GC26-3758-3
File No. S360-21 (OS)

Program Product

OS Assembler H
General Information Manual

IBM®

GC26-3758-3
File No. S360-21 (OS)

Program Product

OS Assembler H
General Information Manual

Program Number 5734-AS1

IBM

Fourth Edition (January, 1974)

This is a reprint of GC26-3758-2 incorporating changes released in the following Technical Newsletter: GN33-8151 (dated September 29, 1972).

This edition applies to version 4 of the Assembler H Program Product (Program Number 5734-AS1) and to all subsequent modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography, Order No. GA22-6822, for the editions that are applicable and current.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 printer using a special print chain.

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

Forms are provided at the back of this publication for readers' comments. If the forms have been removed, comments may be addressed to IBM Nordic Laboratory, Programming Publications, Box 962, S-181 09 Lidingö 9, Sweden. Comments become the property of IBM.

Read This First

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

If your installation uses OS/VS, you should read the first part, 'Part I: For the VS User'. This part assumes that you are familiar with the language supported by the OS/VS Assembler as described in OS/VS and DOS/VS Assembler Language, Order No. GC33-4010. That manual is a corequisite for this part.

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 assumes that you are familiar with the language supported by the OS Assembler F as described in OS Assembler Language, Order No. GC28-6514. That manual is a corequisite for this part.

Part I: For the VS User

Preface

This manual is an introduction to the language and facilities of OS Assembler H. It is for system programmers who are already familiar with the assembler language.

The manual highlights the features of the language supported by Assembler H compared with the language supported by the OS/VS Assembler. The major subjects are:

- Basic structure and features of Assembler H.
- System requirements for Assembler H.
- Additions and extensions to the basic assembler language supported by Assembler H.
- Additions and extensions to the macro and conditional assembly facilities supported by Assembler H.
- Additions and extensions to the diagnostic services supported by Assembler H.

The reader should be familiar with the assembler language supported by the VS Assembler, as described in OS/VS and DOS/VS Assembler Language, GC33-4010. More detailed information about the language supported by Assembler H may be found in OS Assembler H Language, GC33-3771.

The reader should also have a general idea of the basic concepts of OS. Such information can be found in OS Introduction, GC28-6534.

Contents

Figures

Introduction

OS Assembler H is an assembler language processor with major extensions to the language and performance of the VS Assembler.

Language Compatibility

Any program successfully assembled with no warning or diagnostic messages by the VS Assembler will be assembled correctly by Assembler H. Programs using features supported only by Assembler H will not be assembled correctly by the VS Assembler.

Language Changes

Many limitations of the assembler language supported by the VS Assembler are relaxed or eliminated in the language supported by Assembler H. Changes and additions to the language fall into the following categories:

- Conditional assembly instructions have expanded functions and flexibility.
- Many assembler instructions have fewer restrictions and allow improved programmer control.
- New assembler instructions are added to the language.

Performance

The high-speed assembly capability of Assembler H is a result of the following factors:

- All text processing is performed in virtual storage if the region allocated to the assembler is sufficiently large.
- The number of source text passes is reduced.
- Multiple assemblies can be performed under the control of one set of job control cards (in one job step).
- You can use conditional assembly instructions to bypass macro definitions included in the source text.

System Requirements

Central Processing Unit: Assembler H operates on an IBM System/370 Model 135 or higher.

Operating Environments: Assembler H operates under OS/VS.

Virtual Storage: Assembler H operates in a minimum region size of 192K bytes of virtual storage. However, the recommended size is 256K bytes.

Data Sets: System input and output device requirements are roughly the same as those of the VS Assembler. The only difference is that Assembler H requires only one utility data set.

Library Space: In terms of the IBM 2314 Direct Access Storage Facility, cataloged procedures for Assembler H require a maximum of one track on SYS1.PROCLIB and the Assembler H load modules need approximately 30 tracks on SYS1.LINKLIB or a private link library.

Internal Design

The internal organization of Assembler H is an entirely new design. There are only two source text passes, as opposed to three for the VS Assembler. Pass one of the source text by Assembler H edits and expands macros, and builds dictionaries and the symbol table. Pass two completes the assembly and produces the desired output. Figure 1 shows the general processing steps of the VS Assembler and Assembler H.

RESOLVING SYMBOL ATTRIBUTE REFERENCES

The symbol table is built as symbols are encountered in macro generation and open-code assembly. If an attribute reference is made to a previously undefined symbol, Assembler H proceeds with a forward scan of the source text, called "lookahead" mode. It continues the forward scan until the symbol that initiated the scan is resolved or until the end of the source program is encountered. During the scan, the assembler conditionally places all other symbols that are encountered into the symbol table, so further forward scans are avoided unless a forward reference is made to a symbol, at some point beyond the previous forward reference. The symbol attributes established by the forward scan are not fixed, however, and can be overridden when the symbol is placed in the symbol table as a result of regular assembly. If the symbol that initiated the forward scan is not found, a diagnostic message is issued.

