GC26-3771-3
File No. S360-21(OS)

Program Product

OS Assembler H Language

Program Number 5734-AS1

IBM

Fourth Edition (June, 1974)

This is a reprint of GC26-3771-2 incorporating changes issued in Technical Newsletter GN33-8149, dated August 28, 1972. This edition applies to version 4 of the Assembler H Program Product (Program Number 5734-AS1). Information in this publication is subject to change. Before using this publication, be sure you have the latest edition and any Technical Newsletters.

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

Read This First

This manual is divided into two independent parts. You should only use one of the parts.

If your installation uses VS, you should read the first part, 'Part I: For the VS User'. This part is written as a supplement to OS/VS and DOS/VS Assembler Language, Order No. GC33-4010, which you must use in conjunction with this supplement. Further instructions on how to use this supplement are contained in the Preface, which you should read to be able to use this supplement correctly.

If your installation uses OS/MFT or OS/MVT, you should read the second part, 'Part II: For the MFT or MVT User'. This part is written as a supplement to OS Assembler Language, Order No. GC28-6514, which you must use in conjunction with this supplement. Further instructions on how to use this supplement are contained in the Preface, which you should read to be able to use this supplement correctly.

Part I: For the VS User

Preface

Purpose and Subject of This Part

This part is a supplement to the OS/VS and DOS/VS Assembler Language manual. That manual describes the language supported by the OS/VS Assembler, which is a subset of the language supported by Assembler H. This supplement defines the differences between the two languages; together the two manuals give you the information you need to write instructions that can be assembled by Assembler H.

Who This Part Is For

This part has the same audience as defined for the parent manual. The knowledge that the reader is assumed to have is defined in that manual.

How to Use This Part

First read the parent manual in the section relevant to your enquiries. Then check the table of contents in this supplement for the identical section heading. If listed, this supplement contains a section on the subject. The relevant section describes the differences between the language supported by the OS/VS Assembler and the language supported by Assembler H.

Organization of This Part

This supplement has exactly the same organization as the parent manual: chapter and section headings are the same, but only those chapters and sections which reflect differences between the two languages are included in the supplement. Chapter and section headings that are common to both the parent manual and the supplement are underlined in the table of contents in this supplement.

Bibliography

OS/VS and DOS/VS Assembler Language, GC33-4010. This manual must be used with the supplement.

OS Assembler H General Information Manual, GC26-3758. This manual gives general information about Assembler H, describing the language and facilities of the program, and comparing it with the OS/VS Assembler.

Contents

Section B: Coding Conventions

B1 – Coding Specifications

B1A -- STATEMENT FIELD BOUNDARIES

The Identification-Sequence Field

COLUMNS CHECKED BY ISEQ : The columns checked by the ISEQ facility are not restricted to columns 73 through 80, or by the boundaries determined by any ICTL instruction. The columns specified in the ISEQ instruction can be anywhere on the input cards; they can also coincide with columns that are occupied by the instruction field.

B1B -- CONTINUATION LINES

NORMAL STATEMENT FORMAT: Nine continuation lines are allowed. Thus, a single assembler language statement can occupy up to ten lines.

ALTERNATE STATEMENT FORMAT : The alternate statement format, which allows as many continuation lines as are needed, can be used for the following instructions:

- the prototype statement of a macro definition
- the macro instruction statement
- the AGO conditional assembly statement
- the AIF conditional assembly statement
- the GBLA, GBLB, and GBLC conditional assembly statements
- the LCLA, LCLB, and LCLC conditional assembly statements
- the SETA, SETB, and SETC conditional assembly statements

Examples of the alternate statement format for each of these instructions are given in the context where the individual instruction is described.

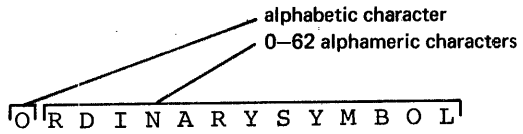
Section C: Assembler Language Structure

C4 -- Terms

C4A -- SYMBOLS

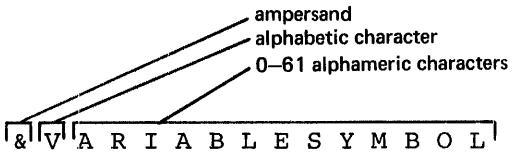
Specifications

ORDINARY SYMBOLS: The format of an ordinary symbol is:



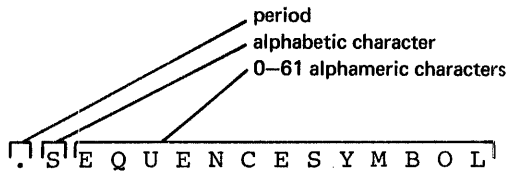
examples: ORDSYM#435A
K4
B49467LITTLENAIL

VARIABLE SYMBOLS: The format of a variable symbol is:



examples: &VARYINGSYMABC
&F346944
&@ME

SEQUENCE SYMBOLS: The format of a sequence symbol is:



examples: .BLABEL04
.#359
.BRANCHTOMEFIRST

Restrictions on Symbols

USE OF DUPLICATE SYMBOL IN NAME FIELD OF CERTAIN INSTRUCTIONS: The symbol in the name field of a LOCTR instruction can be the same as the name of a previous START, CSECT, DSECT, COM, or LOCTR instruction. It identifies the resumption of the location counter specified by the name field.

PREVIOUS DEFINITION NOT REQUIRED: Previous definition is not required for symbols in the following operand entries:

EQU (first operand)
CNOP
ORG
DC and DS (in duplication factor and modifier expressions)

Previously Defined Symbols

Ordinary symbols do not have to be previously defined if they appear in operand expressions of ORG and CNOP instructions, in modifier expressions of DC, DS, and DXD statements, in the first operand of EQU statements, or in Q-type constants.

Allowing forward reference in the above statement types creates two new kinds of errors which you should guard against.

- Circular definition of symbols, such as

```
X     EQU     Y
Y     EQU     X
```

- Circular location-counter dependency, as in this example:

```
A     DS      (B-A)C
B     LR      1,2
```

Statement A cannot be resolved because the value of the duplication factor is dependent on the location of B, which is in turn dependent upon the length of A.

Literals may contain symbolic expressions in modifiers, but any ordinary symbols used must have been previously defined. See Section L, "Attribute Definition and Lookahead", in this manual.

C4B -- LOCATION COUNTER REFERENCE

MULTIPLE LOCATION COUNTERS: When using Assembler H, you can specify multiple location counters for your control sections. See "E2C -- Location Counter Setting".

C4D -- OTHER ATTRIBUTE REFERENCES

ADDITIONAL ATTRIBUTE AVAILABLE: When using Assembler H there is one attribute additional to the six attributes available when using the OS/VS Assembler. The additional attribute is the defined (D') attribute (see "L1B -- Data Attributes").

Section E: Program Sectioning

E1 - The Source Module

E1A -- THE COPY INSTRUCTION

NESTED COPY INSTRUCTIONS: Copied code may contain other COPY statements. The OS/VS Assembler allows up to five levels of nested COPY instructions. The Assembler H has no restrictions on the nesting of COPY statements.

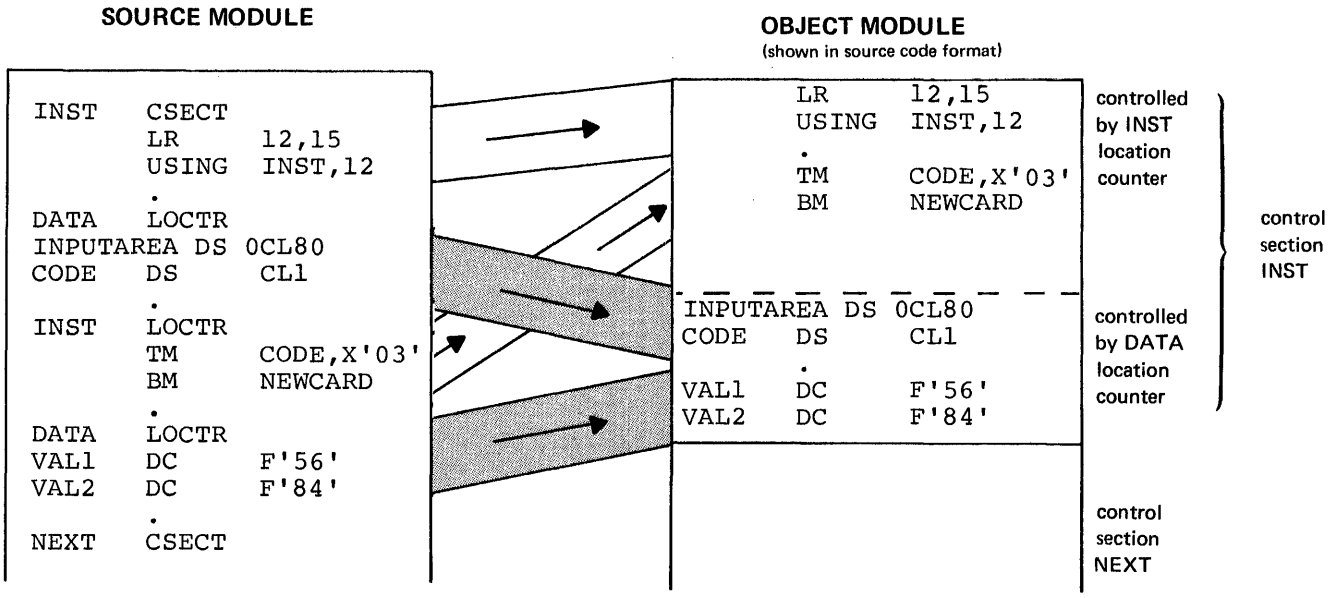
The COPY nesting must not be recursive. Thus, if the statement "COPY A" is coded and A contains a statement "COPY B", B must not contain a statement "COPY A".

E2 - General Information About Control Sections

E2C -- LOCATION COUNTER SETTING

USE OF MULTIPLE LOCATION COUNTERS: Assembler H allows you to use multiple location counters for each individual control section. You use the LOCTR instruction (whose format and specifications are described under "E3E - The LOCTR Instruction") to assign different location counters to different parts of a control section. The assembler will then rearrange and assemble the coding together according to the different location counters you have specified: all coding using the first location counter will be assembled together, then the coding using the second location counter will be assembled together, etc.

A practical use of multiple location counters is illustrated below. There, the programmer has interspersed executable instructions and data areas throughout the coding in their logical sequence, each group of instructions preceded by a LOCTR instruction identifying the location counter under which it is to be assembled. The assembler will rearrange the control section so that the executable instructions are grouped together and the data areas together.



E2G -- EXTERNAL SYMBOL DICTIONARY ENTRIES

LIMITS TO THE NUMBER OF ESD ENTRIES IN A MODULE: There is no limit to the number of individual control sections and external symbols that can be defined in a source module.

E3 - Defining a Control Section

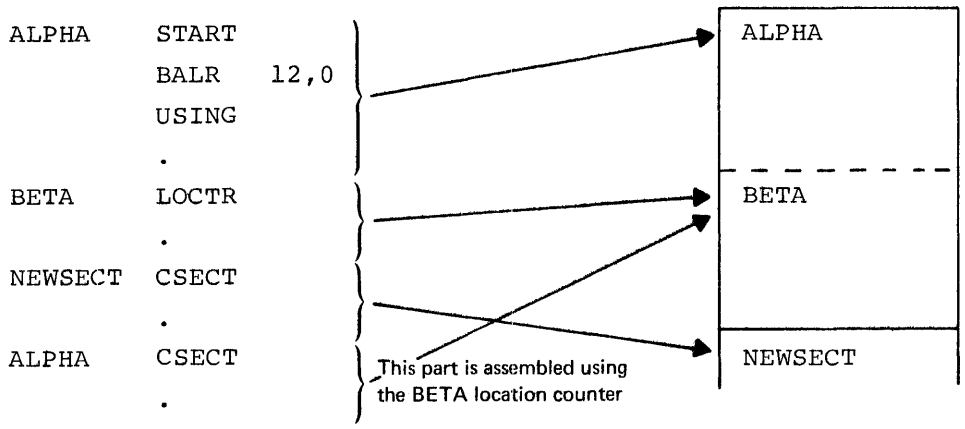
E3A -- THE START INSTRUCTION

Specifications

EXPRESSION ALLOWED IN OPERAND: The operand of a START instruction can be an absolute expression, provided that any symbols referenced have been previously defined.

E3B -- THE CSECT INSTRUCTION

RESUMING AN INTERRUPTED CONTROL SECTION: When an interrupted control section is resumed using the CSECT instruction, the location counter last specified in that control section will be resumed. Consider the following coding:



For further information on the use of multiple location counters, refer to "E3E -- The LOCTR Instruction".

E3C -- THE DSECT INSTRUCTION

RESUMING AN INTERRUPTED DUMMY CONTROL SECTION: When an interrupted dummy control section is resumed using the DSECT instruction, the location counter last specified in that control section will be resumed.

E3D -- THE COM INSTRUCTION

RESUMING AN INTERRUPTED COMMON CONTROL SECTION: When an interrupted common control section is resumed using the COM instruction, the location counter last specified in that control section will be resumed.

E3E -- THE LOCTR INSTRUCTION

Purpose

The LOCTR instruction allows you to specify multiple location counters within a control section. The assembler assigns consecutive addresses to the segments of code using one location counter before it assigns addresses to segments of coding using the next location counter. By using the LOCTR instruction, you can code your control section in a logical order. For example, you can code work areas and data constants within the section of code using them without having to branch around them. The concept of multiple location counters is described more fully under "E2C - Location Counter Setting".

Specifications

The LOCTR instruction has the following format:

Name	Operation	Operand
A variable or ordinary symbol	LOCTR	Blank

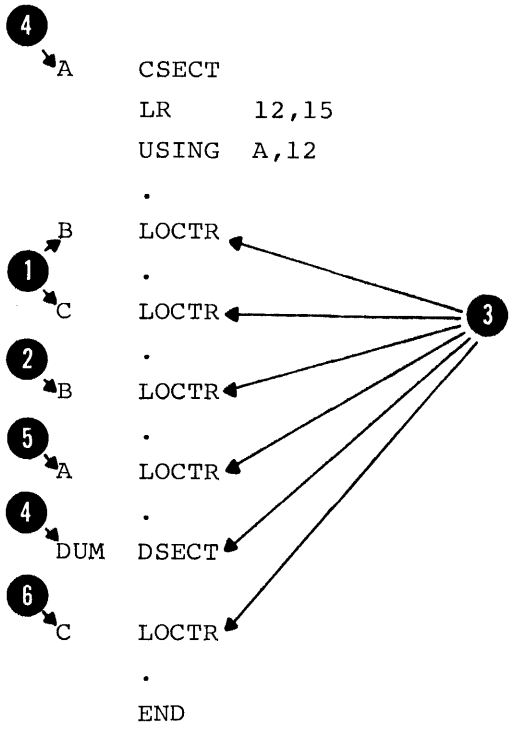
1. The LOCTR instruction defines a location counter
2. or resumes a previously defined location counter.

