

Programmer Response: Supply a non-negative duplication factor.

Severity Code: 8

Module Originating Message: CWAX5D

CWA207 OPERAND OF Q-TYPE CONSTANT DOES NOT NAME A DSECT OR DXD

Explanation: The symbol in the operand field of a Q-type constant must have been previously defined as the name of a DSECT or DXD section.

Assembler Action: The value of the constant is set to 0.

Programmer Response: Define the symbol as the name of a DSECT or DXD section. The symbol must be defined before being used in the constant.

Severity Code: 8

Module Originating Message: CWAX5D

CWA209 ADDRESSABILITY ERROR--BASE AND DISPLACEMENT CANNOT BE RESOLVED AND ARE SET TO 0

Explanation: The assembler cannot resolve the address of this statement or the address referenced by this statement because no USING registers are available.

Assembler Action: The base and displacement fields of the machine instruction are set to 0.

Programmer Response: Make sure you have correctly set up base registers with the USING instruction. Be sure the referenced address can be specified by the value in a USING register plus a displacement in the range of 0 through 127.

Severity Code: 8

Module Originating Message: CWAX5M

CWA210 TOO FEW OPERANDS

Explanation: More operands are required for this statement, but they were not found.

Assembler Action: The value of any missing operand is set to 0.

Programmer Response: Supply the necessary operands.

Severity Code: 12

Modules Originating Message: CWAX5A, CWAX5M

CWA211 TOO MANY OPERANDS

Explanation:

- More than 255 operands on a DC, DS, or DXD instruction; or
- Too many operands on a machine instruction.

Assembler Action: The extra operands are ignored.

Programmer Response: Delete the extra operands. Refer to *Principles of Operation* for details on operands required for individual machine instructions.

Severity Code: 12

Module Originating Message: CWAX5A

CWA213 COMPLEXLY RELOCATABLE EXPRESSION NEAR OPERAND COLUMN nn

Explanation: The indicated operand contains a complexly relocatable expression. The expression should be absolute or simply relocatable.

Assembler Action: The value of the complexly relocatable expression is set to 0.

Programmer Response: Be sure that only absolute and simply relocatable expressions are used in the operand field of this statement.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5M

CWA214 COMMA EXPECTED AS DELIMITER NEAR OPERAND COLUMN nn

Explanation: A machine instruction:

- did not have a comma separating two sub-operands; or
- had one operand when two sub-operands were expected.

Assembler Action: The value of the machine instruction is set to zero.

Programmer Response: Review the format of the failing machine instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5M

CWA215 ILLEGAL DELIMITER, RIGHT PARENTHESIS EXPECTED NEAR OPERAND COLUMN nn

Explanation: A right parenthesis was expected as a delimiter, but none was found.

Assembler Action: The value of the operand that is lacking a right parenthesis is set to 0.

Programmer Response: Supply a right parenthesis.

Severity Code: 8

Module Originating Message: CWAX5M

CWA216 ILLEGAL OPERAND FORMAT NEAR OPERAND COLUMN nn

Explanation: The operand of this statement is illegally constructed.

Assembler Action: The value of the operand is set to 0.

Programmer Response: Supply a valid operand.

Severity Code: 12

Modules Originating Message: CWAX5A, CWAX5M

CWA217 RELOCATABILITY ERROR NEAR OPERAND COLUMN nn

Explanation: One of the following fields contains a relocatable value. All values in these fields must be absolute.

- Immediate field in an RI instruction
- Mask field
- Register specification
- Length modifier

Assembler Action: If any of the above fields contains a relocatable value, the value of the field is set to 0.

Programmer Response: Be sure that the field contains an absolute value.

Severity Code: 12

Modules Originating Message: CWAX5A, CWAX5M, CWAX5V

CWA218 INVALID REGISTER SPECIFICATION–ODD NUMBERED REGISTER REQUIRED

Explanation: An even-numbered register was specified in a context that requires an odd-numbered register.