INTERNAL TEXT PROCESSING

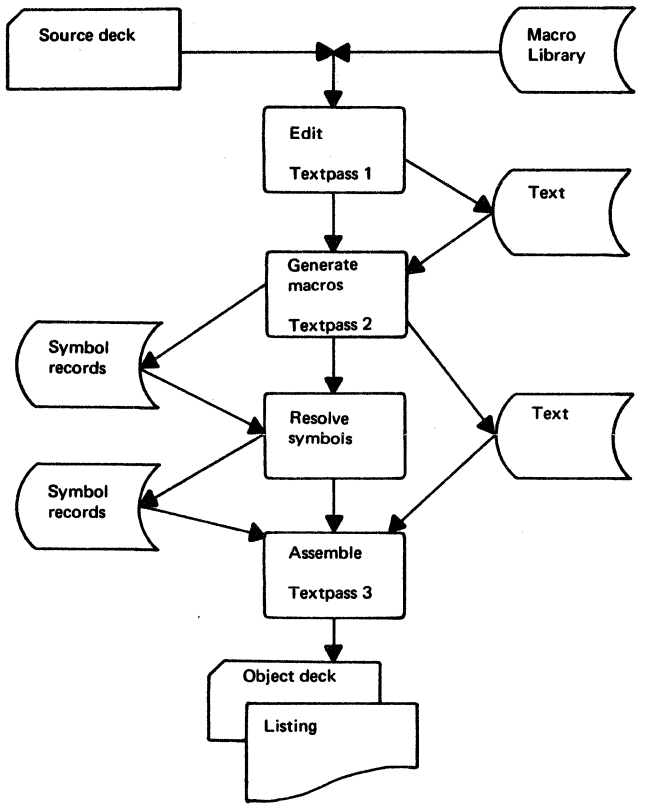
Because the VS Assembler is designed to process intermediate text using external workfiles, it cannot use region sizes larger than 40K bytes as effectively as Assembler H. Assembler H processes all intermediate assembly text in virtual storage, if the region allocated to the assembler is large enough to contain the text. There is no limit to the region size that can be used efficiently by Assembler H, provided that the source module is large enough.

Within the region allocated for an assembly of a source program, the amount of working storage Assembler H uses to perform an operation can dynamically expand to meet the storage requirements of that operation. When one block of working storage is filled with processed text, processing continues with the allocation of another block from within the region acquired for the assembly.

As blocks of working storage are filled with processed text, they are flagged to indicate whether they can be written out to the external workfile (SYSUT1). Partially filled blocks and those blocks taken up by the symbol table must remain in virtual storage at all times during the assembly. Those blocks that can be written out are put on the workfile, but the blocks of text are also retained in working storage and continue to be effective for all assembly purposes. Only when all unallocated workspace within the region is exhausted are the written-out text blocks in working storage reinitialized and overlaid with newly processed text. Then, when needed, the overlaid blocks of text must be accessed from the workfile.

NOTE: If all blocks of working storage are allocated and flagged as resident when an operation requires additional workspace, the assembly is terminated. Such assemblies must either be broken down into subroutines or be assembled in a larger region.

Assembler XF:



Assembler H:

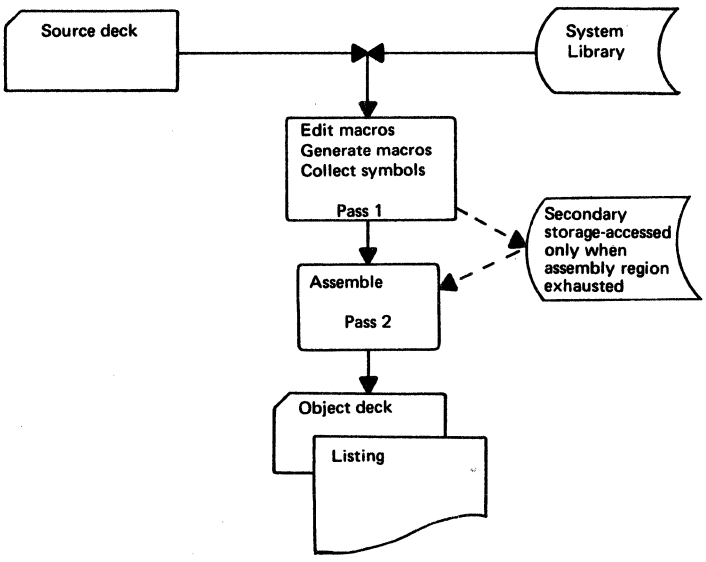


Figure 1. Processing Steps of Assembler H and the VS Assembler

Modifying Assembler H when Adding It to Your System

There are several optional modifications that you can make to Assembler H to tailor it to fit the requirements of your installation. These modifications allow you to alter the default values for assembler options, to change the DD names for assembler data sets, and to choose the machine instruction set you want the assembler to support. You make the modifications when the assembler is added to your system, using an option-setting routine that is provided with the assembler. This routine is used only if you want to specify default options other than the standard options specified in the assembler when it is delivered to you. This section describes in detail the modifications you can make to the Assembler H.

DEFAULTS FOR ASSEMBLER OPTIONS

When you call the assembler with the EXEC job control statement, you can specify values for assembler options in its PARM field to override the standard values set in the assembler. Standard values for the options are set at delivery. However, they can be respecified by your installation when the assembler is added to your system. The options are:

Standard Value	Alternate Value
DECK	NODECK
NOOBJECT	OBJECT
LIST	NOLIST
XREF(FULL)	XREF(SHORT), NOXREF
NORENT	RENT
NOTEST	TEST
NOBATCH	BATCH
ALIGN	NOALIGN
ESD	NOESD
RLD	NORLD
LINECOUNT(55)	LINECOUNT(a value in the range 1 - 99)
FLAG(0)	FLAG(a value in the range 0 - 255)
SYSPARM() (null string)	SYSPARM(a character string 1 - 255 characters long)

NOTE: If you use the PARM field of the EXEC statement to specify a SYSPARM value, the maximum length you can use is 56 characters because of job control language restrictions.

Your installation can also remove certain options, so that they cannot be specified. For example, you may want to change the standard value from DECK to NODECK and remove DECK so that it cannot be specified.