3. A location counter remains in use until it is interrupted by a LOCTR, CSECT, DSECT, or COM instruction.

4. The first location counter of a control section is defined by the name of the START, CSECT, DSECT, or COM instruction defining the section.

5. A LOCTR instruction with the same name as a control section resumes the first location counter of that section.

6. A LOCTR instruction with the same name as a LOCTR instruction in a previous control section causes that control section to be resumed using the location counter specified.



A control section cannot have the same name as a previous LOCTR instruction. A LOCTR instruction placed before the first control section definition will initiate an unnamed control section before the LOCTR instruction is processed.

The length attribute of a LOCTR name is 1.

LOCTR instructions do not force alignment; code running under a location counter other than the first location counter of a control section will be assembled starting at the next available byte after the previous segment.

E5 - Defining an External Dummy Section

E5A -- THE DXD INSTRUCTION

ENTRY IN THE EXTERNAL SYMBOL DICTIONARY: An external dummy section identified by a DXD instruction will not generate an entry in the external symbol dictionary (ESD) unless it is referenced by a Q-type address constant.

Section F: Addressing

F2 -- Addressing Between Source Modules: Symbolic Linkage

NO RESTRICTION ON THE TOTAL NUMBER OF EXTERNAL SYMBOLS: There is no restriction on the number of control sections, external symbols, and external dummy sections allowed by the assembler. The maximum number depends on main storage available during linkage editing.

G2 -- Defining Symbols

G2A -- THE EQU INSTRUCTION

Specifications

PREVIOUS DEFINITION OF SYMBOLS IN THE FIRST OPERAND NOT REQUIRED: Any symbols used in the first operand of the EQU statement need not be previously defined.

COMPLEXLY RELOCATABLE FIRST OPERAND: The first operand of the EQU instruction may assume any value allowed for an assembly expression: absolute (including negative), relocatable, or complexly relocatable.

If the expression in the first operand is complexly relocatable, the whole expression rather than its value is assigned to the symbol. During the evaluation of any expression that includes a complexly relocatable symbol, that symbol is replaced by its own defining expression.

Consider the following example, in which A1 and A2 are defined in one control section and B1 and B2 in another:

Name	Operation	Operand
X	EQU	A1+B1
Y	EQU	X-A2-B2

The first EQU statement assigns a complexly relocatable expression (A1+B1) to X. During the evaluation of the expression in the second EQU statement, X is replaced by its defining relocatable expression (A1+B1), and the assembler evaluates the resulting expression (A1+B1-A2-B2) and assigns an absolute value to Y, because the relocatable terms in the expression are paired.

G3 -- Defining Data

G3B -- GENERAL SPECIFICATIONS FOR CONSTANTS

Subfield 1: Duplication Factor

SYMBOLS NEED NOT BE PREVIOUSLY DEFINED: Symbols used in operand subfield 1 need not be previously defined. This does not apply to literals.

Subfield 3: Modifiers

SYMBOLS NEED NOT BE PREVIOUSLY DEFINED: Symbols used in operand subfield 3 need not be previously defined. This does not apply to literals.

G3M -- THE Q-TYPE ADDRESS CONSTANT

PREVIOUS DEFINITION OF REFERENCED NAMES NOT REQUIRED: The DXD or DSECT names referenced in the Q-type address constant need not be previously defined.

Section H: Controlling the Assembler Program

H1 -- Structuring a Program

H1A -- THE ORG INSTRUCTION

RESTRICTION ON ORG WHEN THE LOCTR INSTRUCTION IS USED: If you specify multiple location counters with the LOCTR instruction, the ORG instruction can alter only the location counter in use when the instruction appears. Thus you cannot control the structure of the whole control section using ORG, but only the part that is controlled by the current location counter.

Specifications

PREVIOUS DEFINITION OF OPERANDS NOT REQUIRED: In general, symbols used in the operand field need not have been previously defined. However, the simply relocatable component of the expression (that is, the unpaired relocatable term) must have been previously defined or be equated to a previously defined value.

MULTIPLE LOCATION COUNTERS: ORG changes the value of the current location counter only. Therefore, it cannot reference a location outside the current location counter, even though that location may belong to the same contrcl section.

H1C -- THE CNOP INSTRUCTION

PREVIOUS DEFINITION OF SYMBOLS NOT REQUIRED: Symbols appearing in the operand field of CNOP instructions need not be previously defined.

H2 -- Determining Statement Format and Sequence

H2B -- THE ISEQ INSTRUCTION

WHAT COLUMNS CAN BE SEQUENCE CHECKED?: The columns to be sequence checked by the ISEQ facility can be anywhere on the cards in the input. Thus they can also be between the begin and end columns.

H5 -- Redefining Symbolic Operation Codes

H5A -- THE OPSYN INSTRUCTION

PLACEMENT OF OPSYN STATEMENTS: The OPSYN instruction can be coded anywhere in the program to redefine an operation code.

REDEFINING CONDITIONAL ASSEMBLY INSTRUCTIONS: A redefinition of a conditional assembly operation code will have an effect only on macro definitions appearing after the OPSYN instruction. Thus, the new definition is not valid during the processing of subsequent macro instructions calling a macro that was defined prior to the OPSYN statement.

Any OPSYN statement redefining the operation code of an instruction generated from a macro instruction will, however, be valid, even if the definition of the macro was made prior to the OPSYN statement. The following example illustrates this difference between conditional assembly instructions and model statements within macro instructions.

Name	Operation	Operand	Comment
	MACRO		macro header
	MAC	....	macro prototype
	AIF	....	
	MVC	....	
	.		
	MEND		macro trailer
	.		
AIF	OPSYN	AGO	assign AGO properties to AIF
MVC	OPSYN	MVI	assign MVI properties to MVC
	.		
	MAC	....	macro call
	[AIF	....	evaluated as AIF instruction
			generated AIFs not printed]
+	MVC	....	evaluated as MVI instruction
	.		
	.		open code started at this point
	AIF	....	evaluated as AGO instruction
	MVC	....	evaluated as MVI instruction

AIF and MVC instructions are used in a macro definition. OPSYN instructions are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC, after the macro definition has been edited. In subsequent calls to that macro, AIF is still defined as an AIF operation while MVC is treated as an MVI operation. In open code following the OPSYN instructions, the operations of both instructions are derived from their new definitions. If the macro is redefined, either by means of a loop to a point before the macro definition or by a subsequent macro definition defining the same macro, the new definitions of AIF and MVC (that is, AGO and MVI) will be fixed for future expansions.

J1 -- Using a Macro Definition

J1B -- SPECIFICATIONS

Where to Define a Macro in a Source Module

MACRO DEFINITIONS NEED NOT APPEAR AT THE BEGINNING: Macro definitions can appear anywhere in a source module. They remain in effect for the rest of your source module or until another macro definition defining a macro with the same operation code is encountered. Thus, you can redefine a macro at any point in your program. The new definition will be used for all subsequent calls to the macro in the program.

NESTED MACRO DEFINITIONS: Macro definitions can also appear inside other macro definitions. There is no limit to the levels of macro definitions permitted.

The assembler does not process inner macro definitions until it finds the definition during the processing of a macro instruction calling the outer macro.

Consider the following example:

```
| Name  | Operation | Operand            | Comments                        |
|-------|-----------|--------------------|---------------------------------|
|       | MACRO     |                    | macro header for outer macro    |
|       | OUTER     | &A,&C=             | macro  prototype                |
|       | AIF       | ('&C' EQ '').A     |                                 |
|       | MACRO     |                    | macro header for inner macro    |
|       | INNER     |                    | macro  prototype                |
|       | .         |                    |                                 |
|       | .         |                    |                                 |
|       | MEND      |                    | macro trailer for inner macro   |
| .A    | ANOP      |                    |                                 |
|       | .         |                    |                                 |
|       | MEND      |                    | macro trailer for outer macro   |
```

The assembler does not process the macro definition for INNER until OUTER is called with a value for &C other than a null string.

J2 -- Parts of a Macro Definition

J2C -- THE MACRO PROTOTYPE STATEMENT: CODING

Specifications

INSERTING COMMENT STATEMENTS BEFORE THE PROTOTYPE: The Assembler H allows internal comments statements between the macro header and the macro prototype. Internal comments statements are listed only with the macro definition.

MAXIMUM NUMBER OF OPERANDS: The maximum number of symbolic parameters allowed by Assembler H is 240.

J2D -- THE MACRO PROTOTYPE STATEMENT: ENTRIES

The Operation Entry

OPERATION ENTRY PREVIOUSLY DEFINED: Any operation code can be specified in the prototype operation field. If the entry is the same as an assembler or machine operation code, the new definition overrides the previous use of the symbol. The same is true if the specified operation code has been defined earlier in the program as a macro, or is the operation code of a library macro.

The Operand Entry

MAXIMUM NUMBER OF OPERANDS: The maximum number of symbolic parameters that can be specified in the operand field is 240.

J2E -- THE BODY OF A MACRO DEFINITION

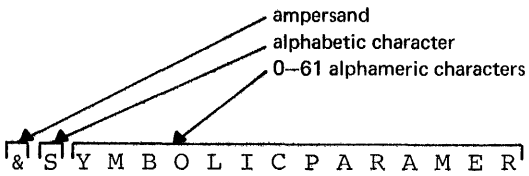
Specifications

NESTED MACRO DEFINITIONS: You can include macro definitions in the body of a macro definition. This is explained under "J1 -- Using a Macro Definition".

J3 - Symbolic Parameters

General Specifications

FORMAT: The format of a symbolic parameter is:



J4 - Model Statements

J4B -- SPECIFICATIONS

Format of Model Statements

LISTING OF GENERATED FIELDS: The different fields in a macro-generated statement or a statement generated in open code appear in the listing in the same column as they are coded in the model statement, with the following exceptions:

- If the substituted value in the name or operation field is too large for the space available, the next field will be moved to the right with one blank separating the fields.
- If the substituted value in the operand field causes the remarks field to be displaced, the remarks field is written on the next line, starting in the column where it is coded in the model statement.
- If the value substituted in the operation field of a macro-generated statement contains leading blanks, the blanks are ignored.
- If the value substituted in the operation field of a model statement in open code contains leading blanks, the blanks will be used to move the field to the right.
- If the value substituted in the operand field contains leading blanks, the blanks will be used to move the field to the right.
- If the value substituted contains trailing blanks, the blanks are ignored.

Rules for Model Statement Fields

SPECIFYING OPSYN AS A MODEL STATEMENT: OPSYN is allowed in the operation field of a model statement. Substitution is allowed in the name, operation, and operand fields of the instruction.

J5 -- Processing Statements

J5F -- THE AREAD INSTRUCTION

Purpose

The AREAD instruction allows you to assign to a SETC symbol the character string value of a card that is placed immediately after a macro instruction. The AREAD facility functions in much the same way as symbolic parameters, but instead of supplying your input to macro processing as part of the macro instruction, you add the values in the form of whole 80-character input records that follow immediately after the macro instruction. Any number of successive cards can be read into the macro for processing.

Specifications

The AREAD instruction can only be used inside macro definitions. Its format is:

Name	Operation	Operand
Any SETC symbol	AREAD	[NOSTMT] [NOPRINT]

The SETC symbol in the name field may be subscripted. When the assembler encounters the AREAD statement during the processing of a macro instruction, it reads the source card following the macro instruction and assigns an 80-character string to the SETC symbol in the name field. In the case of nested macros, it reads the card following the outermost macro instruction.

If no operand is specified, the card to be read by AREAD is printed in the listing and assigned a statement number. If NOSTMT is specified in the operand, the card is printed, but not given any statement number. If NOPRINT is specified, the card does not appear in the listing and no statement number is assigned to it.

Repeated AREAD statements read successive cards.

The records read by the AREAD instruction can be in code brought in with the COPY instruction, if the macro instruction appears in such code. If no more records exist in the code brought in by the COPY instruction, subsequent cards are read from the ordinary input stream.

Example:

```
            MACRO
            MAC1
            .
    &VAL    AREAD
            .
    &VAL1   AREAD
            .
            MEND
            CSECT
            .
            MAC1
    THIS IS THE CARD TO BE PROCESSED FIRST
    THIS IS THE SECOND CARD FOR THE SECOND AREAD
            .
            END
```

J6 - Comments Statements

J6A -- INTERNAL MACRO COMMENTS STATEMENTS

PLACEMENT OF COMMENTS STATEMENTS: The Assembler H allows internal comments statements between the macro header and the prototype statement.

J7 - System Variable Symbols

ADDITIONAL SYSTEM VARIABLE SYMBOL AVAILABLE: Assembler H supports the system variable symbol &SYSLOC which gives you the name of the location counter in effect when the macro instruction appears. See below under "J7G -- &SYSLOC".

J7C -- &SYSLIST

USING MORE THAN TWO SUBSCRIPTS: When referring to multilevel sublists in operands of macro instructions, reference to elements of inner sublists can be made using the appropriate number of subscripts for &SYSLIST. See examples under "K4 - Sublists in Operands".

J7E -- &SYSPARM

DEFAULT VALUE FOR &SYSPARM: If &SYSPARM is omitted in the PARM field of the EXEC statement that invokes the assembler, the system parameter is assigned the value that was specified when the Assembler H was generated (added to your system).

J7G -- &SYSLOC

Purpose

You can use &SYSLOC in a macro definition to generate the name of the location counter currently in effect. If you have not coded a LOCTR instruction between the macro instruction and the preceding START, CSECT, DSECT, or COM instruction, the value of &SYSLOC is the same as the value of &SYSECT.

Specifications

The assembler assigns to the system variable symbol &SYSLOC a local read-only value each time a macro definition containing it is called. The value assigned is the symbol representing the name of the location counter in use at the point where the macro is called.

&SYSLOC can only be used in macro definitions.

Section K: The Macro Instruction

K1 - Using a Macro Instruction

K1B -- SPECIFICATIONS

PLACEMENT OF MACRO INSTRUCTIONS: A macro instruction can be coded anywhere in your program, if the assembler finds its definition either in a macro library, or in the source module before it finds the macro instruction.

K2 - Entries for Fields

K2B -- THE OPERATION ENTRY

GENERATED MACRO INSTRUCTION: You can code a variable symbol in the operation field of a macro instruction if the value of the variable symbol specifies the operation code of a library or source macro that has been previously defined. Thus, if MAC1 has been defined as a macro, you can use the following statements to call it:

```
&CALL   SETC    'MAC1'
        &CALL
```

K4 - Sublists in Operands

MULTILEVEL SUBLISTS: You can specify multilevel sublists (sublists within sublists) in macro operands. The depth of this nesting is limited only by the constraint that the total operand length must not exceed 255 characters. Inner elements of the sublists are referenced using additional subscripts on symbolic parameters or on &SYSLIST.