Assembler Action: The invalid operand is set to 0.

Programmer Response: Specify an available odd-numbered register.

Severity Code: 12

Module Originating Message: CWAX5A

CWA219 REGISTER OR IMMEDIATE FIELD OVERFLOW NEAR OPERAND COLUMN nn

Explanation:

- The value of the immediate field used in an RI instruction is greater than 255; or
- A register number was specified that was greater than 7.

Assembler Action: The value of the field where the overflow occurred is set to 0.

Programmer Response: Be sure the value of an immediate field does not exceed 255 and that no register number greater than 15 is specified.

Severity Code: 8

Module Originating Message: CWAX5M

CWA220 ALIGNMENT ERROR NEAR OPERAND COLUMN nn

Explanation: The operand of this instruction refers to a main storage location that is not on the boundary required by the instruction.

Assembler Action: The faulty alignment is unchanged.

Programmer Response: Align the main storage location referenced in the operand field.

Severity Code: 4

Module Originating Message: CWAX5A

CWA221 REGISTER NUMBER 0 NOT ALLOWED NEAR OPERAND COLUMN nn

Explanation: Specification of register 0 is not allowed at this point in the operand.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the invalid instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5M

CWA222 REGISTER NUMBER LESS THAN 0 NEAR OPERAND COLUMN nn

Explanation: A register was given a negative value.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the invalid machine instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5M

CWA223 EXTERNAL REGISTER GREATER THAN 127

Explanation: An RE-type machine instruction has a register specified that is not within the range of 0-127.

Assembler Action: The machine instruction is set to zero.

Programmer Response: Review the format of the RE-type machine instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5M

CWA224 LENGTH ERROR NEAR OPERAND COLUMN nn

Explanation:

- The length modifier of a constant is invalid for that type of constant; or
- A constant of type C, X, or B is too long; or
- A relocatable address constant has an invalid length.

Assembler Action: The operand in error and any following operands of the DC, DS, or DXD statement are processed as comments. An address constant with an invalid length is truncated.

Programmer Response: Supply a valid length modifier or decrease the length of the operand.

Severity Code: 8

Module Originating Message: CWAX5D

CWA226 BASE REGISTER OF MACHINE INSTRUCTION NOT ABSOLUTE NEAR OPERAND COLUMN nn

Explanation: An explicit base register has been specified as a relocatable value; an absolute term or expression is required.

Assembler Action: The operand in error (base and displacement) is assembled to 0.

Programmer Response: Use an absolute term or expression to specify the base register.

Severity Code: 12

Module Originating Message: CWAX5M

CWA228 RELOCATABLE DISPLACEMENT IN MACHINE INSTRUCTION NEAR OPERAND COLUMN nn

Explanation: In a machine instruction that has an explicit base register specification, the specification for the displacement field is relocatable. As this would imply a second base register, the combination is invalid.

Assembler Action: The displacement field of the machine instruction is assembled to 0.

Programmer Response: Either specify the displacement as an absolute term or expression, or delete the explicit base register.

Severity Code: 8

Module Originating Message: CWAX5M

CWA229 POSSIBLE REENTERABILITY ERROR

Explanation: This machine instruction could store data into a control section or common area that is not dynamically acquired.

This message is produced only when the RENT assembler option is specified in the PARM field of the EXEC statement.

Assembler Action: The statement is assembled as written.

Programmer Response: If you want reentrant code, correct the instruction so that it references a DSECT or other dynamically acquired space. Otherwise you can suppress reentrant checking by specifying the NORENT assembler option.

Note: Absence of this message does not guarantee reentrant code, as the assembler has no control over addresses actually loaded into base and index registers at program execution time.

Severity Code: 4

Module Originating Message: CWAX5M

CWA230 BASE REGISTER NUMBER GREATER THAN 7 NEAR OPERAND COLUMN nn

Explanation: An explicit base register in a machine instruction is greater than 7.

Assembler Action: The base register field of the machine instruction is assembled to 0.