DATA DEFINITION NAMES FOR ASSEMBLER H DATA SETS

Assembler H requires the following data set DD names:

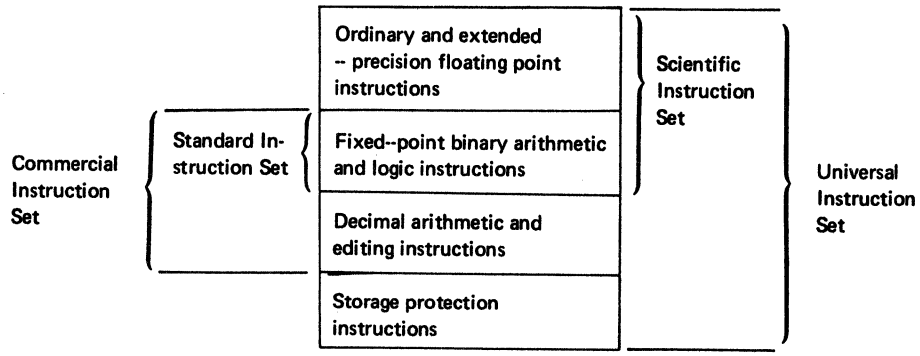
SYSIN
SYSLIB (if library members are called by macro instructions or COPY statements)

SYSLIN (if OBJECT is specified)
SYSPRINT (if LIST is specified)
SYSPUNCH (if DECK is specified)
SYSUT1

Any of these names can be changed when the assembler is added to your system. For example you may wish to replace SYSUT1 with WORK001, SYSIN with SYSINPUT, etc.

INSTRUCTION SET OPTIONS

The instruction set, or operation code table available to the Assembler H can be specified when the assembler is added to your system. There are four instruction sets available, as shown below.



Extensions to Macro and Conditional Assembly Language

The macro and conditional assembly language supported by Assembler H offers significant extensions over the language supported by the VS Assembler. Many restrictions are relaxed or eliminated to increase flexibility and extend language functions. For example, many ordering restrictions are removed from conditional assembly statements in macro definitions and in open code.

General Advantages in Using Macros

You can think of a macro definition as a subroutine which can be modified each time it is called by a macro instruction. In modifying this subroutine, the assembler uses values passed in the macro instruction (for symbolic parameters). Further, the assembler uses values passed from other macros or from open code (global SET symbols) and data attribute references. The modified subroutine is included in your program in basic assembler language format; the assembler then processes it in the same way as any other source statements. By varying the symbolic parameters and global SET symbols, you can vary the generated assembler instructions and the sequence in which they are generated.

Using macros gives you a scope similar to what you have when using a problem-oriented language. In fact, you can use macros to create your own language, tailored to your specific applications.

Extensions to the Macro Language

The extended functions of the macro definition and the macro instruction in Assembler H improve programmer control and coding flexibility. For example, macro definitions can appear anywhere in your source module; they can even be nested within other macro definitions. They can also be redefined at a later point in your program, and macro instruction operation codes can be generated by substitution.

PLACEMENT OF MACRO DEFINITIONS

The VS Assembler allows macro definitions only at the very beginning of a source module. Under Assembler H, the only restriction is that the macro definition must be encountered before it is called.

EDITING MACRO DEFINITIONS

The initial processing of a macro definition is called editing. Editing of a macro involves, among other things, checking of the syntax of the instructions and changing the source statements to special edited text used throughout the remainder of the assembly. The edited version of the macro definition is used to generate assembler language statements

when the macro is called by a macro instruction. Therefore, a macro must always be edited before it can be called by a macro instruction.

Under the VS Assembler all source macros must appear at the beginning. They cannot be bypassed using conditional assembly instructions. Under Assembler H, however, you can use conditional assembly statements to avoid editing of certain macros. In the following example, the macro definition for MACSHOW is bypassed and not edited, if the value of the system parameter (&SYSPARM) is NOTMACSHOW. Any macro instructions calling the macro are invalid.

```
Name    Operation   Operand
        AIF         ('&SYSPARM' EQ 'NOTMACSHOW').PASS
        MACRO
        MACSHOW
        .
        MEND
.PASS   ANOP
```

REDEFINING MACROS

A macro definition can be redefined at any point in your source module. When a macro is redefined, the new definition is effective for all subsequent macro instructions that call it.

Once a macro has been redefined by a macro definition, its previous function is lost, unless prior to redefinition, the operation is assigned to another symbol with an OPSYN instruction. Later, if you wish the initial function of the operation code to be reestablished, you can include another OPSYN instruction to redefine it. The following example illustrates this:

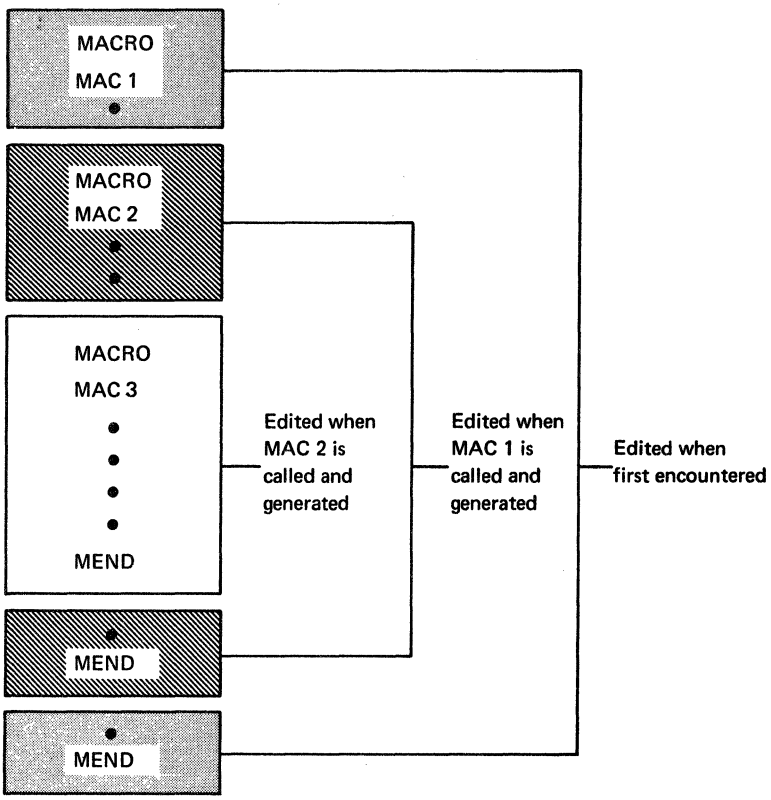
Name	Operation	Operand	
	MACRO MAC1 . MEND		The symbol MAC1 is assigned as the name of this macro definition.
	.		
MAC2	OPSYN	MAC1	MAC2 is assigned as an alias for MAC1
	MACRO MAC1 . MEND		MAC1 is assigned as the name of this macro definition.
	.		
MAC1	OPSYN	MAC2	MAC1 is assigned to the first definition. The second definition is lost.