N'&SYSLIST with an n-element subscript array gives the number of operands in the indicated n-th level sublist. The number attribute (N') and a parameter name with an n-element subscript array gives the number of operands in the indicated (n+1)th level sublist.

Example: If &P is the first positional parameter and the value assigned in a macro instruction is (A,(B,(C)),D) then:

&P	=&SYSLIST(1)	=(A,(B,(C)),D)
&P(1)	=&SYSLIST(1,1)	= A
&P(2)	=&SYSLIST(1,2)	= (B,(C))
&P(2,1)	=&SYSLIST(1,2,1)	= B
&P(2,2)	=&SYSLIST(1,2,2)	= (C)
&P(2,2,1)	=&SYSLIST(1,2,2,1)	= C
&P(2,2,2)	=&SYSLIST(1,2,2,2)	=null
&P(3)	=&SYSLIST(1,3)	= D
N'&P(2,2)	=N'&SYSLIST(1,2,2)	=1
N'&P(2)	=N'&SYSLIST(1,2)	=2
N'&P(3)	=N'&SYSLIST(1,3)	=1
N'&P	=N'&SYSLIST(1)	=3

PASSING SUBLISTS TO INNER MACRO INSTRUCTIONS: You can pass a suboperand of an outer macro instruction sublist as a sublist to an inner macro instruction.

K5 - Values in Operands

Specifications

EQUAL SIGN IN POSITIONAL OPERANDS: The assembler issues a warning message for a positional operand containing an equal sign, if the operand resembles a keyword operand. Thus, if we assume that the following is the prototype of a macro definition:

```
MAC1        &F
```

the following macro instruction would generate a warning message:

```
MAC1        K=L      (K is a valid keyword)
```

while the following macro instruction would not:

```
MAC1        2+2=4    (2+2 is not a valid keyword)
```

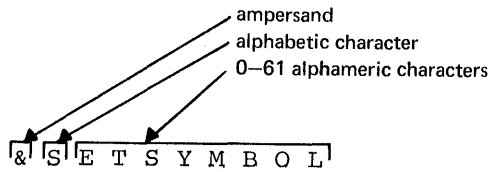
Section L: The Conditional Assembly Language

L1 – Elements and Functions

L1A -- SET SYMBOLS

Specifications

FORMAT: The format of a SET symbol is:



examples:
```
&ARITHMETICVALUE439
&BOOLEANFORCOMPARISON37
&CHARACTERSTRINGFORA35
```

IMPLICIT DECLARATION: Local SET symbols need not be declared by explicit declarations. The assembler considers any undeclared variable symbol found in the name field of a SETx instruction as a local SET symbol.

Subscripted SET Symbols -- Specifications

DIMENSION OF SUBSCRIPTED SET SYMBOLS: The dimension (the maximum value of the subscript) of a subscripted SET symbol is not determined by the explicit or implicit declaration of the symbol. The dimension specified can be exceeded in subsequent SETx instructions.

Created SET Symbols

Assembler H can create SET symbols during conditional assembly processing from other variable symbols and character strings. A SET symbol thus created has the form &(e), where "e" represents one or more of the following:

- Variable symbols, optionally subscripted.
- Strings of alphameric characters.
- Other created SET symbols.

After substitution and concatenation "e" must consist of a string of up to 62 alphameric characters, the first of which is alphabetic. The assembler will consider the preceding ampersand and this string as the name of a SET variable.

You can use created SET symbols wherever ordinary SET symbols are permitted, including declarations. You can also nest them in other created SET symbols. Consider the following example:

Name	Operation	Operand
&ABC(1)	SETC	'MKT','27','$5'

Let &(e) equal &(&ABC(&I)QUA&I).

&I	&ABC(&I)	Created SET Symbol	Comment
1	MKT	&MKTQUA1	Valid
2	27	&27QUA2	Invalid: first character after & not alphabetic
3	$5	&$5QUA3	Valid
4		&QUA4	Valid

The created SET symbol can be thought of as a form of indirect addressing. With nested created SET symbols, you can get this kind of indirect addressing to any level.

In another sense, created SET symbols offer an associative memory facility. For example, a symbol table of numeric attributes can be referenced by an expression of the form &(&SYM)(&I) to yield the "Ith" attribute of the symbol name in &SYM.

Created SET symbols also enable you to get some of the effect of multiply-dimensioned arrays by creating a separate name for each element of the array. For example, a three-dimensional array of the form &X(&I,&J,&K) could be addressed as &(X&I.$&J.$&K). Thus "&X(2,3,4)" would be represented by &X2$3$4. The "$"s guarantee that &X(2,33,55) and &X(23,35,5) are unique:

&X(2,33,55) becomes &X2$33$55
&X(23,35,5) becomes &X23$35$5

L1B -- DATA ATTRIBUTES

What Attributes Are

NUMBER OF ATTRIBUTES AVAILABLE: Under the Assembler H there are seven attributes available. In addition to the six attributes available under

the OS/VS Assembler, the Assembler H also supports the defined (D') attribute described below.

Purpose

THE DEFINED ATTRIBUTE: The defined attribute determines whether a symbol has been defined prior to the point where the attribute reference is coded.

Specifications

COMBINATION WITH SYMBOLS: The following table shows the seven kinds of attributes, identifying the types of symbols they can be combined with.

	ATTRIBUTES SPECIFIED						
SYMBOLS SPECIFIED	Type T'	Length L'	Scaling S'	Integer I'	Count K'	Number N'	Defined D'
IN THE OPEN CODE Ordinary Symbols	YES	YES	YES	YES	NO	NO	YES
SET Symbols	YES	SETC only	SETC only	SETC only	YES	YES subscripted	SETC only
System Variable Symbols: &SYSPARM &SYSDATE &SYSTIME	YES	NO	NO	NO	YES	NO	NO
IN MACRO DEFINITIONS Ordinary Symbols	YES	YES	YES	YES	NO	NO	YES
SET Symbols	YES	SETC only	SETC only	SETC only	YES	YES subscripted	SETC only
Symbolic Parameters	YES	YES	YES	YES	YES	YES	YES
System Variable Symbols: &SYSLIST	YES	YES	YES	YES	YES	YES	YES
&SYSECT, &SYSLOC, &SYSNDX, &SYSPARM, &SYSDATE, &SYSTIME	YES	NO	NO	NO	YES	NO	NO

REFERENCES TO GENERATED STATEMENTS: You can reference instructions generated by conditional assembly substitution or macro expansion with

attributes. However, no such reference can be made until the instruction is generated.

THE TYPE ATTRIBUTE (T'): Because Assembler H allows attribute references to statements generated through substitution, certain cases where a type attribute of U (Undefined) or M (Macro) is given under the OS/VS Assembler, may give a valid type attribute under Assembler H. Assembler H allows you to use the type attribute with a SETC symbol if the value of the SETC symbol is equal to the name of an instruction that can be referenced by the type attribute.

THE LENGTH ATTRIBUTE (L'): Assembler H allows you to use the length attribute with a SETC symbol if the value of the SETC symbol is equal to the name of an instruction that can be referenced by the length attribute.

THE SCALING ATTRIBUTE (S'): Assembler H allows you to use the scaling attribute with a SETC symbol if the value of the SETC symbol is equal to the name of an instruction that can be referenced by the scaling attribute.

THE INTEGER ATTRIBUTE (I'): Assembler H allows you to use the integer attribute with a SETC symbol, if the value of the SETC symbol is equal to the name of an instruction that can be referenced by the integer attribute.

THE NUMBER ATTRIBUTE (N'): The number attribute, when applied to a subscripted SET symbol, is equal to the highest element to which a value has been assigned in a SETx instruction.

For example, if the only references to &A have been

```
        LCLA        &A(100)
&A(5)   SETA        20,,,70              (See description of extended
        AIF         (&A(20) GT 50).M     SET statements)
```

then N'&A is equal to 8, because &A(8) is assigned the value 70.

THE DEFINED ATTRIBUTE (D'): The defined attribute indicates whether or not the symbol referenced has been defined prior to the attribute reference. A symbol is considered as defined if it has been encountered in the operand field of an EXTRN or WXTRN statement or in the name field of any other statement. The value of the defined attribute is 1 if the symbol has been defined or 0 if the symbol has not been defined.

The defined attribute can reference all symbols that can be referenced by the scaling (S') attribute.

The following is an example of how you can use the defined attribute:

```
            AIF     (D'A).AROUND
A           LA      1,4
.AROUND     ANOP
```

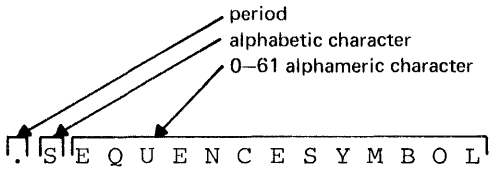
In this example, the statement at A would be assembled, since the branch around it would not be taken. However, if by a branch, the same statement were processed again, the statement at A would not be assembled:

```
.UP          AIF     (D'A).AROUND
A            LA      1,4
.AROUND      ANOP
             .
             AGO      .UP
```

You can save assembly time using the defined attribute. Each time the assembler finds a reference (attribute or branch) to an undefined symbol, it initiates a forward scan until it finds that symbol or reaches the END statement. You can use the defined attribute in your program to prevent the assembler from making this time-consuming forward scan.

L1C -- SEQUENCE SYMBOLS

FORMAT: The format of a sequence symbol is:



example: .BRANCHINGLABEL1

ATTRIBUTE DEFINITION AND LOOKAHEAD

Symbol attributes are established in either definition mode or lookahead mode. Lookahead mode is entered when Assembler H encounters an attribute reference to a symbol that is not yet defined.

Definition Mode

Definition occurs whenever a previously undefined symbol is encountered in the name field of a statement, or in the operand field of an EXTRN or WXTRN statement during open code processing. Symbols within a macro definition are defined when the macro is generated.

Lookahead Mode

Lookahead is a sequential, statement-by-statement, forward scan over the source text. It is initiated when reference is made to an attribute (other than D') of a symbol not yet encountered, either by macro or open-code attribute reference, or by a forward AGO or AIF branch in open code.

If reference is made in a macro, forward scan begins with the first source statement following the outermost macro instruction. Programmer macros are bypassed. The text is not assembled. Lookahead attributes are tentatively established for all intervening undefined symbols. Tentative attributes are replaced and fixed when the symbol is subsequently encountered in definition mode. No macro expansion or open-code substitution is performed; no conditional or unconditional (AIF or AGO) branches are taken. COPY instructions are excuted during lookahead, and the copied statements are scanned.

Lookahead ends when the desired symbol or sequence symbol is found, or when the END card or end of file is reached. All statements passed over by lookahead are saved on an internal file and processed when the lookahead ends.

For purposes of attribute definition, a symbol is considered undefined if it depends in any way upon a symbol not yet defined. For example, if the symbol is defined by a forward EQU that is not yet resolved, or if a DC, DS, or DXD modifier expression contains symbols not yet defined, that symbol is assigned a type attribute of U.

NOTE: Since no variable symbol substitution is performed by a lookahead, you should be careful when using a macro or open code substitution to generate END statements that separate source modules assembled in one job step (option BATCH). If a symbol is undefined within a module, lookahead will read in records past the point where the END statement is to be generated. All statements between the generated statement and the point where lookahead stops (either because it finds a matching symbol, or because it finds an END statement) are ignored by the assembler. The next module will start at the point where lookahead stops.

Lookahead Restrictions

Assembler statements are analyzed only to the extent necessary to establish attributes of symbols in their name fields.

Variable symbols are not replaced. Modifier expressions are evaluated only if all symbols involved were defined prior to lookahead, Possible multiple or inconsistent definition of the same symbol is not diagnosed during lookahead because conditional assembly may eliminate one (or both) of the definitions.

Lookahead does not check undefined op codes against library (system) macro names. If the name field contains an ordinary symbol and the op code cannot be matched with one in the current op code table, then the ordinary symbol is assigned the type attribute of M. If the op code contains special characters or is a variable symbol, a type attribute of U is assumed. This may be wrong if the undefined op code is later defined by OPSYN. OPSYN statements are not processed; thus, labels are treated in accordance with the op code definitions in effect at the time of entry to lookahead.

L2 -- Declaring SET Symbols

IMPLICIT DECLARATION: Local SET symbols need not be declared explicitly with LCLA, LCLB, or LCLC statements. The assembler considers any undeclared variable symbol found in the name field of a SETA, SETB, or SETC statement to be a local SET symbol. It is given the initial value specified in the operand field. If the symbol in the name field is subscripted, it is declared as a subscripted SET symbol.

L2A -- THE LCLA, LCLB, AND LCLC INSTRUCTIONS

Specifications

MULTIPLE DECLARATIONS OF SET SYMBOLS: Multiple LCLx statements can declare the same variable symbol if only one declaration for a given symbol is encountered during the expansion of a macro.

MAXIMUM VALUE OF SET SYMBOL SUBSCRIPT: There is no limit to SET symbol dimensioning. The limit specified in the explicit (LCLx) or implicit (SETx) declaration can also be exceeded, by subsequent SETx statements.

ALTERNATE FORMAT FOR LCLX STATEMENTS: The Assembler H permits the alternate statement format for LCLx instructions. An example is:

```
Statement Field                                    Continuation Indicator
   LCLA      &LOCALSYMBOLFORDCGEN,                          X
             &COUNTERFORINNERLOOP,                          X
             &COUNTERFOROUTERLOOP,                          X
             &COUNTERFORTRAILINGLOOP
```

L2B -- THE GBLA, GBLB, AND GBLC INSTRUCTIONS

Specifications

MULTIPLE DECLARATIONS OF SET SYMBOLS: Multiple GBLx statements can declare the same variable symbol if only one declaration for a given symbol is encountered during the expansion of a macro.

MAXIMUM VALUE OF SET SYMBOL SUBSCRIPT: There is no limit on the maximum subscript allowed. Also, the limit specified in the global declaration (GBLx) can be exceeded.

ALTERNATE FORMAT FOR GBLX STATEMENTS: The Assembler H permits the alternate statement format for GBLx instructions. An example is:

```
Statement Field                                    Continuation Indicator
   GBLA      &GLOBALSYMBOLFORDCGEN,                         X
             &LOOPCONTRLA,                                  X
             &VALUEPASSEDTOMACDUFF,                         X
             &VALUERETURNEDFROMMACDUFF
```

L3 -- Assigning Values to SET Symbols

Extended SET Statements

You can assign values to multiple elements in an array of a subscripted SET symbol with one single SETx instruction. Such an instruction is called an extended SET statement.