Programmer Response: Specify the base register in the range of 0 to 7.

Severity Code: 12

Modules Originating Message: CWAX5A, CWAX5M

CWA231 SYMBOL NOT PREVIOUSLY DEFINED–xxxxxxxx

Explanation: A symbol in this statement is used in a way that requires previous defintion, but it has not been previously defined. For example, a symbol in a duplication factor expression of a DC statement must be previously defined.

Assembler Action: The value of the symbol or the expression that contains it is set to 0.

Programmer Response: Define the symbol earlier in the program. Add a defining statement if it does not exist, or place the existing defining statement ahead of the statement that references it.

Severity Code: 8

Module Originating Message: CWAX5D, CWAX5V

CWA233 MORE THAN 6 LEVELS OF PARENTHESES NEAR OPERAND COLUMN nn

Explanation: An expression in this statement contains more than 6 nested levels of parentheses.

Assembler Action: The value of the expression is set to 0.

Programmer Response: Rewrite the expression to reduce the number of levels of parentheses, or use a preliminary statement (such as an EQU) to partially evaluate the expression.

Severity Code: 8

Module Originating Message: CWAX5V

CWA234 PREMATURE END OF EXPRESSION NEAR OPERAND COLUMN nn

Explanation: An expression in this statement ended prematurely due to one of the following errors:

- Unpaired parenthesis
- Invalid character
- Invalid operator
- Operator not followed by a term

Assembler Action: The value of the expression is set to 0.

Programmer Response: Check the expression for omitted or mispunched characters or terms.

Severity Code: 8

Module Originating Message: CWAX5V

CWA235 ARITHMETIC OVERFLOW NEAR OPERAND COLUMN nn

Explanation: The intermediate value of a term or an expression is not in the range -2^{31} through $2^{31}-1$.

Assembler Action: The value of the expression is set to 0.

Programmer Response: Rewrite the expression or term. The assembler computes all values using fixed-point full-word arithmetic. Or, perform arithmetic operations in a different sequence to avoid overflow.

Severity Code: 8

Module Originating Message: CWAX5V

CWA236 ILLEGAL CHARACTER IN EXPRESSION NEAR OPERAND COLUMN nn

Explanation: Syntax error. A character in an expression has no symtactic meaning in the context used; the assembler cannot determine if it is a symbol, an operator, or a delimiter.

Assembler Action: The value of the expression is set to 0.

Programmer Response: Check the expression for unpaired parentheses, invalid delimiter, invalid operator, or a character (possibly unprintable) that is not recognized by the assembler. The 51 characters recognized by the assembler are:

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

Severity Code: 8

Modules Originating Message: CWAX5D, CWAX5V

CWA237 CIRCULAR DEFINITION

Explanation: The value of the first expression in the operand field of an EQU statement is dependent upon the value of the symbol being defined in the name field.

Assembler Action: The value of the expression defaults to the current location counter value.

Programmer Response: Remove circularity in the definition.

Severity Code: 8

Module Originating Message: CWAX5A

CWA238 ILLEGAL AMPERSAND IN SELF-DEFINING TERM NEAR OPERAND COLUMN nn

Explanation: An ampersand in a self-defining term is unpaired and/or not part of a quoted character string.

Assembler Action: The value of the expression containing the self-defining term is set to 0.

Programmer Response: Check that all ampersands in the term are paired and part of a quoted character string. (The only valid use of a single ampersand is as the first character of a variable symbol.) Note that ampersands produced by substitution must also be paired.

Severity Code: 8

Module Originating Message: CWAX5V

CWA240 CHARACTER STRING OR SELFDEFINING TERM TERMINATED BEFORE ENDING QUOTE FOUND

Explanation: The assembler has found what appears to be a quoted character string or a self-defining term, but the closing quote is missing, or an illegal character is found before the closing quote.

Assembler Action: The term or expression is ignored.