You can also reestablish a previous source macro definition by issuing a conditional assembly branch (AGO or AIF) to a point prior to the initial definition of the macro. Then that definition will be edited and effective for subsequent macro instructions calling it. Consider the following example:

Name	Operation	Operand	
.UP	ANOP		
	MACRO		
	MAC1		Assign MAC1 to first macro definition.
	.		
	MEND		
	.		
	MACRO		
	MAC1		Assign MAC1 to second definition.
	.		
	MEND		
	.		
	AGO	.UP	Branch to a point prior to first definition.

NESTING MACRO DEFINITIONS

In the VS Assembler, macro definitions can contain inner macro instructions but not inner macro definitions. Assembler H allows both inner macro instructions and inner macro definitions. The inner macro definition is not edited until the outer macro is generated as the result of a macro instruction calling it, and then only if the inner macro definition is encountered during the processing of the outer macro. Thus, if the outer macro is not called, or if the inner macro is not encountered in the generation of the outer macro, the inner macro definition is never edited. The following figure illustrates the editing of inner macro definitions.



First MAC1 is edited, and MAC2 and MAC3 are not. When MAC1 is called MAC2 is edited (unless it is passed by an AIF or AGO branch) and when MAC2 is called, MAC3 is edited. No macros can be accessed by a macro instruction until they have been edited.

There is no limit to the number of nestings allowed for inner macro definitions.

GENERATED MACRO INSTRUCTION OPERATION CODES

The VS Assembler does not allow the operation code of a macro instruction to be generated by substitution. That restriction does not apply to Assembler H. Macro instruction operation codes can be generated by substitution, either in open code or inside macro definitions.

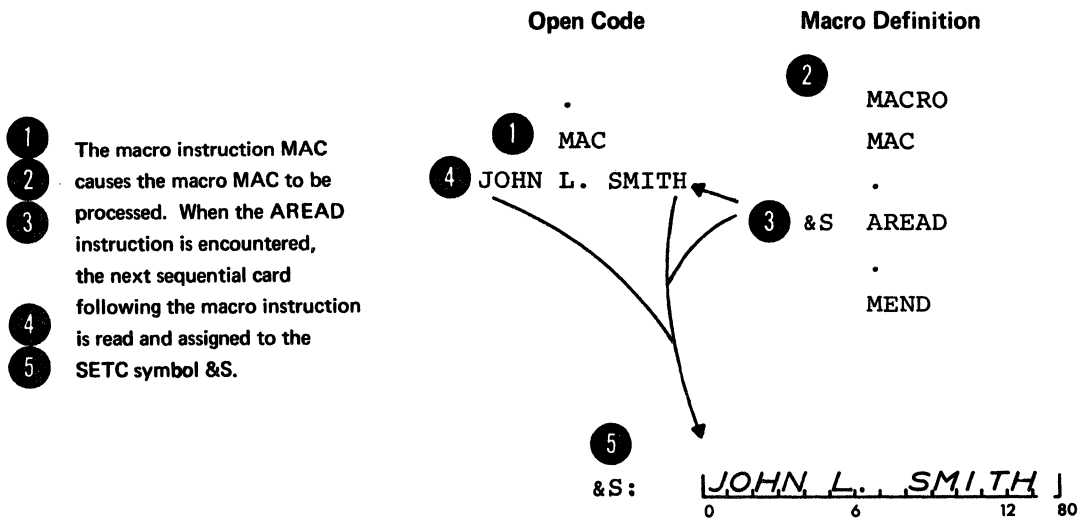
Consider the following example:

```
|Name          |Operation  |Operand
|              |           |
|              |MACRO      |
|              |MAC        |&X
|              |.          |
|              |.          |
|              |&X         |A,B,C     inner macro instruction
|              |.          |
|              |MEND       |
|              |MACRO      |
|              |MACALL     |
|              |.          |
|              |MEND       |
|              |MAC        |MACALL    outer macro instruction
```

The outer macro instruction calls the macro MAC. If the inner macro instruction is encountered during the processing of MAC, MACALL is substituted for &X, and an inner macro instruction calling MACALL is processed.

ARBITRARY LANGUAGE INPUT - AREAD

An entirely new concept is introduced by a macro language instruction that reads source deck data directly into macro processed text. The AREAD assembler operation permits a macro to "read cards" directly from the source stream into SETC variable symbols. The card image is assigned in the form of an 80-byte character string to the symbol specified in the name field of the instruction. The following illustrates how the instruction is used:



Repeated AREAD Statements read successive cards:

```
Name          Operation     Operand
              MACRO
              MAC           &N
              .
.LOOP         ANOP
&K            SETA          &K+1                Increment loop counter
&S(&K)        AREAD
              AIF           (&K LT &N).LOOP     Check loop counter
              .
              MEND
              MAC           2
JOHN L. SMITH
HECTOR S.BROWN
              END
```

The coding in this example assigns to the SETC symbol element &S(1) an 80-character-long string of JOHN L. SMITH followed by 67 blanks, and to &S(2), HECTOR S. BROWN followed by 65 blank characters.