The format of an extended SETx statement is:

Name	Operation	Operand
A subscripted variable symbol	SETA SETB SETC	operand1,operand2, operand3,....,operandn

The name field specifies the name of the SET symbol and the position in the array to which the first value in the operand field is to be assigned. The successive operand values are then assigned to the successive positions in the array. If an operand is omitted, the corresponding element of the array is unchanged. Consider the following example:

Name	Operation	Operand
	LCLA	&LIST(50)
&LIST(11)	SETA	5,10,,20,25,30

The first instruction declares &LIST as a subscripted local SETA symbol. The second instruction assigns values to certain elements of the array &LIST:

Number of Element	10	11	12	13	14	15	16	
not assigned	*not assigned*	5	10	*not assigned*	20	25	30	*not assigned*

Thus the instruction does the same as the following sequence:

Name	Operation	Operand
&LIST(11)	SETA	5
&LIST(12)	SETA	10
&LIST(14)	SETA	20
&LIST(15)	SETA	25
&LIST(16)	SETA	30

ALTERNATE STATEMENT FORMAT: You can use the alternate statement format for extended SETx statements. The above coding could then be written as follows:

```
Name       Operation   Operand    Remarks          Continuation
                                                    Indicator
&LIST(11)  SETA        5,         THIS IS             X
                       10,,       AN ARRAY            X
                       20,25,30   SPECIFICATION
```

L4 -- Using Expressions

L4A -- ARITHMETIC (SETA) EXPRESSIONS

SETC VARIABLES IN ARITHMETIC EXPRESSIONS: The Assembler H permits a SETC variable to be used as a term in an arithmetic expression if the character string value of the variable is a self-defining term. The value represented by the string is assigned to the arithmetic term. A null string is treated as zero. The OS/VS Assembler allows SETC variables as arithmetic terms only if the value of the variable is a decimal self-defining term, not longer than ten characters.

Examples:

```
┌──────────────────┬──────────┬──────────────────────────────┐
│       Name       │ Operation│           Operand            │
├──────────────────┼──────────┼──────────────────────────────┤
│                  │LCLC      │&C(5)                         │
│&C(1)             │SETC      │'B''101'''                    │
│&C(2)             │SETC      │'C''A'''                      │
│&C(3)             │SETC      │'23'                          │
│&A                │SETA      │&C(1)+&C(2)-&C(3) [1]         │
│&AA               │SETA      │&C(3) [2]                     │
└──────────────────┴──────────┴──────────────────────────────┘
```

[1] (Allowed only by Assembler H)
[2] (Allowed by the OS/VS Assembler and Assembler H)

In evaluating the arithmetic expression in the fifth statement, the first term (&C(1)) is assigned the binary value 101 (5). To that is added the value represented by the EBCDIC character A (hexadecimal C1 which corresponds to decimal 193). Then the value represented by the third term (&C(3)) is subtracted, and the value of &A becomes 5+193-23=175.

This feature allows you to associate numeric values with EBCDIC or hexadecimal characters to be used in such applications as indexing, code conversion, translation, and sorting.

Assume that &X is a character string with the value ABC.

```
┌──────────────────┬──────────┬──────────────────────────────┐
│       Name       │ Operation│           Operand            │
├──────────────────┼──────────┼──────────────────────────────┤
│&I                │SETC      │'C'''.'&X'(1,1).''''          │
│&VAL              │SETA      │&TRANS(&I)                    │
└──────────────────┴──────────┴──────────────────────────────┘
```

The first statement sets &I to C'A'. The second statement extracts the 193rd element of &TRANS (C'A' = X'C1' = 193).

The following code will convert a hex value in &H into a decimal value in &VAL:

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│&X                │SETC      │'X''&H'''                 │
│&VAL              │SETA      │&X                        │
└──────────────────┴──────────┴──────────────────────────┘
```

An arithmetic expression must not contain two terms in succession; however, any term may be preceded by any number of unary operators. +&A*-&B is a valid operand for a SETA instruction. The expression &FIELD+- is invalid because it has no final term.

MAXIMUM SIZE: The maximum number of levels of parentheses allowed in an arithmetic expression is 255.

L4C -- LOGICAL (SETB) EXPRESSIONS

LIMITATION TO LOGICAL OPERATORS: There is no limit to the level of parentheses allowed in logical expressions.

L6 - Branching

L6A -- THE AIF INSTRUCTION

Extended AIF Instruction

The extended AIF instruction allows you to combine several successive AIF statements into one statement. The extended AIF instruction has the following format:

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│A sequence        │AIF       │(logical expression).S1,  │
│symbol or         │          │(logical  expression).S2, │
│blank             │          │...,(logical  expression).Sn│
│                  │          │                          │
└──────────────────┴──────────┴──────────────────────────┘
```

It is exactly equivalent to n successive AIF statements. The branch is taken to the first sequence symbol (scanning left to right) whose corresponding logical expression is true. If none of the logical expressions is true, no branch is taken.

Consider the following example:

Operation	Operand	Col.72
AIF	('&L'(&C,1) EQ '$').DOLR,('&L'(&C,1) EQ ' # ').POUND, ('&L'(&C,1) EQ '@').AT,('&L'(&C,1) EQ '=').EQUAL, ('&L'(&C,1) EQ '(').LEFTPAR,('&L'(&C,1) EQ '+').PLUS, ('&L'(&C,1) EQ '-').MINUS	X X X

This statement looks for the occurrence of a $, #, @, =, (, +, and -, in that order; and causes control to branch to .DOLR, .POUND, .AT, .EQUAL, .LEFTPAR, .PLUS, and .MINUS, respectively, if the string being examined contains any of these characters.

THE ALTERNATE STATEMENT FORMAT: The alternate statement format is allowed for extended AIF instructions. The format is illustrated by the example above.

L6B -- THE AGO INSTRUCTION

Computed AGO Instruction

The computed AGO instruction allows you to make branches according to the value of an arithmetic expression specified in the operand. The format of the computed AGO instruction is as follows:

Name	Operation	Operand
A sequence symbol or blank	AGO	(arithmetic expression).S1,.S2,...,.Sn

If the arithmetic expression evaluates to k, where k lies between 1 and n (inclusive), then the branch is taken to the "k-th" sequence symbol in the list. If k is outside that range, no branch is taken.

In the following example

Name	Operation	Operand
	AGO	(&I).FIRST,.SECOND,.THIRD,.FOURTH

control passes to the statement at .THIRD if &I=3. Control passes through to the statement following the AGO if &I is less than 1 or greater than 4.

THE ALTERNATE STATEMENT FORMAT: The alternate statement format is allowed for computed AGO instructions. The example above could be coded:

Statement Field		Continuation Indicator
AGO	(&I).FIRST,	X
	.SECOND,	X
	.THIRD,	X
	.FOURTH	

L9 - Macro Trace Facility

L9A -- THE MHELP INSTRUCTION

Purpose

The MHELP instruction controls a set of trace and dump facilities. Options are selected by an absolute expression in the MHELP operand field. MHELP statements can occur anywhere in open code or in macro definitions. MHELP options remain in effect until superseded by another MHELP statement.

Specifications

The format of this instruction is as follows:

Name	Operation	Operand
A sequence symbol or blank	MHELP	Absolute expression, binary or decimal options as discussed below.

Macro Call Trace -- Operand=1

This option provides a one-line trace listing for each macro call, giving the name of the called macro, its nested depth, and its &SYSNDX value. The trace is provided only upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

Macro Branch Trace -- Operand=2

This option provides a one-line trace listing for each AGO and true AIF conditional-assembly statement within a macro. It gives the model-statement numbers of the "branched from" and the "branched to"

statements, and the name of the macro in which the branch occurs. This trace option is suppressed for library macros.

Macro AIF Dump -- Operand=4

This option dumps undimensioned SET symbol values from the macro dictionary immediately before each AIF statement that is encountered.

Macro Exit Dump -- Operand=8

This option dumps undimensioned SET symbols from the macro dictionary upon entering a MEND or a MEXIT statement.

Macro Entry Dump -- Operand=16

This option dumps parameter values from the macro dictionary immediately after a macro call is processed.

Global Suppression -- Operand=32

This option suppresses global SET symbols in two preceding options, MHELP 4 and MHELP 8.

MHELP Suppression -- Operand=128

This option suppresses all currently active MHELP options.

MHELP Control On &SYSNDX

The MHELP operand field is actually mapped into a fullword. Previously-defined MHELP codes correspond to the fourth byte of this fullword.

&SYSNDX control is turned on by any bit in the third byte (operand values 256-65535 inclusive). Then, when &SYSNDX (total number of macro calls) exceeds the value of the fullword which contains the MHELP operand value, control is forced to stay at the open-code level, by in effect making every statement in a macro behave like a MEXIT. Open code macro calls are honored, but with an immediate exit back to open code.

Examples:

```
MHELP 256        Limit &SYSNDX to 256.
MHELP 1          Trace macro calls.
MHELP 256+1      Trace calls and limit &SYSNDX to 257.
MHELP 65536      No effect. No bits in bytes 3,4.
MHELP 65792      Limit &SYSNDX to 65792.
```

When the value of &SYSNDX reaches its limit, the message "ACTR EXCEEDED -- &SYSNDX" is issued.

Combining Options

As shown in the example above, multiple options can be obtained by combining the option codes in one MHELP operand. For example, call and branch traces can be invoked by MHELP B'11', MHELP 2+1, or MHELP 3. Substitution by means of variable symbols may also be used.

Index

&

A

C

D

E

N

O

P

Q

R

S

T

V

Part II: For the MFT or MVT User

Preface

This part presents reference information about Assembler H. Specific information contained in the following pages supersedes the corresponding information for Assembler F that appears in the OS Assembler Language, Order Number GC28-6514. Except where contradicted by this part, the information in GC28-6514 applies to Assembler H.

Assembler H operates under the control of OS/MFT or OS/MVT. Assembler H supports the features of lower level operating system assemblers. Any program successfully assembled with no warning or diagnostic message by a lower level OS assembler will assemble correctly under Assembler H.

Knowledge of the IBM System/360 and 370 machine operations, particularly storage addressing, data formats, and machine instruction formats and functions, is prerequisite to using this publication. So is experience with programming concepts and techniques or completion of basic courses of instruction in these areas. IBM System/360 and 370 machine operations are discussed in IBM System/360 Principles of Operation, Order Number GA22-6821, and in System/370 Principles of Operation, Order Number GA22-7000.

You are also assumed to be generally familiar with assembler language and with macro and conditional-assembly processing. Such information may be found in the following manual, which will be referred to in this manual as "the Assembler Language manual."

OS Assembler Language, Order Number GC28-6514

This part is a supplement to the Assembler Language manual. To have complete documentation of the Assembler H language, you need both the Assembler Language manual and this manual. Concepts introduced in the Assembler Language manual are not reworked in this manual.

The structure of this manual parallels that of the Assembler Language manual; sections of this book correspond exactly to sections of that book. For example, there is no information applicable only to Assembler H that corresponds to Section 1 of the Assembler Language manual; therefore, this manual begins with Section 2.

In the text of this manual, braces { } are used to indicate that only one of the items enclosed is to be used. Brackets [ ] indicate that all the enclosed items are optional.

Other publications containing pertinent information are the following:

OS Assembler H Programmer's Guide, Order Number SC26-3759

The Assembler H Programmer's Guide gives detailed information about programming with Assembler H, including assembler options and job control language procedures applicable to Assembler H. It also explains the listing produced by the assembler.

OS Assembler H Messages, Order Number SC26-3770

The Assembler H Messages manual provides an explanation of each of the diagnostic and abnormal termination messages issued by Assembler H and suggests how you should respond in each case.

OS Assembler H System Information, Order Number GC26-3768

The System Information manual consists of three self-contained chapters on performance estimates, storage estimates, and system generation of Assembler H.

OS Assembler H Logic, Order Number LY26-3760

The Logic manual describes the design logic and functional characteristics of Assembler H.

OS Introduction, Order Number GC28-6534

The Introduction describes and interrelates all OS control program facilities. It shows how these facilities work with the language translators and service programs, so you can better learn how to use the system.

OS Job Control Language Reference, Order Number GC28-6704

The Job Control Language book tells how to code the job control language necessary to initiate and control the processing of any program, and contains a discussion of cataloged procedures.

OS Loader and Linkage Editor, Order Number GC28-6538

The Loader and Linkage Editor manual provides information on the operation and use of the loader and linkage editor, which are two programs that prepare the output of language translators for execution.

Contents

Section 2: General Information

Assembler-Language Coding Conventions

Continuation Lines

Nine continuation lines are allowed, ten cards in all.

Statement Format

Name Entry: Symbols may be up to 63 alphanumeric characters long. External Symbol Dictionary items (contained in START, CSECT, COM, DXD, EXTRN, WXTRN, and ENTRY statements, and in Q-type and V-type constants) are limited to 8 characters. For all cases, the first character must be a letter.

Operation Entry: Op codes defined by OPSYN and macro names may be up to 63 characters long.

Terms and Expressions

Symbols

All symbols except External Symbol Dictionary items may include up to 63 characters. The first character, or the first character following a period or ampersand, must be a letter.

Ordinary Symbols may contain up to 63 alphanumeric characters. The first of these must be a letter. RECORDAREA2, for example, is a valid symbol. Special characters and blanks are not allowed.

Variable Symbols may contain up to 62 alphanumeric characters following the ampersand. The first of these must be a letter.

Sequence Symbols may contain up to 62 alphanumeric characters following the period. The first of these must be a letter.

Note that a valid expression which consists of anything more than a single term is considered by the assembler to be an arithmetic

combination of terms. For example, the following expressions are single terms:

```
ALPHA
L'BETA
*
L'*
=A(GAMMA)
12345
B'1100'
X'FFFF'
C'WXYE'
```

The following expressions are not single terms, but are arithmetic combinations of terms:

```
+ALPHA
(L'BETA)
-12345
-(ALPHA)
-(+(-BETA))
-2*(BETA-ALPHA)
```

Previously Defined Symbols

Ordinary symbols do not have to be previously defined if they appear in operand expressions of ORG and CNOP instructions, in modifier expressions of DC, DS, and DXD statements, in the first operand of EQU statements, or in Q-type constants.

Allowing forward reference in the above statement types creates two new kinds of errors which you should guard against.

- Circular definition of symbols, such as

```
X       EQU     Y
Y       EQU     X
```

- Circular location-counter dependency, as in this example:

```
A       DS      (B-A)C
B       LR      1,2
```

Statement A cannot be resolved because the value of the duplication factor is dependent on the location of B, which is in turn dependent upon the length of A.

Literals may contain symbolic expressions in modifiers, but any ordinary symbols used must have been previously defined. See Section 9, "Attribute Definition and Lookahead", in this manual.