Programmer Response: Supply the missing quote or check for other syntax errors.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5V

CWA241 FOURTH OPERAND OF CW NOT BETWEEN 0 AND X'3FFFF'

Explanation: The fourth operand of a CW instruction, which specifies the data address, is outside the range of 0 to X'FFFF'.

Assembler Action: The low-order three bytes of the operand are used.

Programmer Response: Supply a correct term or expression for the second operand.

Severity Code: 12

Modules Originating Message: CWAX5A

CWA242 SPACE OPERAND NOT A SINGLE POSITIVE DECIMAL SELFDEFINING TERM

Explanation: The operand of a SPACE instruction is not a zero or positive decimal self-defining term.

Assembler Action: The SPACE statement is processed as a comment.

Programmer Response: Use a single decimal self-defining term with a zero or positive value.

Severity Code: 4

Module Originating Message: CWAX5A

CWA234 CW OPERAND SUBFIELD NEAR OPERAND COLUMN nn IS MISSING

Explanation: One of the required subfields of the CW operand is missing.

Assembler Action: The value of the CW instruction is set to 0.

Programmer Response: Review the format of the CW instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5A

CWA244 INVALID BRANCH ADDRESS—NOT HALF WORD ALIGNED

Explanation: An RA or RT machine instruction has a branch address that was not on a half-word boundary.

Assembler Action: The value of the instruction is set to zero.

Programmer Response: Review the format of failing RA or RT machine instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5M

CWA245 CW OPERAND VALUE EXCEEDS 3 NEAR OPERAND COLUMN nn

Explanation:

- The first subfield of the CW operand (Command Code) is not in the range of 0-3; or
- The second subfield of the CW operand (Flags) is not in the range of 0-3.

Assembler Action: The instruction is set to zero.

Programmer Response: Review the format of the failing CW instruction and recode.

Severity Code: 12

Module Originating Message: CWAX5A

CWA246 LOCATION COUNTER OVERFLOW

Explanation: The location counter is greater than X'3FFFF' 2^{18}-1.

Assembler Action: The location counter is 4 bytes long. The overflow is carried into the high-order byte and the assembly continues. However, the resulting code will probably not execute correctly.

Programmer Response: The probable cause of the error is a high ORG statement value or a high START statement value. Correct the value or split up the control section.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5M

CWA247 INVALID LOCATION FOR RELOCATABLE R-TYPE CONSTANT

Explanation: A relocatable R-type constant was found within the first two bytes of a control section.

Assembler Action: This is an informational message. The value of the R-type constant, if valid, otherwise is accepted.

Programmer Response: Review the description and use of the R-type constant and recode.

Severity Code: 4

Module Originating Message: CWAX5D

CWA248 INVALID EQUR OPERAND FORMAT

Explanation: An EQUR expression was found that did not have the format R(N).

Assembler Action: The value of the EQUR expression is set to zero.

Programmer Response: Review the format of the invalid EQUR expression and recode.

Severity Code: 12

Module Originating Message: CWAX5A

CWA249 REGISTER/BYTE DESIGNATOR OF EQUR EXPRESSION IS NOT AN ABSOLUTE VALUE

Explanation: An EQUR instruction was detected where either the register value or the byte designator was not absolute.

Assembler Action: The value of the EQUR expression is set to zero.

Programmer Response: Review the format of the invalid EQUR and expression recode.

Severity Code: 8

Module Originating Message: CWAX5A

CWA250 REGISTER DESIGNATOR IS NOT AN ABSOLUTE VALUE NEAR OPERAND COLUMN nn

Explanation: A machine instruction that has an operand subfield of the format R(N) specified an 'R' that did not have an absolute value.

Assembler Action: The value of the instruction is set to zero.

Programmer Response: Review the format of the invalid instruction and recode.

Severity Code: 8

Module Originating Message: CWAX5M

CWA251 BYTE DESIGNATOR IS A SYMBOLIC REGISTER EXPRESSION NEAR OPERAND COLUMN nn

Explanation: An instruction that uses the format R(N) in its operand was found to have a non-absolute value for 'N'.