When macro instructions are nested, the cards read by AREAD always have to follow the outermost macro instruction regardless of the level of nesting in which the AREAD instruction is found. Consider the following:

```
              MACRO
              MACIN
              .
&F            AREAD
              .
              MEND
              MACRO
              MACOUT
              .
              MACIN
THIS CARD IS NOT READ BY AREAD
              .
              MEND
              MACOUT
THIS CARD IS READ BY AREAD IN MACIN
```

If the macro instruction containing the AREAD instruction is found in code included by the COPY instruction, source cards are read from the code brought in by the COPY instruction until end of file is reached, then from the input stream.

NOTE: Cards that are read in by the AREAD instruction are not checked by the assembler. Therefore, no diagnostic will be issued if your AREAD statements read cards that are meant to be part of your source program. For example, if a macro containing an AREAD statement appears immediately before the END instruction, the END instruction is lost to the assembler.

LISTING OPTIONS: Normally the AREAD input cards are printed in the assembler listing and assigned statement numbers. However, if you do not want them printed or assigned statement numbers you can specify NOPRINT or NOSTMT in the operand of the AREAD instruction.

AREAD/PUNCH INPUT/OUTPUT CAPABILITY: The AREAD facility complements the PUNCH facility to provide macros with direct I/O capability. The total I/O capability of macros can be described as follows:

Implied Input:	Parameter values and global SET symbol values that are passed to the macro.
Implied Output:	Generated statements passed to the assembler for later processing and global SET symbol values set for use in other macros or open code.
Direct Input:	AREAD.
Direct Output:	MNOTE for printed messages; PUNCH for punched cards.

For example, you can use AREAD and PUNCH to write card conversion programs. The following macro interchanges the left and right halves of cards placed immediately after a macro instruction calling it. End of input is indicated with the word FINISHED in the first columns of the last card in the input to the macro.

```
Name      Operation  Operand
          MACRO
          SWAP
.LOOP     ANOP
&CARD     AREAD
          AIF        ('&CARD'(1,8) EQ 'FINISHED').MEND
&CARD     SETC       '&CARD(41,40).'&CARD'(1,40)
          PUNCH      &CARD
          AGO        .LOOP
.MEND     MEND
```

MULTILEVEL SUBLISTS IN MACRO INSTRUCTION OPERANDS

Multilevel sublists (sublists within sublists) are permitted in macro instruction operands and in the keyword default values in prototype statements, as shown in the following:

```
MAC1    (A,B,(W,X,(R,S,T),Y,Z),C,D)
MAC2    &KEY=(1,12,(8,4),64)
```

The depth of this nesting is limited only by the constraint that the total length of an individual operand cannot exceed 255 characters.

To access individual elements at any level of a multilevel operand you use additional subscripts after &SYSLIST or the symbolic parameter name. For example, if &P is the first positional parameter and the value assigned to it 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
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
```

REDEFINING CONDITIONAL ASSEMBLY OPERATION CODES

Under Assembler H, you can use the OPSYN instruction to redefine operation codes anywhere in your source module. The new definitions of operation codes then remain in effect for all subsequent statements, including those generated from macros. However, the definitions of conditional assembly statements are fixed when the macro definition is edited. Thus, OPSYN statements placed after a definition of a macro have no effect on the conditional assembly statements of that macro, if it is called later in the source code. Consider the following example:

```
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
```

In this example, AIF and MVC instructions are used in a macro definition. OPSYN statements are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC. In subsequent generations of the macro involved, AIF is still defined as an AIF operation, and MVC is treated as an MVI operation. In open code following the macro call, the operations of both instructions are derived from their new definitions assigned by the OPSYN statements. If the macro is redefined (by another macro definition), the new definitions of AIF and MVC (that is, AGO and MVI) are fixed for any further expansions.

NOTE: Because the assembler does not edit inner macro definitions until it encounters them during the processing of a macro instruction calling

the outer macro, this description does not apply to nested macro definitions. An OPSYN statement placed before the outer macro instruction will affect conditional assembly statements in the inner macro definition.

OTHER EXTENSIONS

The following rules apply to further language extensions of Assembler H relative to the VS Assembler:

- Macro names, variable symbols (including the ampersand), and sequence symbols (including the period), can be a maximum of 63 alphameric characters. The first character, excluding ampersands and periods, must be alphabetic.
- Internal comments (.*) can be inserted between the macro header and the prototype. Such comments are not printed with the generation of the macro.
- Any mnemonic operation code of the IBM System/360 and 370 Standard Instruction Set or any assembler operation code can be defined as a macro instruction. When any of the operation codes is redefined as a macro instruction, subsequent use is interpreted as a macro call.
- Any instruction, except ICTL, is permitted within a macro definition.

Extensions to Conditional Assembly Instructions

The flexibility of the AIF, AGO, SETA, SETB, and SETC instructions is increased in Assembler H. In Assembler H, multiple AIF statements can be merged in one AIF statement, the AGO statement has an expanded interpretive function, and a single SETx instruction (SETx is either SETA, SETB, or SETC) can assign values to more than one element of a SET symbol array. Format and ordering restrictions are also revised, and a new system variable symbol is introduced. In addition, generated statements have new functions, and the availability of symbol attributes is increased.

EXTENDED AGO STATEMENTS

In Assembler H, one AGO instruction can contain computed branch sequence information. The extended AGO statement has the following format:

Operation	Operand
AGO	(k).S1,.S2,.....,.Sn

where "k" is a SETA arithmetic expression. If the value of "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. The statement is exactly equivalent to the following sequence of AIF instructions:

Operation	Operand
AIF	(arithmetic expression EQ 1) .S1
AIF	(arithmetic expression EQ 2) .S2
.	
.	
.	
AIF	(arithmetic expression EQ n) .Sn

EXTENDED AIF STATEMENTS

The AIF statement in Assembler H can include a string of logical expressions and related sequence symbols. There is no limit to the number of expressions and symbols that you can use in an extended AIF statement. The format is:

Operation	Operand	Column 72
AIF	(logical expression) .S1,	X
	(logical expression) .S2,	X
	....., (logical expression) .Sn	

This is equivalent to "n" successive AIF statements. The branch is taken to the first sequence symbol (scanning left to right) that corresponds to a true logical expression. If none of the logical expressions is true, control passes to the next sequential instruction.

EXTENDED SETx STATEMENTS

The SETA, SETB, and SETC statements are used to assign arithmetic, binary, and character values, respectively to SET variable symbols. In Assembler H, you can use the SET statement to assign lists, or arrays, of values to subscripted SET symbols. In the VS Assembler, a separate SETx statement is required for each element of an array. For example, a list of 100 SETx values requires 100 SETx statements. In Assembler H, such a list can be coded in one extended SETx statement. The extended SETx statement has the following format:

Name	Operation	Operand
&SYM(k)	SETx	X1,X2,,X4,.....,Xn

The form of the name and operation fields is the same as that used in the VS Assembler for assignment of a dimensioned variable SET symbol: &SYM is a dimensioned SET symbol, "k" is a SETA arithmetic expression, and SETx is SETA, SETB, or SETC. Each of the operands ("Xn") has the form of an ordinary SETx operand, or may be omitted. Whenever an operand is omitted, the corresponding element of the dimensioned variable SET symbol (&SYM) is left unchanged.

When none of the operands is omitted, the extended SETx statement is equivalent to the following sequence of statements:

```
┌──────────────┬──────────┬──────────────────────────────────────────┐
│Name          │Operation │Operand                                   │
│              │          │                                          │
├──────────────┼──────────┼──────────────────────────────────────────┤
│&SYM(k)       │SETx      │X1                                        │
│&SYM(k+1)     │SETx      │X2                                        │
│.             │          │                                          │
│.             │          │                                          │
│.             │          │                                          │
│&SYM(k+n-1)   │SETx      │Xn                                        │
└──────────────┴──────────┴──────────────────────────────────────────┘
```

Following are examples of the use of extended SETx statements:

1.
```
&X(3)       SETA    3,,5,,7
```

This is equivalent to the sequence:

```
&X(3)       SETA    3
&X(5)       SETA    5
&X(7)       SETA    7
```

2.
```
&X(1)       SETA    1,&X(1)+1,&X(2)+1
```

This is equivalent to the sequence:

```
&X(1)       SETA    1
&X(2)       SETA    2
&X(3)       SETA    3
```

3.
```
&Y(1)       SETC    '',,''
```

This sets &Y(1) and &Y(3) to null values and leaves &Y(2) unchanged.

SET SYMBOL FORMAT AND DEFINITION CHANGES

Extensions in Assembler H to SETx statements, and local and global definition statements, are discussed in the following list.

- In Assembler H, global and local SET symbol declarations are processed at generation time in the assembly process, not edit time as in the VS Assembler. Either a macro definition or open code can contain more than one declaration for a given SET symbol, as long as only one is encountered during a given macro expansion or conditional assembly of open code.

- A SET symbol that has not been declared in a LCLx or GBLx statement is implicitly declared by appearing in the name field of a SETx statement. Such a declaration is interpreted as local, with the type determined by the SETx operator, and the dimensionality is determined by the occurrence of a subscript in the name field. Any explicit declaration encountered thereafter is flagged as a duplicate declaration. Undeclared SET symbols are not valid in the VS Assembler; they cause error messages to be generated.

- A SET symbol can be defined as an array of values by adding a subscript after it, when it is declared, either explicitly or implicitly. Under the VS Assembler, the subscript specified in the declaration determines the maximum number of elements that the SET symbol array can contain. Under Assembler H, however, all SET symbol arrays are open-ended; the subscript value specified in the declaration does not limit the size of the array. Thus the following set of statements is allowed under Assembler H but not under the VS Assembler:

```
|Name       |Operation |Operand                                          |
|           |          |                                                 |
|           |LCLA      |&J(50)                                           |
|&J(45)     |SETA      |415          Allowed under both assemblers       |
|&J(89)     |SETA      |38           Allowed only under H                |
```

CREATED SET SYMBOLS

Assembler H allows SET symbols to be created during the generation of a macro. A created SET symbol has the form &(e) where "e" represents one or more of the following:

- Variable symbols, optionally subscripted
- Strings of alphameric characters
- Created SET symbols

After substitution and concatenation, "e" must consist of a string of 1 to 62 alphameric characters, the first being alphabetic. This string is then used as the name of a SETx variable. For example:

```
|Name       |Operation |Operand                                          |
|           |          |                                                 |
|&Y(1)      |SETC      |'A1','A2','A3','A4'                              |
|&(&Y(3))   |SETA      |5                                                |
```

These statements have an effect similar to: &A3 SETA 5.

Created SET symbols can be used wherever ordinary SET symbols are permitted, including declarations; they can even be nested in other created SET symbols. The following nested variable could generate a valid created SET symbol:

```
&(&(&X(&(&Y))))
```

The created SET symbol can be thought of as a form of indirect addressing. Thus, in the first example above, &Y is a variable whose value is the name of the variable to be updated. With nested created SET symbols, you can get such indirect addressing to any level.

Created SET symbols can also 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" element of the symbol substituted for &SYM.

A related application is illustrated in the following macro definition. This macro is designed to push an item into the specified push-down stack. A new stack is created for each new stack name given as a parameter in the macro call. Note that &LIST becomes as long as required.