Previous Definition of Symbols

Some coding restrictions are imposed in certain cases where expressions are allowed as operands, because the operand field must be processed when it is first encountered during Pass 1. The cases are:

Items	Affected Fields
START statements	Any operand
EQU statements	Expression 2 and Expression 3 (See in this manual, "EQU -- Equate Symbol" in section 5)
Literals	Duplication factor subfield and length modifier subfield

The value of an expression in one of the fields above can be determined in Pass 1 only if all symbols in the expression have been defined by previous statements (or the same statement). Thus, all symbols referenced in those statements must have been previously defined.

In addition, these expressions must have absolute values. Thus, if a relocatable term is used, it must be paired with another relocatable term, with the opposite sign, from the same section.

However, paired relocatable terms must be used with caution. Assembler H features may have caused temporary segmentation of the location counter between the statements which define the two terms of the relocatable pair. If this has occurred, the difference of the two terms cannot be computed and the expression cannot be evaluated, as it must be, in Pass 1.

Self-Defining Terms

All types of self-defining terms are extended to 32-bit logical values with the number of source digits as follows:

Term	Maximum Number Source Digits
Decimal	10
Hexadecimal	8
Binary	32
Character	4

Note: The value of a decimal self-defining term must lie in the range 0 through 2,147,483,647. A 32-bit hexadecimal, binary, or character self-defining term with a 1 in the sign bit is treated as a negative number when used in an arithmetic expression.

Location Counter Segmentation

In Pass 1, the location counter is temporarily interrupted (and the assignment of temporary values to relocatable symbols is restarted at zero) whenever any of the following situations is encountered:

- A new LOCTR name is defined.
- Forward reference or a segmented pair of relocatable terms is used in a DC, DS, ORG, or CNOP statement.
- The location counter cannot be aligned because of a prior segmentation.
- An ORG statement with a blank operand occurs after a prior segmentation in the same LOCTR, so that the highest previous location-counter setting in the LOCTR is not yet known.

GENERAL RESTRICTIONS ON SYMBOLS: A symbol may appear only once in the name field of a statement, except for START, CSECT, DSECT, COM, and LOCTR names.

Location Counter Reference

The assembler internally maintains the location counter value as a 32-bit value. It flags any overflow into the high-order byte as a "location counter error". When this error occurs, the location counter will continue to carry the internal value of four bytes, even though addressability is limited to three bytes. As an illustration, consider the following example:

```
LOC      OBJECT CODE     NAME      OPERATION       OPERAND
000000                   A         CSECT
000000   FFFFFE                    ORG             *+X'FFFFFE'
FFFFFE   58506004                  L               5,4(6)
   IEV039      * * *ERROR* * *LOCATION COUNTER ERROR
000002   07FF            B         BR              15
000004   01000002        C         DC              A(B)
```

Note that the location counter value of B prints as three bytes, but the location counter value for the address constant at C is four bytes long.

Location counter values for EQU and USING statements are printed as four bytes in the text portion of the listing. The values for COM, CSECT, DSECT, DXD, EQU, EXTRN, WXTRN, LOCTR, and START symbols are four-byte values in the Cross Reference Listing.

Literals

Symbolic expressions are allowed in the duplication factor and length modifier fields of literals, as well as in the scale and exponent modifier fields. Any symbols involved in the duplication factor or length modifier must have been previously defined. Thus

```
L     REG,=(A)F'6'
```

is allowed if A has been defined in the program prior to this statement.

All literals are cross referenced. Cross reference entries for symbols used in literals refer to the literal pool where the literals are located, and not to the statements that reference the literals.

Symbol Length Attribute Reference

In either of the following instructions

```
A     EQU     *
A     EQU     'expression'
```

where 'expression' consists of a self-defining term or begins with a self-defining term, Assembler H sets the length attribute of A to 1. In any EQU statement, Assembler H uses the value of a second operand, if specified, as the length attribute of the symbol in the name field. Thus in the statement

```
A     EQU     *,5
```

the length attribute of A is 5. See, in Section 5 of this manual, "EQU -- Equate Symbol", and, in Section 9, "Attribute Definition and Lookahead".

Terms in Parentheses

There is no restriction on the number of terms or levels of parentheses in an expression.

EXPRESSIONS

Assembler H allows unary operators, as well as binary operators, in expressions.

Unary operators affect the algebraic sign of a single term or expression; these are some examples:

```
-(L'FIELD)
+(BETA-ALPHA)
-5
```

A binary operator indicates an operational relationship between two terms such as

```
GAMMA+(BETA-ALPHA)
13+(-7)
```

A unary operator can precede a term, a left parenthesis, or another unary operator; it can begin an expression or follow a left parenthesis, unary operator, or binary operator.

Thus the rules for coding expressions with Assembler H are:

- Unary (+, -,) and binary (+, -, /, *) arithmetic operators are allowed in expressions.
- An expression may start with a unary operator. Thus -A, -X+5, and +5 are all valid expressions.
- An expression cannot contain two terms or two binary operators in succession.
- An arithmetic expression cannot contain a literal.

Evaluation of Expressions

Arithmetic operators + or - between terms are evaluated as follows:

Expression	Evaluation
A+B	B is added to A
A+-B	The negative of B is added to A (A+(-B)

Thus it is possible to have any number of plus and minus signs in succession between terms. The leftmost (or only) + or - between the terms is treated by the assembler as a binary operator. Any arithmetic operators to the immediate right of the first are treated as unary operators.

The rules for evaluation of expressions given for Assembler F in Section 2 of the Assembler Language manual are valid for Assembler H except that:

- Expressions are not truncated to the rightmost 24 bits.
- Unary operations are performed before binary operations. The sequence of binary operations is the same: multiplication and division are performed before addition and subtraction.

The final and intermediate results of expression evaluation must lie in the range -2^{31} through $2^{31}-1$.

Relocatable Expressions: A simply relocatable expression may have a negative value, but the unpaired relocatable term must not be algebraically subtracted, or else the expression will be complexly relocatable.

Section 3: Addressing -- Program Sectioning and Linking

BASE REGISTER INSTRUCTIONS

Addresses are broken down into base and displacement form by means of the USING statement(s) currently in effect. The assembler keeps 32-bit positive or negative values for USING statements and prints them in the assembly listing. The current USING may be saved by means of the PUSH assembler operation and restored later with a corresponding POP operation. (See "PUSH -- Save Current USING or PRINT Status" and "POP -- Restore USING or PRINT Status", in Section 5 of this manual.

USING -- USE BASE ADDRESS REGISTER

A 32-bit signed value corresponding to the first operand of a USING statement is saved and used in subsequent base-displacement computation. The value of the first operand is printed as a four-byte value in the location counter field.

The current USING status may be saved with an assembler operation, PUSH, and restored later with a corresponding operation, POP. (See in this book Section 5, "Listing Control Instructions", "PUSH", "POP".)

DROP -- DROP BASE REGISTER

The operand field of a DROP instruction may be blank. DROP with a blank operand field causes all currently active USING registers to be dropped.

PROGRAM SECTIONING AND LINKING

The total number of control sections, dummy sections, external symbols, entry symbols, external dummy sections, LOCTR instructions, and common sections may not exceed $2^{16}-1$. Note, however, that the lower limits imposed by the Linkage Editor and Loader are in effect. See the OS Loader and Linkage Editor manual for details.

Control Section Location Assignment

CSECTs and LOCTRs can be intermixed. The assembler provides a location counter for each. Each LOCTR after the first in a control section begins at the next available byte; that is, the instruction does not force location counter alignment. Invalid section names of any type default to blank names. See "LOCTR--Define Location Counter" later in this section.

START -- Start Assembly

The START instruction is allowed after DXD statements and after DSECT and COM sections. The operand of the START instruction may be an expression, but any symbols referenced must have been previously defined. Location counter reference may not be used.

CSECT -- Identify Control Section

Each control section or LOCTR instruction initiates a different location counter or continues an interrupted one. All statements following a control section or LOCTR instruction are assembled within that location counter, until a different location counter is produced by another control section or LOCTR. Resuming an interrupted control section forces resumption of the most recently active LOCTR in that section. If no LOCTR instruction has been specified, the location counter initiated by that control section is continued. See "LOCTR--Define Location Counter" later in this section.

DSECT -- Identify Dummy Section

The format of the DSECT instruction is:

Name	Operation	Operand
Any symbol or blank	DSECT	Blank

A DSECT instruction may be given a blank name. If more than one dummy section is defined, all but one of them must be named because the blank name field effectively "names" the dummy section.

Note: A symbol defined in a dummy section may be used in an A-type or Y-type constant which generates RLD items only if the symbol is paired

with another symbol (with the opposite sign) from the same dummy section.

RLD items are generated by A-, Y-, V- or Q-type address constants only in a DC operand defined within a CSECT and without a zero duplication factor. Both named and unnamed dummy sections may be further segmented by using the LOCTR assembler operation.

DXD--DEFINE EXTERNAL DUMMY SECTION

DXD will not generate an ESD item unless the DXD is referenced by a Q-type constant that is a control section begun by CSECT or START (or in a LOCTR that continues a control section begun by CSECT or START).

The operand format and alignment is identical to that of the DS instruction. See "DS -- Define Storage" in Section 5 of this manual.

Note: A symbol that names a DXD statement may be used in A- or Y-type constants which generate RLD items only if the symbol is paired with the same symbol with the opposite sign, as in this example:

Name	Operation	Operand
JOE	DXD	F
	DC	A(JOE+ALPHA-JOE)

COM -- DEFINE COMMON CONTROL SECTION

Assembler H allows named, as well as unnamed, COM instructions. The format of the COM statement is as follows:

Name	Operation	Operand
Any symbol or blank	COM	Blank

As with Assembler F, instructions or data appearing in a common control section are never assembled.

Both named and unnamed common control sections may be further segmented using the LOCTR assembler operation.

LOCTR -- DEFINE LOCATION COUNTER

The LOCTR instruction allows you to define multiple location counters within a control section.

The format of the LOCTR instruction is:

Name	Operation	Operand
A variable symbol or ordinary symbol	LOCTR	Blank

A control section name automatically names the first location counter in that section. A LOCTR instruction with the same name as another location counter definition forces a return to that location counter, as well as to the control section in which it was defined.

The assembler will assign consecutive addresses to the statements assembled under each new or resumed location counter. A control section name is in error if it is identical to a previously defined LOCTR name. The length attribute of a LOCTR name is 1.

A location counter may be interrupted by a CSECT, a DSECT, a COM, or another LOCTR. A LOCTR which occurs before the first control section will initiate an unnamed control section (private code) before the LOCTR is processed. LOCTR does not force location counter alignment.

This instruction enables you to code logically, defining data for instance, where and when the need arises. Figure 1 illustrates location counter definition.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