Assembler Action: The value of the instruction is set to zero.

Programmer Response: Review the invalid instruction and recode.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5M

CWA252 BYTE DESIGNATOR IS NOT 0 OR 1 NEAR OPERAND COLUMN nn

Explanation: An instruction that uses the format R(N) in its operand did not have the 'n' value in the range 0-1. This value designates which (of the two) bytes in the register is being addressed.

Assembler Action: The instruction is set to zero.

Programmer Response: Review the invalid instruction and recode.

Severity Code: 8

Modules Originating Message: CWAX5A, CWAX5M

CWA253 RELOCATABLE R-TYPE ADDRESS CONSTANT IN THIS ASSEMBLY

Explanation: This is an informational message only indicating that at least one relocatable R-type address constant exists in this assembly.

Assembler Action: None--informational message only.

Programmer Response: Review the requirements for an R-type address constant.

Severity Code: 4

Module Originating Message: CWAX6B

CWA254 ILLEGAL FORMAT OF SECOND OPERAND OF END STATEMENT

Explanation: Second operand of END instruction is inconsistent with the format required.

Assembler Action: Second operand is ignored.

Programmer Response: Correct the operand.

Severity Code: 4

Module Originating Message: CWAX5A

CWA255 FIXED-POINT EXPRESSION ERROR NEAR OPERAND COLUMN nn

Explanation: An error occurred during conversion of a decimal number into a fixed-point number.

Assembler Action: The number is assembled as zeros.

Programmer Response: Check the scale and exponent modifier of the number for validity.

Severity Code: 8

Module Originating Message: CWAX5D

CWA256 SYSGO DD CARD MISSING—NOOBJECT OPTION USED

Explanation: A DD statement for the SYSGO data set is not included in the JCL for this assembly. The SYSGO data set normally receives the object module output of the assembler when it is to be used as input to the linkage editor or loader, executed in the same job.

Assembler Action: The program is assembled using the NOOBJECT option. No output is written on SYSGO. If the DECK option is specified, the object module will be written on the device specified in the SYSPUNCH DD statement.

Programmer Response: Optional. If the assembly is error free and the object module has been produced on SYSPUNCH, you can execute it without reassembling. Otherwise, reassemble the program and include a SYSGO DD statement in the JCL or use a cataloged procedure that includes it.

Severity Code: 16

Module Originating Message: CWAX6B

CWA257 SYSPUNCH DD CARD MISSING—NODECK OPTION USED

Explanation: A DD statement for the SYSPUNCH data set is not included in the JCL for this assembly. The SYSPUNCH data set is normally used when the object module of the assembly is directed to the card punch.

Assembler Action: The program is assembled using the NODECK option. No deck is punched on SYSPUNCH. If the OBJECT option has been specified, the object module will be written on the device specified in the SYSGO DD statement.

Programmer Response: Optional. The object module can be link edited and executed from SYSGO instead of SYSPUNCH by adjusting JCL. Otherwise, if you want a punch data set, reassemble the program with a SYSPUNCH DD statement.

Severity Code: 16

Module Originating Message: CWAX6B

CWA258 INVALID ASSEMBLER OPTION ON EXEC CARD—OPTION IGNORED

Explanation: One or more of the assembler options specified in the PARM field of the EXEC statement are invalid. The error may be caused by use of the wrong option, a misspelled option, or syntax errors in coding the options.

Assembler Action: Invalid options are ignored. The assembly is performed using the valid options.

Programmer Response: Check the spelling of the options, the length of the option list (100 characters maximum), and the syntax of the option list. The options must be separated by commas, and parentheses in the option list (including SYSPARM) must be paired. Two quotes or ampersands are needed to represent a single quote or ampersand in a SYSPARM character string.

Severity Code: 16

Module Originating Message: CWAX6B

CWA259 RELOCATABLE Y-TYPE ADDRESS CONSTANT IN THIS ASSEMBLY

Explanation: This is an informational message only indicating that a least one relocatable Y-type address constant exists in this assembly.