000000                              1 A        CSECT                    INITIALIZATION                00060000
000004                              2 B        LOCTR                    MAIN ROUTINE                  00070000
000008                              3 C        LOCTR                    COMPLETION                    00080000
00000E                              4 D        LOCTR                    CONSTANTS                     00090000
00000000                            5          USING A,15               ESTABLISH ADDRESSABILITY      00100000
000004                              6 B        LOCTR                    BEGIN WITH MAIN ROUTINE       00110000
000004 5850 F010            00010   7          L     5,CON20                                        00120000
00000E                              8 D        LOCTR                    CONSTANT LOCATION COUNTER     00130000
00000E 0000
000010 00000014                     9 CON20    DC    F'20'              DEFINE THE CONSTANT           00140000
000008                             10 B        LOCTR                    BACK TO THE MAIN ROUTINE      00150000
                                   11 *        ...                                                    00160000
                                   12 *        ...                                                    00171000
                                   13 *   BALANCE OF THE MAIN ROUTINE                                 00180000
                                   14 *        ...                                                    00190000
                                   15 *        ...                                                    00200000
000000                             16 A        LOCTR                    INITIALIZATION COUNTER        00210000
                                   17          SAVE  (14,12)            INITIALIZATION BEGUN          00220000
000000                             18+         DS    0H                                               01-SAVE
000000 90EC D00C            0000C  19+         STM   14,12,12(13)             SAVE REGISTERS          01-SAVE
000008                             20 C        LOCTR                    COMPLETION COUNTER            00230000
                                   21          RETURN (14,12)           COMPLETION BEGUN              00231000
000008 98EC D00C            0000C  22+         LM    14,12,12(13)             RESTORE THE REGISTERS   01-RETUR
00000C 07FE                        23+         BR    14                       RETURN                  01-RETUR
                                   24          END                                                    00232000
```

Figure 1. Location Counter Definition

In Figure 1, all statements coded for a particular location counter are assigned adjacent addresses in storage.

Symbolic Linkages: There is no practical restriction on the total number of control sections, dummy sections, external symbols, ENTRY items, and external dummy sections. The effective maximum for each of these items depends upon main storage available at linkage edit time, and thus may vary from program to program.

EXTENDED MNEMONIC CODES

The following extended mnemonics provide RR format instructions which correspond to the existing RX format extended mnemonic branch instructions.

Extended Code	Meaning *	Machine Instructions
BOR R2	Branch on Overflow (Ones)	BCR 1,R2
BHR R2	Branch on High	BCR 2,R2
BPR R2	Branch on Plus	BCR 2,R2
BLR R2	Branch on Low	BCR 4,R2
BMR R2	Branch on Minus (Mixed)	BCR 4,R2
BNER R2	Branch on Not Equal	BCR 7,R2
BNZR R2	Branch on Not Zero	BCR 7,R2
BER R2	Branch on Equal	BCR 8,R2
BZR R2	Branch on Zero	BCR 8,R2
BNLR R2	Branch on Not Low	BCR 11,R2
BNMR R2	Branch on Not Minus (Mixed)	BCR 11,R2
BNHR R2	Branch on Not High	BCR 13,R2
BNPR R2	Branch on Not Plus	BCR 13,R2
BNOR R2	Branch on Not Overflow (Ones)	BCR 14,R2

*All instructions are RR format

Table of New Instructions

The instructions are all privileged. For a description, see the latest edition of the IBM System/370 Principles of Operation, Order No. GA22-7000. The machine instruction formats are given in the latest edition of Assembler Language, Order No. GC28-6514, Appendix C.

Instruction	Mnemonic	Machine op code	Type of instruction	Literal allowed (Y=yes, N=no)	Storage alignment[1]
LOAD REAL ADDRESS	LRA	B1	RX	Y	1
PURGE TLB[2]	PTLB	B20D	S	-	-
RESET REFERENCE BIT	RRB	B213	S	N	1
SET CLOCK COMPARATOR	SCKC	B206	S	Y	8
SET CPU TIMER	SPT	B208	S	Y	8
STORE CLOCK COMPARATOR	STCKC	B207	S	N	8
STORE THEN AND SYSTEM MASK	STNSM	AC	SI	N	1
STORE THEN OR SYSTEM MASK	STOSM	AD	SI	N	1
STORE CPU TIMER	STPT	B209	SI	N	8

[1]Storage alignment for the storage operand is required. '8' means alignment to double word.

[2]No operand: if specified is ignored.

Section 5: Assembler Instruction Statements

These two listing control statements are unique to Assembler H:

PUSH -- Save current USING or PRINT status

POP -- Restore current USING or PRINT status

These are discussed later in this section to conform to the structure of the Assembler Language manual.

Symbol Definition Instructions

EQU -- EQUATE SYMBOL

The format of the equate instruction is:

Name	Operation	Operand
A variable symbol or ordinary symbol	EQU	Expression1 [{ ,Expression2 [,Expression3] / ,,Expression3 }]

Expression 1 must be present and may assume any value allowed for assembly expressions: absolute (including negative), relocatable, or complexly relocatable. (See the OS Assembler Language manual and this book: "Complex Relocatable Expressions" under "Operand Subfield 4: Constant", in Section 5). Previous definition is not required for symbols appearing in Expression 1.

If the expression is complexly relocatable, the value attribute is meaningless. During the evaluation of any expression that includes a complexly relocatable symbol, the symbol is replaced by its own defining expression. Thus if A1 and A2 belong to one control section and B1 and B2 belong to another, and if these equate statements have been used:

Name	Operation	Operand
X	EQU	A1+B1
	.	
	.	
	.	
Y	EQU	X-A2-B2

then X is complexly relocatable but Y is absolute.

Expression 2 and Expression 3 are both optional and any symbols appearing in their operands must have been previously defined. Both expressions, if present, must be absolute.

The value of Expression 2 must be in the range 0 through 65535. The value of Expression 2 overrides the normal length attribute of the symbol in the name field. Two commas following Expression 1 indicate the absence of Expression 2 and the presence of Expression 3. If Expression 2 is omitted, the length attribute of the symbol in the name field is based on the leftmost (or only) term of Expression 1:

- If it is an asterisk (which references the current value of the current location counter) or a self-defining term, the length attribute is 1;
- If it is the name of a DC or DS instruction, the length attribute equals the implicit or explicit length of the first constant.
- If it is the name of a machine instruction or channel command word, the length attribute is the length of the instruction itself.

The value of Expression 3 must be in the range 0 through 255. The value of Expression 3 overrides the normal type attribute of the symbol in the name field. The corresponding EBCDIC character is taken as the type attribute.

For example,

Name	Operation	Operand
A	EQU	SYM,13,194

The value of A is that of SYM, the length attribute is 13, and the type attribute is the letter B. The same type attribute could be achieved by writing C'B', X'C2', or B'11000010' as the third operand. If Expression 3 is not present, the type attribute defaults to "U".

OPSYN -- EQUATE OPERATION CODE

The OPSYN instruction is used, as with Assembler F, to redefine or delete existing operation codes or to define new operation codes equivalent to existing ones. The modified format is:

Name	Operation	Operand
An ordinary symbol or variable symbol	OPSYN	A machine instruction mnemonic code, an extended mnemonic code, a macro operation, an assembler operation, an operation code defined by a previous OPSYN instruction, or a blank.

The symbol in the name field may be a previously defined operation code. The latest definition takes precedence. The OPSYN mechanism does not search the macro library for possible system macro definitions. Thus:

Name	Operation	Operand
PDQ	OPSYN	DCB

will yield the desired results if DCB has been expanded or edited (as in COPY DCB) prior to this OPSYN. Otherwise, DCB will be flagged as an undefined opcode, and PDQ will be flagged when used later on.

If you want to prevent the assembler from recognizing the operation code in the name field, use the following format:

Name	Operation	Operand
A mnemonic operation code	OPSYN	Blank

OPSYN symbols conflict in no way with the use of the same symbols as names on other statements.

OPSYN statements are ignored by lookahead. You should therefore especially beware of an OPSYN involving COPY within a lookahead request, or an OPSYN involving END. If such OPSYNs are used, they should be placed early in the program before any statement that may initiate lookahead mode.

Data Definition Instructions

Literal Definitions: The following rules differentiate the coding of literals from the coding of constants:

- The literal is preceded by an equal sign.
- Multiple operands may not be specified.
- Every symbol in an expression used to specify the duplication factor or length modifier must have been defined by appearing in the label field of a previous (or the same) instruction.
- The duplication factor may not be zero.

Operand Subfield 1: Duplication Factor

With the exception of literals, as noted above, symbols in the expression need not have been previously defined.

Operand Subfield 3: Modifiers

With the exception of literals, as noted above, symbols used as modifiers need not be previously defined. The length attribute of the current statement name is not available for use in the length modifier expression of the first operand for that statement.

Scale Modifier: Symbols used in an expression need not have been previously defined.

Exponent Modifier: Symbols used in an expression need not have been previously defined.

Operand Subfield 4: Constant

X-Type and B-Type Constants: Multiple constants are allowed in X- or B-type operands.

V-Type Constants: V-type constants with zero duplication factors will not generate ESD items.

S-Type Address Constant: S-type address constants can be used in literals. Address decomposition into base and displacement form is based on the USING statements in effect at the associated literal pool.

Q-Type Address Constants: Q-type address constants can also be used in literals. Previous definition is not required for DXD or DSECT names referenced in Q-type constants. A Q-type constant with a zero duplication factor will not generate an ESD item.

DS -- DEFINE STORAGE

The format of the DS operand is identical to the DC operand. See above, "DC -- DEFINE CONSTANT".

Listing Control Instructions

TITLE -- IDENTIFY ASSEMBLY OUTPUT

Only one TITLE instruction, but not necessarily the first, can have a name other than a sequence symbol. The name field in a TITLE instruction may contain from one to eight non-blank characters. The contents of the name field of the first named TITLE statement are punched into the output cards beginning in column 73. Any remaining columns in the sequence field are devoted to deck sequencing. Output cards produced in the program by PUNCH and REPRO assembler instructions are not affected by a TITLE instruction.

Note: If the name field starts with a period, it is considered to be a sequence symbol.

PRINT -- PRINT OPTIONAL DATA

The PRINT instruction is allowed within macro definitions. The PRINT statement is always printed, except when PARM=NOLIST is coded in the EXEC statement that includes Assembler H, regardless of print options already in effect.

PUSH -- SAVE CURRENT USING OR PRINT STATUS

PUSH may be used to save the current PRINT or USING status in push down storage. Conditions may be restored later on a last-in-first-out basis by the use of a corresponding POP operation. PUSH does not change the PRINT or USING currently in effect. The format for the PUSH instruction is as follows:

Name	Operation	Operand
Sequence symbol or blank	PUSH	{USING [,PRINT] / PRINT [,USING]}

The operands may appear in either order. At least one of them must be specified.

POP -- RESTORE USING OR PRINT STATUS

The POP assembler instruction is used to restore the PRINT or USING status saved by the most recent PUSH operation. The format for the POP instruction is:

Name	Operation	Operand
Sequence symbol or blank	POP	{USING [,PRINT] / PRINT [,USING]}

The operands may appear in either order, and need not be in the order in which they appear in the PUSH statement.

ISEQ -- INPUT SEQUENCE CHECKING

Sequence columns to be checked may be anywhere on the card, and thus may be between the begin and end columns as defined by the ICTL instruction. The EBCDIC collating sequence is used. An out-of-sequence card will produce an error diagnostic.

A blank field being checked for sequencing will produce a warning diagnostic, but the checking will continue, relative to the last card with a non-blank sequence field.

ORG -- SET LOCATION COUNTER

The general form of this instruction is:

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

In general, symbols used in the operand need not have been previously defined. However, the simply relocatable component of the operand (that is, the unpaired relocatable term) must have been previously defined, or equated to a previously defined value. The resolved value of the operand must fall in the same LOCTR as the ORG instruction. (See this manual, "LOCTR -- DEFINE LOCATION COUNTER" for a discussion of that instruction.) Remember that CSECT, DSECT, and COM, as well as LOCTR instructions, define a location counter.

The ORG statement cannot be used to reference a location outside the location counter in effect when the ORG is assembled. A negative value is thus not allowed.

A symbol in the name field of an ORG statement is assigned the value of the location counter prior to the application of the ORG. An asterisk (*) in the operand field has the same value as the symbol in the name field. The location counter value, both before and after the processing of the ORG, will appear in the assembly listing.

An ORG statement with a blank operand causes the current location counter to be reset to the highest previous setting within the current LOCTR.

Special Addressing Consideration

All literals not located by LTORG statements within the program are placed at the end of the most recently active LOCTR in the first control section. This includes the case where literals are used after the last LTORG statement in the program as well as the case where no LTORG statements are used at all.

CNOP -- CONDITIONAL NO OPERATION

The format of the CNOP statement is as follows:

Name	Operation	Operand
Any symbol or blank	CNOP	Two absolute expressions, separated by commas

CNOP causes initial halfword alignment, if required. Then the symbol in the name field is assigned the value of the current location counter, and the alignment specified in the operand is completed.

An asterisk (*) in the operand field is assigned the same value as the symbol in the name field. Symbols appearing in the operand expression need not have been previously defined.

COPY -- COPY PREDEFINED SOURCE CODING

Any statement (except ICTL) is allowed within copied code, so long as it conforms to the following rules:

- COPY statements may be nested to any level, but may not be recursive. Thus in the statement "COPY A," if A contains the statement "COPY B" and B contains the statement "COPY C," the copy requests are valid unless B contains the statement "COPY A," or C contains either "COPY A" or "COPY B".
- If a copied partitioned data set member includes a MACRO statement, it must also include the corresponding MEND statement.
- If an END statement or an OPSYN equivalent is encountered, either inside or outside copied code, the assembly is terminated. In BATCH mode, the next assembly begins with the next card in the SYSIN stream.

COPY statements are executed during lookahead and the copied text is scanned. See Section 9, "Attribute Definition and Lookahead".

END -- END ASSEMBLY

The END instruction terminates the assembly whenever it is encountered in the source stream, in copied code, or during the expansion of a macro instruction. The END statement is treated as a model statement in a programmer or library macro while the macro is being edited.

MNOTE -- REQUEST ERROR MESSAGE

MNOTE instructions may be used in open code. They have all the properties of macro instruction MNOTEs.

The severity code is optional. If it is omitted, the MNOTE becomes a comment starting in the begin column, and is not treated as a diagnostic.

Section 6: Introduction to the Macro Language

The Macro Instruction

All macro instructions in Assembler H are mixed mode, which permits you to use the features of both positional and keyword operands within the same macro instruction. Positional and keyword operands may appear in any sequence.

The Macro Definition

The prototype statement specifies only the mnemonic operation code and parameters; defining the type of instruction is not necessary, because all macro instructions for Assembler H are considered mixed mode.

SYSTEM AND PROGRAMMER MACRO DEFINITIONS

Programmer macro instructions may appear at any point in the assembly source text. However, the assembler must encounter a macro definition for a given macro prior to the first call to that macro. Otherwise, the macro instruction will be fetched from SYSLIB or treated as an undefined operation code. The assembler flags a syntax error in a programmer macro immediately following the erroneous statement in the listing.

Source syntax errors in system macro definitions are flagged immediately after the first call to the macro. A system macro can be processed (and therefore listed) as a programmer macro if a COPY is issued to copy the entire macro definition into the source text before a call is made to the macro. For instance:

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│                  │COPY      │SYSMAC                    │
│                  │CSECT     │                          │
│                  │USING     │*,12                      │
│                  │BALR      │12,0                      │
└──────────────────┴──────────┴──────────────────────────┘
```

In this case, SYSMAC is a cataloged system macro. It is copied into the source text, and errors, if any, are flagged as in programmer macros, immediately following the erroneous statements in the listing.

Variable Symbols

A variable symbol may contain up to 62 alphanumeric characters after the ampersand. The first character following the ampersand must be a letter.

Global SET Symbols

Global and explicit local definitions of SET symbols can occur anywhere in open code or in macros, as long as the definitions precede the first use of the SET symbol.

Section 7: How to Prepare Macro Definitions

Programmer macro definitions can appear at any point within the assembly source text. However, the assembler must encounter and edit the definition of a given macro prior to the first call to that macro. It is also possible to redefine a machine or assembler instruction operation code by including a macro definition in the program.

A macro definition may appear within another macro definition. This feature is discussed later in this section under "Nested Macro Definitions".

Macro Instruction Prototype

The symbol appearing in the operation field may be any valid ordinary symbol up to 63 characters long. This includes machine operations, assembler operations, extended mnemonics, etc. A maximum of 240 parameters is permitted in a prototype statement.

All macro definitions are considered to be in mixed mode. Positional and keyword parameters may be mixed freely in both the prototype and the call. The positional parameters are processed from left to right, skipping over the keyword parameters.

The default values specified for keyword parameters can contain sublists nested within sublists. They can also contain embedded equal signs.

Model Statements

The different fields in a macro-generated statement or a statement generated in open code appear in the listing in the same column as they are coded in the model statement, with the following exceptions:

- If the substituted value in the name or operation field is too large for the space available, the next field will be moved to the right with one blank separating the fields.
- If the substituted value in the operand field causes the remarks field to be displaced, the remarks field is written on the next line, starting in the column where it is coded in the model statement.
- If the value substituted in the operation field of a macro-generated statement contains leading blanks, the blanks are ignored.
- If the value substituted in the operation field of a model statement in open code contains leading blanks, the blanks will be used to move the field to the right.

- If the value substituted in the operand field contains leading blanks, the blanks will be used to move the field to the right.
- If the value substituted contains trailing blanks, the blanks are ignored.

Nine continuation cards are allowed on a model statement.

Local and global declarations are processed at generation time. A macro (or a section of open code) may contain more than one declaration for a given SET symbol, so long as only one is encountered during a given macro expansion (or open code conditional assembly), as controlled by AIF and AGO statements.

Operation Field

The operation field may contain any machine, macro or assembler instruction listed in Section 5, except for ICTL. The Assembler H listing control instructions, PUSH and POP, as well as END and PRINT, may appear in this field. The operation field may also contain variable symbols, which may be concatenated with character strings. The result, after substitution and concatenation, is recognized as an op code, subject to the following restrictions.

The following operation codes, or their OPSYN equivalents, may not be created by substitution:

ACTR	GBLA	MACRO
AGO	GBLB	MEND
AGOB	GBLC	MEXIT
AIF	ICTL	REPRO
AIFB	ISEQ	SETA
ANOP	LCLA	SETB
AREAD	LCLB	SETC
COPY	LCLC	

A macro instruction, however, may be created by substitution.

If a statement in a macro definition is recognized as a conditional assembly statement during Pass 1 of the assembly, the meaning of this statement is fixed for all expansions of the macro. All other statements in the edited definition will use the op code definitions in effect whenever the macro is expanded. This must be considered when OPSYN is used with an assembler operation.

Symbolic Parameters

A symbolic parameter consists of an ampersand followed by a letter followed by from 0 to 61 alphanumeric characters. Thus, &RUMPELSTILTSKIN is a valid symbolic parameter.

Nested Macro Definitions

Macro definitions may appear within other macro definitions to any level. These nested definitions are edited only when the macro definition header is encountered during expansion of the immediately containing macro.

Consider the following example:

```
| Name | Operation | Operand          | Comments                    |
|------|-----------|------------------|-----------------------------|
|      | MACRO     |                  | Header                      |
|      | OUTER     | &A,&B,&C=&D      |                             |
|      | AIF       | (K'&C EQ 0).A    |                             |
|      | MACRO     |                  | Header                      |
|      | INNER     | &E,&F,&G,&H=     |                             |
|      | .         |                  | Nested definition of INNER  |
|      | .         |                  |                             |
|      | MEND      |                  | Trailer                     |
| .A   | MEND      |                  |                             |
|      |           |                  | Trailer                     |
```

In order for INNER to be edited and recognized as a macro instruction, OUTER must be encountered as a macro instruction and &C must have a non-zero length.

The operand parameters of a nested macro definition prototype are distinct from those of the containing macro.

Section 8: How to Write Macro Instructions

Macro Instruction Operands

EMBEDDED EQUAL SIGN IN PARAMETER

In a macro call, an equal sign (=) may be embedded in a positional operand or within the value of a keyword operand. The positional operand will be accepted and handled properly. However, a warning message will appear if the (=) is preceded by an alphanumeric string that appears to be a keyword.

Operand Sublists

MULTILEVEL SUBLISTS

Multilevel sublist (sublists within sublists) are permitted in macro operands. The depth of this nesting is limited only by the constraint that the total operand length still may not exceed 255 characters. Inner elements of such sublists are referenced via additional subscripts on parameters or &SYSLIST.

N'&SYSLIST with an n-element subscript array gives the number of operands in the indicated "n-th" level sublist. N' of a parameter name with an n-element subscript array gives the number of operands in the indicated "(n+1)th" level sublist.

Example: If &P is the first positional parameter and the value assigned in a macro instruction is (A,(B,(C)),D) then:

```
┌──────────────────────────────────────────────────────────────────┐
|&P                =&SYSLIST(1)          =(A,(B,(C)),D)            |
|&P(1)             =&SYSLIST(1,1)        = A                       |
|&P(2)             =&SYSLIST(1,2)        =   (B,(C))               |
|&P(2,1)           =&SYSLIST(1,2,1)      =    B                    |
|&P(2,2)           =&SYSLIST(1,2,2)      =      (C)                |
|&P(2,2,1)         =&SYSLIST(1,2,2,1)    =       C                 |
|&P(2,2,2)         =&SYSLIST(1,2,2,2)    =null                     |
|&P(3)             =&SYSLIST(1,3)        =          D              |
|                                                                  |
|N'&P(2,2)         =N'&SYSLIST(1,2,2)    =1                        |
|N'&P(2)           =N'&SYSLIST(1,2)      =2                        |
|N'&P(3)           =N'&SYSLIST(1,3)      =1                        |
|N'&P              =N'&SYSLIST(1)        =3                        |
└──────────────────────────────────────────────────────────────────┘
```

Inner Macro Instructions

An operand of an outer macro instruction sublist may be passed as a sublist to an inner macro instruction.

Section 9: How to Write Conditional Assembly Instructions

AREAD -- Insert Macro Input

The AREAD instruction is used inside a macro definition to process source cards that immediately follow the outermost macro call.

The format of the AREAD instruction is:

Name	Operation	Operand
Any SETC symbol	AREAD	{NOSTMT | NOPRINT}

The SETC symbol in the name field of the AREAD statement may be subscripted. When the macro generator encounters an AREAD statement, the next card in the input stream following the outermost macro call is read and assigned as an 80-character string to the SETC symbol in the name field.

If the macro call containing the AREAD statement occurs within copied code, the cards are read and assigned from the COPY file. The string value may then be examined and processed within the logic of that macro as any SETC value. If neither of the options is used, the card is printed in the source listing and assigned a statement number. If the NOSTMT option is used, the card is printed, but no statement number is assigned to it. If the NOPRINT option is used, the card is not printed and a statement number is not assigned. Repeated AREAD statements read successive cards.

SET Symbols

Defining SET Symbols

A SET symbol is defined in either of two ways: explicitly if the macro generator encounters it in the operand field of a LCLx or GBLx statement, or implicitly by appearing in the name field of a SETx instruction. The implicit declaration defaults to LCLx. The type is determined by the SETx operator, and any dimensionality is determined by the occurrence of a subscript in the name field. If an explicit declaration is encountered later in the assembly, it is flagged as a duplicate declaration.

A SET symbol must be defined before its value is referenced. Implicit declarations are recognized before the operand field of the SET statement is processed. The following examples are both valid uses of a previously undeclared variable symbol:

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│&ABC              │SETA      │&ABC+1                    │
│&X(&X(1)+1)       │SETA      │1                         │
└──────────────────┴──────────┴──────────────────────────┘
```

Attributes

Attributes of symbols produced by macro expansion or open code substitution are available immediately after the defining statement is generated.

Within the proper context of a SETA, SETB, or SETC expression type (T'), length (L'), scaling (S'), integer (I'), and defined (D', see below) attribute reference may be made to a macro instruction parameter, to a SETC symbol inside a macro definition, to a SETC symbol or an ordinary symbol in open code, or to ordinary symbols produced by macro expansion or open code substitution.

Count (K') attribute reference may be applied to all SET variables. In the case of SETA and SETB variables, the value of the variable is first converted to a character string and leading zeros are discarded. If the character string is composed of one or more zeros, the rightmost character is a significant digit and the value of count attribute is 1. In the following example,

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│&A                │SETA      │0100                      │
│&B                │SETB      │0000                      │
└──────────────────┴──────────┴──────────────────────────┘
```

K'&A equals 3 (the leading zero is lost) and K'&B equals 1.

The number (N') attribute may be applied to subscripted SET variables. The number (N') attribute is the highest subscript value that has been involved in a receiving operation. For example, if the only references to variable symbol &A have been the following,

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
│                  │          │                          │
├──────────────────┼──────────┼──────────────────────────┤
│&A(5)             │SETA      │20,,,70                   │
│                  │AIF       │(&A(20) GT 50).M          │
└──────────────────┴──────────┴──────────────────────────┘
```

then N'&A equals 8. (Refer to Section 10, "Extended SET Statements".)

An attribute reference to a variable symbol whose value is an arithmetic expression, in which the leftmost term is a symbol, returns the attribute of that symbol.

Defined Attribute (D')

The defined (D') attribute of a symbol indicates whether the symbol has been defined at the time the D' reference is made. A symbol is considered defined if it has been encountered in the operand field of an EXTRN or WXTRN statement, or in the name field of other statements. The value of D' is 1 (true) if the symbol is defined, 0 (false) if otherwise. For further discussion of symbol definition, see "Attribute Definition and Lookahead" below.

Defined (D') attribute reference may be made to a macro instruction parameter, to a SETC symbol inside a macro definition, to a SETC symbol or an ordinary symbol in open code, and to ordinary symbols produced by macro expansion or open code substitution.

Name	Operation	Operand
&B	SETB	(D'A)
A	LA	1,4

&B is assigned a value of zero because A is undefined at the time D'A is evaluated. The defined attribute value of A is zero.

Name	Operation	Operand
	AIF	(D'A) .AROUND
A	LA	1,4
.AROUND	ANOP	

Similarly, the sequence would cause the statement at A to be assembled only the first time the AIF statement is encountered.

ATTRIBUTE DEFINITION AND LOOKAHEAD

Symbol attributes are established in either definition mode or lookahead mode. Lookahead mode is entered when Assembler H encounters an attribute reference to a symbol that is not yet defined.

Definition Mode

Definition occurs whenever a previously undefined symbol is encountered in the name field of a statement, or in the operand field of an EXTRN or WXTRN statement during open code processing. Symbols within a macro definition are defined when the macro is generated.

Lookahead Mode

Lookahead is a sequential, statement-by-statement, forward scan over the source text. It is initiated when reference is made to an attribute (other than D') of a symbol not yet encountered, either by macro or open-code attribute reference, or by a forward AGO or AIF branch in open code.

If reference is made in a macro, forward scan begins with the first source statement following the outermost macro instruction. Programmer macros are bypassed. The text is not assembled. Lookahead attributes are tentatively established for all intervening undefined symbols. Tentative attributes are replaced and fixed when the symbol is subsequently encountered in definition mode. No macro expansion or open-code substitution is performed; no conditional or unconditional (AIF or AGO) branches are taken. COPY instructions are executed during lookahead, and the copied statements are scanned.

Lookahead ends when the desired symbol or sequence symbol is found, or when the END card or end of file is reached. All statements passed over by lookahead are saved on an internal file and processed when the lookahead ends.

For purposes of attribute definition, a symbol is considered undefined if it depends in any way upon a symbol not yet defined. For example, if the symbol is defined by a forward EQU that is not yet resolved, or if a DC, DS, or DXD modifier expression contains symbols not yet defined, that symbol is assigned a type attribute of U.

NOTE: Since no variable symbol substitution is performed by lookahead, you should be careful when using a macro or open code substitution to generate END statements that separate source modules assembled in one job step (option BATCH). If a symbol is undefined within a module, lookahead will read in records past the point where the END statement is to be generated. All statements between the generated statement and the point where lookahead stops (either because it finds a matching symbol, or because it finds an END statement) are ignored by the assembler. The next module will start at the point where lookahead stops.

Lookahead Restrictions

Assembler statements are analyzed only to the extent necessary to establish attributes of symbols in their name fields.

Variable symbols are not replaced. Modifier expressions are evaluated only if all symbols involved were defined prior to lookahead. Possible multiple or inconsistent definition of the same symbol is not

diagnosed during lookahead because conditional assembly may eliminate one (or both) of the definitions.

Lookahead does not check undefined op codes against library (system) macro names. If the name field contains an ordinary symbol and the op code cannot be matched with one in the current op code table, then the ordinary symbol is assigned the type attribute of M. If the op code contains special characters or is a variable symbol, a type attribute of U is assumed. This may be wrong if the undefined op code is later defined by OPSYN. OPSYN statements are not processed; thus, labels are treated in accordance with the op code definitions in effect at the time of entry to lookahead.

Sequence Symbols

A sequence symbol may be used in the name field of an ACTR, COPY, GBLA, GBLB, GBLC, LCLA, LCLB, LCLC, MACRO, and any other statement where an ordinary symbol is either optional or not allowed.

LCLA, LCLB, LCLC -- Define Local SET Symbols

Because local and global declarations are processed at generation time, a macro or open code may contain more than one declaration for a given SET symbol as long as only one declaration is encountered (via AIF and AGO statements) during a given macro expansion or open code assembly.

SETA -- Set Arithmetic

A SETC variable may be treated as an arithmetic term if its value string represents any valid self-defining term. A null value is treated as a zero.

This feature allows you to associate numeric values with EBCDIC or hexadecimal characters to be used in such applications as indexing, code conversion, translation, and sorting.

Assume that &X is a character string with the value ABC:

```
┌--------------------┬-----------┬-----------------------------┐
|        Name        | Operation |          Operand            |
|                    |           |                             |
├--------------------┼-----------┼-----------------------------┤
|&I                  |SETC       |'C'''.'&X'(1,1).''''         |
|&VAL                |SETA       |&TRANS(&I)                   |
└--------------------┴-----------┴-----------------------------┘
```

The first statement sets &I to C'A'. The second statement extracts the 193rd element of &TRANS (C'A' = X'C1' = 193).

The following code will convert a hex value in &H into a decimal value in &VAL:

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
├──────────────────┼──────────┼──────────────────────────┤
│&X                │SETC      │'X''&H'''                 │
│&VAL              │SETA      │&X                        │
└──────────────────┴──────────┴──────────────────────────┘
```

An arithmetic expression may not contain two terms in succession; however, any term may be preceded by any number of unary operators. +&A*-&B is a valid operand for a SETA instruction. The expression &FIELD+- is invalid because it has no final term.

Evaluation of Arithmetic Expressions

The number of levels of parentheses in a SETA arithmetic expression must not exceed 255.

SETC -- Set Character

Any of the expressions permitted in the operand field of a SETC statement (a type attribute, a character expression, a substring, or a SETA symbol) may optionally be preceded by a SETA expression enclosed in parentheses to be used as a duplication factor. If &J has a value of 3, the following two statements are equivalent.

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
├──────────────────┼──────────┼──────────────────────────┤
│&C                │SETC      │(2) 'XY'.(&J+1) 'PDQ'     │
│&C                │SETC      │'XYXY'.'PDQPDQPDQPDQ'     │
└──────────────────┴──────────┴──────────────────────────┘
```

Note: If a SETC duplication factor and a substring are used in the same SETC statement, the substring is evaluated first, and the duplication factor is evaluated second. For example, the following statement will yield "BCBC", not "BCA".

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
├──────────────────┼──────────┼──────────────────────────┤
│&STRING           │SETC      │(2) 'ABC' (2,3)           │
└──────────────────┴──────────┴──────────────────────────┘
```

Character Expression

All characters in a character expression are assigned to the SETC symbol in the name field. The maximum length character value that can be assigned to a SETC symbol is 255 characters.

Substring Notation

The maximum length substring character value that can be assigned to a SETC symbol is 255 characters.

SETB -- Set Binary

Evaluation of Logical Expressions

There is no limit to the levels of parentheses allowed in a logical expression.

AIF - Conditional Branch

EXTENDED AIF STATEMENTS

The extended AIF statement has the following format:

Name	Operation	Operand
A sequence symbol or blank	AIF	(logical expression).S1, (logical expression).S2, ..., (logical expression).Sn

It is exactly equivalent to n successive AIF statements. The branch is taken to the first sequence symbol (scanning left to right) whose corresponding logical expression is true. If none of the logical expressions is true, no branch is taken.

Consider the following example:

Operation	Operand	Col. 72
AIF	('&L'(&C,1) EQ '$').DOLR,('&L'(&C,1) EQ '#').POUND,	X
	('&L'(&C,1) EQ '@').AT,('&L'(&C,1) EQ '=').EQUAL,	X
	('&L'(&C,1) EQ '(').LEFTPAR,('&L'(&C,1) EQ '+').PLUS,	X
	('&L'(&C,1) EQ '-').MINUS	

This routine looks for the occurrence of a $, #, @, =, (, +, and -, in that order; and causes control to branch to .DOLR, .POUND, .AT, .EQUAL, .LEFTPAR, .PLUS, and .MINUS, respectively, if the string being examined contains any of these characters.

Note: The example shown above indicates that the alternate format is allowed for this statement. Refer to "Alternate Format in Conditional Assembly" in Section 10.