Assembler Action: None—informational message only.

Programmer Response: Review the requirements for a Y-type address constant.

Severity Code: 4

Module Originating Message: CWAX6B

CWA260 ASSEMBLY TERMINATED—DD CARD MISSING FOR SYSxxx

Explanation: This assembler job step cannot be executed because a DD statement is missing for one of the following assembler data sets: SYSUT1, SYSUT2, SYSUT3, or SYSIN. The missing DD statement is indicated in the message text.

Assembler Action: The assembly is terminated before any statements are assembled. No assembler listing is produced, so this message is printed on the system output unit following the job control language statements for the assembly job step and on the operator's console.

Programmer Response: Supply the missing DD statement and reassemble the program. The cataloged procedures supplied by IBM contain all the required DD statements.

If the problem recurs, do the following before calling IBM:

- Have your source program, macro definitions, and associated listings available.
- If a COPY statement was used, execute the IEBPTPCH utility to obtain a copy of the partitioned data set member specified in the COPY statement.
- Make sure that MSGLEVEL=(1,1) was specified in the JOB statement.

Operator Response: If possible, supply the missing DD statement in the JCL statements for the assembly and run the job again.

Severity Code: 20

Modules Originating Message: CWAX0D, CWAX0I

CWA261 ASSEMBLY TERMINATED—PERM I/O ERROR—jobname, stepname, unit address, device type, ddname, operation attempted, error description

Explanation: A permanent I/O error occurred on the assembler data set indicated in the message text. This message, produced by a SYNADAF macro instruction, also contains more detailed information about the cause of the error and where it occurred.

Assembler Action: The assembly is terminated. Depending on where the error occurred, the assembly listing up to the point of the I/O error may be produced. If the listing is produced, this message appears on it. If the listing is not produced, this message appears on the operator's console and on the system output unit following the job control language statements for the assembler job step.

Programmer Response: If the I/O error is on SYSIN or SYSLIB, you may have concatenated the input or library data sets incorrectly. Make sure the DD statement for the data set with the largest blocksize (BLKSIZE) is placed in the JCL before the DD statements of the data sets concatenated to it. Also, make sure that all input or library data sets have the same device class (all DASD or all tape).

In any case, reassemble the program; it may assemble correctly. If the problem recurs, do the following before calling IBM:

- Have your source program, macro definitions, and associated listings available.
- If a COPY statement was used, execute the IEBPTPCH utility to obtain a copy of the partitioned data set member specified in the COPY statement.
- Make sure that MSGLEVEL=(1,1) was specified in the JOB statement.

Operator Response: If the I/O error is on SYSUT1, SYSUT2, or SYSUT3, allocate the data set to a different volume and rerun the job. If the I/O error is on tape, check the tape for errors.

Severity Code: 20

Modules Originating Message: CWAX0C, CWAX0E, CWAX0G, CWAX0I

CWA262 ASSEMBLY TERMINATED—INSUFFICIENT MEMORY

Explanation: The assembler was unable to get at least 32K bytes of main storage for working storage, utility file buffers, and assembler tables and constants.

Assembler Action: The assembly is terminated before any statements are assembled. No assembler listing is produced, so this message is printed on the system output device following the JCL statements for the assembler job step and on the operator's console.

Programmer Response: Increase the size of the region or partition allocated to the assembler. Reassemble the program. If the problem recurs, do the following before calling IBM:

- Have your source program, macro definitions, and associated listings available.
- Make sure that MSGLEVEL=(1,1) was specified in the JOB statement.

Operator Response:

- Increase the size of the region allocated on the JOB card or on the EXEC card for the assembler job step and rerun the job; or
- Run the job in a larger partition.

Severity Code: 20

Modules Originating Message: CWAX0B, CWAX0I

CWA263 ASSEMBLY TERMINATED—PROGRAM LOGIC ERROR

Explanation: The assembly has been abnormally terminated because of a logic error within the assembler.