AGO -- Unconditional Branch

COMPUTED AGO STATEMENTS

The computed AGO statement has the following format:

Name	Operation	Operand
A sequence symbol or blank	AGO	(arithmetic expression) .S1,.S2,...,.Sn

If the arithmetic expression evaluates to k, where k lies between 1 and n (inclusive), then the branch is taken to the "k-th" sequence symbol in the list. If k is outside that range, no branch is taken.

In the following example:

Name	Operation	Operand
	AGO	(&I).FIRST,.SECOND,.THIRD,.FOURTH

control passes to the statement at .THIRD if &I=3. Control passes through to the statement following the AGO if &I is less than 1 or greater than 4.

ACTR -- Conditional Assembly Loop Counter

The ACTR instruction may occur at any point within a macro definition or in the source program. When the ACTR instruction is encountered during macro generation or in open code, the counter is reset as indicated by the operand field. Most errors detected during a macro expansion cause the current ACTR value to be halved rather than decremented by 1. Such errors include all model statements with incorrect syntax and those errors with a severity greater than 4. This shortens any looping brought about by erroneous statements. Use extreme caution when resetting this counter within a loop.

ANOP -- Assembly No Operation

The format for the ANOP instruction is:

Name	Operation	Operand
A sequence symbol or blank	ANOP	Blank

Thus, a blank in the name field of the instruction is no longer diagnosed as an error. An ANOP statement with a blank name field can only be reached sequentially from the previous statement. AGO and AIF statements must reference valid sequence symbols.

Section 10: Extended Features of the Macro Language

MNOTE -- Request for Error Message

The MNOTE instruction may be used either in open code or macro definitions. The severity code is optional. If it is omitted, the MNOTE becomes a comment starting in the begin column and is not treated as a diagnostic.

Global and Local Variable Symbols

Defining Local and Global SET Symbols

Global and local declarations may occur anywhere. Such declarations must precede any reference to the symbol to be effective in the generation process.

Alternate Format in Conditional Assembly

Alternate format allows a group of operands to be spread over several lines of code. Each line, except the last, is followed by a comma, one or more blanks, and a non-blank character in column 72. Comments are inserted optionally between the blank and column 72. In addition to macro prototype and macro call statements, alternate format can be used for extended AGO, AIF, and SETx statements, and GBLx and LCLx declarations.

Subscripted SET Symbols

SET symbol definition is discussed in "Defining SET Symbols" in Section 9 of this book. A subscripted SET symbol is declared explicitly by following the symbol with a positive, decimal self-defined number enclosed in parentheses. A subscripted SET symbol is implicitly subscripted if it occurs first in the name field of a SET statement in subscripted notation. The subscript limit is open-ended and may be raised at any time simply by writing a higher number between the parentheses.

The following example illustrates valid SET symbol subscripting:

```
|Name     |Operation |Operand          |Comments                          |
|         |          |                 |                                  |
|         |LCLA      |&A(1)            |Explicitly declared,              |
|         |          |                 |subscripted                       |
|&A(5)    |SETA      |5                |                                  |
|&B(20)   |SETA      |600              |Implicitly declared,              |
|         |          |                 |subscripted                       |
```

Subscript usage of any global SET symbol must be consistent for the entire assembly: declarations of a global symbol must be either all subscripted or unsubscripted. The subscript limit remains open-ended. Consider the following example:

```
|        Name        | Operation |          Operand          |
|                    |           |                           |
|                    |GBLC       |&CHAR(1)                   |
|                    |MACRO      |                           |
|                    |UPDATE     |&ARG1,&ARG2                |
|                    |GBLC       |&CHAR(50)                  |
|                    |...        |                           |
|                    |MEND       |                           |
```

Once defined as a subscripted SET symbol, subsequent declarations of &CHAR in the previous illustration must be subscripted.

The number attribute (N') of a subscripted SET symbol is the highest subscript value involved in a stored-into operation. Thus, if the only statements involving &X are the following:

```
|        Name        | Operation |          Operand          |
|                    |           |                           |
|&X(5)               |SETA       |10                         |
|&X(10)              |SETA       |5                          |
|                    |AIF        |(&X(20) GT 6).TOOBIG       |
```

then N'&X=10. You should be aware of the consequences. For example, the following loop will be terminated only when the value of the conditional assembly loop counter (ACTR) becomes zero.

```
|        Name        | Operation |          Operand          |
|                    |           |                           |
|                    |LCLA       |&I,&X(5)                   |
|.START              |ANOP       |                           |
|&I                  |SETA       |&I+1                       |
|&X(&I)              |SETA       |&I                         |
|                    |AIF        |(&I GT N'&X).OUT           |
|                    |AGO        |.START                     |
```

Extended SET Statements

A single SET statement can assign values to multiple elements in an array. Consider the following example.

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
├──────────────────┼──────────┼──────────────────────────┤
│&ARRAY(1)         │SETA      │5,10,15,20,25,30          │
└──────────────────┴──────────┴──────────────────────────┘
```

The SET declaration defaults to LCLA if there has been no previous global or local declaration of &ARRAY. The subscript of the variable symbol is incremented by 1 for each operand value. Following the above instruction, &ARRAY(1) has a value of 5, &ARRAY(2) has a value of 10, and so forth.

If one of the operands is omitted, the corresponding element of the array is unchanged. In the following example,

```
┌──────────────────┬──────────┬──────────────────────────┐
│       Name       │ Operation│         Operand          │
├──────────────────┼──────────┼──────────────────────────┤
│&LIST(6)          │SETA      │,5,10,,20,25,30           │
└──────────────────┴──────────┴──────────────────────────┘
```

the elements referenced by &LIST(6) and &LIST(9) are left unchanged. Note that N'&LIST=12 if this is the only statement involving &LIST.

Alternate format may be used for the operands of an extended SET statement, as with the operands of a macro call or macro definition. Thus, the above coding could be written as follows:

```
┌──────────┬──────────┬──────────────┬──────────────────────────┐
│   Name   │ Operation│    Operand   │         Column 72        │
├──────────┼──────────┼──────────────┼──────────────────────────┤
│&LIST(6)  │SETA      │,5,           │X                         │
│          │          │10,           │X                         │
│          │          │,20,25,30     │                          │
└──────────┴──────────┴──────────────┴──────────────────────────┘
```

Created SET Symbols

A created SET symbol has the form &(e) where "e" represents a sequence of one or more of the following:

- Variable symbols, optionally subscripted.
- Strings of alphanumeric characters.
- Created SET symbols.

After substitution and concatenation, "e" must consist of a string of up to 62 alphanumeric characters, the first being alphabetic. This string will then be used as the name of a SET variable.

Created SET symbols may be used wherever ordinary SET symbols are permitted, including declarations; they may also be nested in other created SET symbols. Consider the following example:

```
+------------------+-----------+-----------------------------+
|       Name       | Operation |          Operand            |
|                  |           |                             |
+------------------+-----------+-----------------------------+
|&ABC(1)           |SETC       |'MKT','27','$5'              |
+------------------+-----------+-----------------------------+
```

Let &(e) equal &(&ABC(&I)QUA&I).

```
+-------+----------+--------------------+-------------------------------+
|&I     |&ABC(&I)  |Created SET Symbol  |Comment                        |
|       |          |                    |                               |
+-------+----------+--------------------+-------------------------------+
|1      |MKT       |&MKTQUA1            |Valid                          |
|2      |27        |&27QUA2             |Invalid: first                 |
|       |          |                    |character after                |
|       |          |                    |& not alphabetic               |
|3      |$5        |&$5QUA3             |Valid                          |
|4      |          |&QUA4               |Valid                          |
+-------+----------+--------------------+-------------------------------+
```

The created SET symbol can be thought of in one sense as a form of indirect addressing. With nested created SET symbols, you can get this kind of indirect addressing to any level.

In another sense, created SET symbols offer an associative memory facility. For example, a symbol table of numeric attributes can be referenced by an expression of the form &(&SYM)(&I) to yield the "I-th" attribute of the symbol named in &SYM.

Created SET symbols also enable you to get some of the effect of multiple-dimensioned arrays by creating a separate name for each element of the array. For example, a three-dimensional array of the form &X(&I,&J,&K) could be addressed as &(X&I.$&J.$&K). Thus "&X(2,3,4)" would be represented by &X2$3$4. The "$"s guarantee that &X(2,33,55) and &X(23,35,5) are unique:

```
&X(2,33,55) becomes &X2$33$55
&X(23,35,5) becomes &X23$35$5
```

MHELP -- Macro Trace Facility

The MHELP instruction controls a set of trace and dump facilities. Options are selected by an absolute expression in the MHELP operand field. MHELP statements can occur anywhere in open code or in macro definitions. MHELP options remain in effect until superseded by another MHELP statement. The format of this instruction is as follows:

Name	Operation	Operand
A sequence symbol or blank	MHELP	Absolute expression, binary or decimal options as discussed below.

Macro Call Trace -- Operand=1

This option provides a one-line trace listing for each macro call, giving the name of the called macro, its nested depth, and its &SYSNDX value. The trace is provided only upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

Macro Branch Trace -- Operand=2

This option provides a one-line trace listing for each AGO and AIF conditional-assembly branch within a macro. It gives the model-statement numbers of the "branched from" and the "branched to" statements, and the name of the macro in which the branch occurs. This trace option is suppressed for library macros.

Macro AIF Dump -- Operand=4

This option dumps undimensioned SET symbol values from the macro dictionary immediately before each AIF statement that is encountered.

Macro Exit Dump -- Operand=8

This option dumps undimensioned SET symbols from the macro dictionary whenever a MEND or MEXIT statement is encountered.

Macro Entry Dump -- Operand=16

This option dumps parameter values from the macro dictionary immediately after a macro call is processed.

Global Suppression -- Operand=32

This option suppresses global SET symbols in two preceding options, MHELP 4 and MHELP 8.

MHELP Suppression -- Operand=128

This option suppresses all currently active MHELP options.

MHELP Control On &SYSNDX

The MHELP operand field is actually mapped into a full word. Previously-defined MHELP codes correspond to the fourth byte of this fullword.

&SYSNDX control is turned on by any bit in the third byte (operand values 256-65535 inclusive). Then, when &SYSNDX (total number of macro calls) exceeds the value of the fullword which contains the MHELP operand value, control is forced to stay at the open-code level, by in effect making every statement in a macro behave like a MEXIT. Open code macro calls are honored, but with an immediate exit back to open code.

Examples:

```
MHELP 256        Limit &SYSNDX to 256.
MHELP 1          Trace macro calls.
MHELP 256+1      Trace calls and limit &SYSNDX to 257.
MHELP 65536      No effect. No bits in bytes 3,4.
MHELP 65792      Limit &SYSNDX to 65792.
```

When the value of &SYSNDX reaches its limit, the message "ACTR Exceeded -- &SYSNDX" is issued.

Combining Options

As shown in the example above, multiple options can be obtained by combining the option codes in one MHELP operand. For example, call and branch traces can be invoked by MHELP B'11', MHELP 2+1, or MHELP 3. Substitution by means of variable symbols may also be used.

System Variable Symbols

There are four new system variable symbols available--&SYSDATE, &SYSTIME, &SYSPARM, and &SYSLOC--in addition to &SYSNDX, &SYSLIST, and &SYSECT.

&SYSPARM, &SYSTIME, and &SYSDATE may be used as desired in macro definitions and open code. &SYSNDX, SYSECT, &SYSLIST, and &SYSLOC may be used only inside macro definitions.

&SYSLIST -- Macro Instruction Operand

&SYSLIST refers only to positional macro instruction operands. Interspersed keyword operands are skipped over during &SYSLIST reference.

&SYSDATE -- Date of Assembly

&SYSDATE provides the date of the assembly and corresponds to the date printed in the page heading. The value of &SYSDATE is the 8-character string mm/dd/yy (month/day/year). The value remains constant throughout the assembly, and may be used both inside and outside macro definitions.

&SYSTIME -- Time of Assembly

&SYSTIME provides the time printed in the page heading. The value of &SYSTIME is the five-character string hh.mm (hours.minutes). As with &SYSDATE above, the value remains constant and may be used both inside and outside macro definitions.

For systems without the internal timer feature, &SYSTIME is a five-character string of blanks.

&SYSPARM -- User Assembly Parameter

The system variable &SYSPARM operates in conjunction with the SYSPARM parameter in the PARM field of the EXEC card in the JCL sequence. Together they allow you to control conditional assembly flow and code generation through the use of an externally specified parameter. The value of &SYSPARM is the character string specified in the SYSPARM

parameter. Otherwise, it is the default value specified at system generation time.

For example, in a section of code to conditionally produce a debug request, the execute card might look like this:

```
//STEP EXEC ASMHC,PARM='SYSPARM(DEBUG)'
```

And the code might look like this:

```
        Name        Operation          Operand
                    AIF          ('&SYSPARM' NE 'DEBUG').SKIP
                    SNAP         (might be the debug request)
.SKIP               ANOP
```

If the debug request is no longer desired, the SYSPARM value on the EXEC card may be reset and the AIF will skip over the debug request.

As with &SYSTIME and &SYSDATE, the value of &SYSPARM remains constant and may be used both inside and outside macro definitions.

If no value has been specified for SYSPARM, then &SYSPARM is the default value specified at system generation time. Two quotes are needed to represent a single quote and two ampersands are needed to represent a single ampersand. The character comma is allowed only if the string is enclosed in paired parentheses.

&SYSLOC -- Current Location Counter

&SYSLOC is used to represent the name of the location counter in effect at the time of a macro definition. &SYSECT gets the value of the current CSECT, DSECT, or START control section. If no LOCTR statement is in effect, the value of &SYSLOC is the same as &SYSECT.

It is often desired to return to a caller's section. However, if a CSECT, DSECT, or COM has been generated in a macro, you have no way of knowing what type of a statement to put an &SYSECT reference on. A solution is "&SYSLOC LOCTR". Since LOCTR resumes the section type as well as the location counter, it automatically provides the correct section type.

Index

I

L

M

N

O

P

IBM Technical Newsletter

This Newsletter No. GN33-8204
Date September 30, 1975

Base Publication No. GC26-3771-3
File No. S360-21(OS)

Previous Newsletters None.

OS ASSEMBLER H LANGUAGE

This Technical Newsletter, a part of version 4 of the Operating System Assembler H Program Product 5734-AS1, provides replacement pages for the subject manual. Pages to be inserted and/or removed are:

Front Cover,ii
3,4
25,26
26.1,26.2 (Added)
15,16 (Part II)
Readers Comment Form/Reply (Removed)

A change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

Minor technical changes have been made, and the manual has been placed under class C maintenance.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden

Printed in U.S.A.

GC26-3771-3

OS Assembler H Language (File No. S360-21(OS)) Printed in U.S.A. GC26-3771-3

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)