Assembler Action: Abnormal termination. No assembler listing is produced; the assembler prints this message on the system output device following the JCL statements for the assembler job step.

Programmer Response: Do the following before calling IBM:

- Have your source program, macro definitions, and associated listings available.
- If a COPY statement was used, execute the IEBPTPCH utility program to obtain a copy of the partitioned data set member specified in the operand field of the COPY statement.
- Make sure that MSGLEVEL=(1,1) was specified on the JOB statement.

Severity Code: 20

Module Originating Message: CWAX0I

CWA264 TOO MANY ESD ENTRIES

Explanation: More than 399 entries have been made in the External Symbol Dictionary. Entries in the External Symbol Dictionary are made for the following: control sections, dummy sections, external references (EXTRN and WXTRN), ENTRY symbols, and external dummy sections.

Assembler Action: Entries over the 399 limit are not added to the dictionary and linkage is not provided for them.

Programmer Response: Subdivide your porgram and reassemble each section individually. Be sure that there are not more than 399 ESD entries in each assembly.

Severity Code: 16

Module Originating Message: CWAX6B

CWA265 SYMBOL RESOLUTION DATA AREA HAS BEEN EXHAUSTED

Explanation:

- Too many literals have been encountered since a LTORG statement was encountered, and the assembler has filled available work space with literals; or
- The assembler has filled available work space with ESD entries.

Assembler Action: No assembly is performed.

Programmer Response:

- Insert more LTORG statements in the source deck or allocate more working storage to the assembler; or
- If there are more than 399 ESD entries in your source module, segment it into several modules.

Severity Code: 16

Module Originating Message: CWAX6B

CWA266 LAST ASSEMBLER PHASE LOADED WAS xxxxxxxx

Explanation: This message is issued by the abort routine when the assembly is abnormally terminated.

Assembler Action: Abnormal termination.

Programmer Response: Correct problems indicated by other error messages and reassemble.

Severity Code: 4

Module Originating Message: CWAX0I

CWA267 SYSPRINT DD CARD MISSING—NOLIST OPTION USED

Explanation: The LIST option is specified, but the DD statement for the SYSPRINT data set is not included in the JCL for this assembly. The SYSPRINT data set holds the object module output of the assembly normally directed to the printer.

Assembler Action: The program is assembled using the NOLIST option. The message is printed on the system output device following the JCL statements for the assembler job step and on the operator's console.

Programmer Response: If you want a listing, reassemble the program with a SYSPRINT DD statement. Otherwise, do not specify the LIST option.

Operator Response: Supply, if possible, a SYSPRINT DD card for the assembler job step and rerun the job.

Severity Code: 16

Module Originating Message: CWAX6B

CWA268 SYSTERM DD CARD MISSING—NOTERMINAL OPTION USED

Explanation: The TERMINAL option is specified, but the DD statement for the SYSTERM data set is not included in the JCL statements for this assembly. The SYSTERM data set contains diagnostic information output of the assembly, normally directed to a remote terminal.

Programmer Response: If you want a SYSTERM listing, reassemble the program with a SYSTERM DD statement. Otherwise, do not specify the TERMINAL option.

Operator Response: Supply, if possible, a SYSTERM DD card for the assembly step and rerun the job.

Severity Code: 16

Module Originating Message: CWAX6B

CWA269 SYSLIB DD CARD MISSING

Explanation:

- A COPY instruction appears in the assembly, but no SYSLIB DD statement is included in the JCL statements; or
- An operation code that is not a machine, assembler, or source macro instruction operation code appears in the assembly, but no SYSLIB DD statement is included in the JCL statements. The assembler assumed the operation code to be a library macro operation code.

Assembler Action:

- The COPY instruction is ignored; or
- The operation code is treated as an undefined operation code.

Programmer Response: Supply the missing DD statement or correct the invalid operation code.

Severity Code: 16

Module Originating Message: CWAX6B