```
                        MACRO
                        PUSHDOWN  &STAK,&ITEM
                        GBLA      &(&STAK)(1),&(&STAK.SIZE)
&(&STAK.SIZE)           SETA      &(&STAK.SIZE)+1
&(&STAK)(&(&STAK.SIZE)) SETA      &ITEM
                        MEND
```

The macro call "PUSHDOWN LIST,25" is logically equivalent to:

```
                        GBLA      &LIST(1),&LISTSIZE
LISTSIZE                SETA      &LISTSIZE+1
LIST(&LISTSIZE)         SETA      25
```

Created SET symbols also enable you to get some of the effect of multidimensional arrays by creating a separate named item for each element of the array. For example, a three-dimensional array of the form &X(&I,&J,&K) can be addressed as &(X&I.$&J.$&K). Then &X(2,3,4) would be represented as a reference to the symbol &X2$3$4.

Note that what is being created here is a SET symbol. Both creation and recognition occur at macro generation time. In contrast, parameters are recognized and encoded (fixed) at macro edit time. Consequently, if a created SET symbol name happens to coincide with a parameter name, the fact is ignored, and there is no interaction between the two.

USING SETC VARIABLES IN ARITHMETIC EXPRESSIONS

You can use a SETC variable as an arithmetic term if its character string value represents a valid self-defining term. A null value is treated as zero. A subset of this facility is available in the VS Assembler, which allows character strings that represent decimal self-defining terms.

This expanded facility in Assembler H allows you to associate numeric values with EBCDIC or hexadecimal characters; this can be used for such applications as indexing, code conversion, translation or sorting.

For example, the following set of instructions converts a hexadecimal value in &X into the decimal value 243 in &VAL.

Name	Operation	Operand
&X	SETC	'X''F3'''
&VAL	SETC	&X

ATTRIBUTE REFERENCES

Under the VS Assembler, attributes of symbols in generated statements are never accessible. Under Assembler H, attributes of symbols produced by macro expansion or substitution in open code are available immediately after the statement referenced is generated.

Forward Attribute References

If an attribute reference is made to a symbol that has not yet been encountered, the assembler scans the source code either until it finds the referenced symbol in the name field of a statement in open code, or until it reaches the end of the source module. The assembler makes entries for the symbol, as well as any other not previously defined symbols it encounters during the scan, in the symbol table. The assembler does not completely check the syntax of the statements for which it makes entries in the symbol table. Therefore, a valid attribute reference may result from a forward scan, even though the statement is later found to be in error and therefore not accepted by the assembler. Further, you must be careful with the contents of any AREAD input in your source module. If the first word of an AREAD input card conflicts with an attribute reference, the forward scan will attempt to evaluate that card instead, if it appears before the "true" symbol.

Attribute References Using SETC Variables

The symbol referenced by an attribute reference of type length (L'), type (T'), scaling (S'), integer (I'), and defined (D', see below) can only be an ordinary symbol. The name of the ordinary symbol can, however, be specified in three different ways:

- the name of the ordinary symbol itself.
- the name of a symbolic parameter whose value is the name of the ordinary symbol.
- the name of a SETC symbol whose value is the name of the ordinary symbol.

Consider the following examples:

```
┌------------┬-----------┬-----------------------------------------┐
|Name        |Operation  |Operand                                  |
|            |           |                                         |
├------------┼-----------┼-----------------------------------------┤
|&F          |SETC       |T'ORDSYM                                 |
|ORDSYM      |DC         |H'3'                                     |
└------------┴-----------┴-----------------------------------------┘
```

In this example, the symbol in the attribute specification (T'ORDSYM) is the ordinary symbol itself.

```
|Name       |Operation |Operand                                    |
|           |          |                                           |
|-----------|----------|-------------------------------------------|
|&K         |SETC      |'ORDSYM'                                   |
|&F         |SETC      |T'&K                                       |
|ORDSYM     |DC        |H'3'                                       |
```

In this example, however, the symbol in the attribute reference (T'&K) is a variable symbol whose value is the name of the referenced symbol (ORDSYM). The type attribute in both examples will be the type attribute of the DC instruction named ORDSYM.

Under the VS Assembler, you can only use ordinary symbols and symbolic parameters in attribute specifications of types L', T', S', and I'.

The Defined Attribute (D')

The defined attribute (D') is introduced in Assembler H. It can be used in conditional assembly statements to determine if a given symbol has been defined at a prior point in the source module. If the symbol is already defined, the value of the defined attribute is one; if it has not been defined, the value is zero. By testing a symbol for the defined attribute, you can avoid a forward scan of the source code.

Number Attributes for SET Symbols

The number attribute can be applied to SETx variables to determine the highest subscript value of a SET symbol array to which a value has been assigned in a SETx instruction. For example, if the only occurrences of the SETA symbol &A are:

```
|Name       |Operation |Operand                                    |
|           |          |                                           |
|-----------|----------|-------------------------------------------|
|&A(1)      |SETA      |0                                          |
|&A(2)      |SETA      |0                                          |
|&A(3)      |SETA      |&A(2)                                      |
|&A(5)      |SETA      |5                                          |
|&A(10)     |SETA      |0                                          |
```

then N'&A is 10.

The number attribute is zero for a SET symbol that has not been assigned any value. In the VS Assembler, the N' attribute can only refer to symbolic parameters.

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 character in column 72. Remarks are inserted

optionally between the blank and column 72. The last line terminates the series with a blank in column 72.

In the VS Assembler, alternate format can be used only in macro prototype and macro call statements. In Assembler H, the extended AGO, extended AIF, GBLx, LCLx, and extended SETx statements can also be written in alternate format, as shown in the following examples:

```
|Name   |Operation |Operand              |Remark
|-------|----------|---------------------|-----------------------------
|       |AGO       | (&A).S1,            |remark   X
|       |          | .S2,.S3,            |         X
|       |          | .S4                 |
|       |          |                     |
|       |AIF       | (&L1).S1,           |remark   X
|       |          | (&L2).S2,           |         X
|       |          | (&L3).S3            |
|       |          |                     |
|       |GBLA      | &A1,                |         X
|       |          | &B(5)               |
|       |          |                     |
|       |LCLC      | L1,                 |remark   X
|       |          | L2,L3,              |remark   X
|       |          | L4                  |remark
|       |          |                     |
|&B(1)  |SETB      | 0,                  |remark   X
|       |          | (&A NE 3),          |remark   X
|       |          | ('SC' EQ 'XYZ')     |
```

NEW SYSTEM VARIABLE SYMBOL

System variable symbols are local variable symbols that are assigned values by the assembler when they are encountered. A new system variable symbol is provided in Assembler H for use in macro definitions.

&SYSLOC: &SYSLOC is identical in function to &SYSECT, except that its value is the character string that represents the location counter (as controlled by the LOCTR statement) that is in effect at the time the macro is called. &SYSECT gets the value of the current CSECT, DSECT, or COM section. If no LOCTR statement is in effect, the value of &SYSLOC is the same as the value of &SYSECT. &SYSLOC can be used only in macro definitions. The LOCTR instruction is described in the section of this manual entitled "Changes to Program Sectioning and Linking Controls."

For example, when the following statements occur in a source program, &SYSLOC will have the character string value XYZ during expansion of MAC1.

```
|Name        |Operation |Operand
|------------|----------|-----------------------------------------
|            |MACRO     |
|            |MAC1      |
|&C          |SETC      |'&SYSLOC'
|            |          |
|            |MEND      |
|XYZ         |LOCTR     |
|            |MAC1      |
```

Extensions to Basic Assembler Language

This section covers the extensions to the basic assembler language supported by Assembler H.

Revised Assembler Operations

Several assembler operations used in the VS Assembler are extended in Assembler H. The revised operations are described in the following.

OPSYN INSTRUCTION EXTENSION

You can code OPSYN instructions anywhere in your source module. This offers an advantage over the VS Assembler which only allows OPSYN statements at the beginning of source modules.

EQU INSTRUCTION EXTENSION

Under the VS Assembler, any symbols appearing in the first operand of the EQU instruction must be previously defined. This is not required under Assembler H. Thus in the following example, both WIDTH and LENGTH can be defined later in the source code:

```
┌----------┬---------┬----------------------------------------┐
|Name      |Operation|Operand                                 |
|          |         |                                        |
├----------┼---------┼----------------------------------------┤
|VAL       |EQU      |40-WIDTH+LENGTH,4,F                     |
└----------┴---------┴----------------------------------------┘
```

COPY INSTRUCTION EXTENSION

The VS Assembler allows COPY instructions within code that has been brought into your program by another COPY instruction. However, there is a limit of five such nestings. Under the Assembler H no such limit exists. Any number of nestings is permitted.

CNOP INSTRUCTION EXTENSION

The restriction that symbols in the operand field of a CNOP instruction must be previously defined does not exist under Assembler H.

ISEQ INSTRUCTION EXTENSION

Assembler H allows sequence checking of columns that are placed anywhere on the input cards. The VS Assembler allows checking only of columns that fall outside the statement field, that is, the columns checked cannot appear between the begin and end columns.

Assembler Language Syntax Extensions

The syntax of the assembler language deals with the structure of individual elements of an instruction statement and with the order in which the elements are presented in that statement. Several syntactical elements of the language supported by the VS Assembler are extended in the language supported by Assembler H.

CONTINUATION LINES

Assembler H allows a maximum of nine continuation lines in an ordinary assembler language statement. The VS Assembler allows only two continuation lines for ordinary assembler language statements.

The alternate format, which allows as many continuation lines as needed, is allowed for certain instructions. The VS Assembler allows the alternate format only for macro prototype statements and macro instruction statements. In addition to those instructions, the Assembler H allows the alternate format for AIF, AGO, SETx, LCLx, and GBLx instructions.

SYMBOL LENGTH

Assembler H allows a maximum of 63 characters for a symbol. This limit includes the ampersand for variable symbols and the period for sequence symbols. The VS Assembler allows only 8 characters.

Because the linkage editor does not accept symbols longer than eight characters, external symbols are limited to eight characters. External symbols are those used in the name field of START, CSECT, COM, and DXD statements and in the operand field of ENTRY, EXTRN, and WXTRN statements. Symbols used in V- and Q-type address constants are also restricted to eight characters.

LEVELS WITHIN EXPRESSIONS

Assembler H allows any number of terms or levels of parentheses in an expression. The VS Assembler allows only 35 operators including left parentheses.

Changes to Programming Sectioning and Linking Controls

Operations controlling program sectioning and linking are extended in Assembler H to allow increased freedom of program organization. A new instruction is introduced, and several instructions in the language supported by the VS Assembler are revised.

USE OF MULTIPLE LOCATION COUNTERS

The assembler instruction LOCTR allows multiple location counters to be defined within a control section during the assembly. The format of this new instruction is:

Name	Operation	Operand
Any ordinary or variable symbol	LOCTR	Blank

The assembler assigns consecutive addresses to all segments of a location counter in a control section before it continues address assignment with the first segment of the next location counter. By using the LOCTR instruction, you can cause your program object-code structure to differ from the logical order appearing in the listing. You can code sections of a program as independent logical and sequential units. For example, you can code work areas and constants within the section of code that requires them, without branching around them. Figure 2 illustrates this procedure.

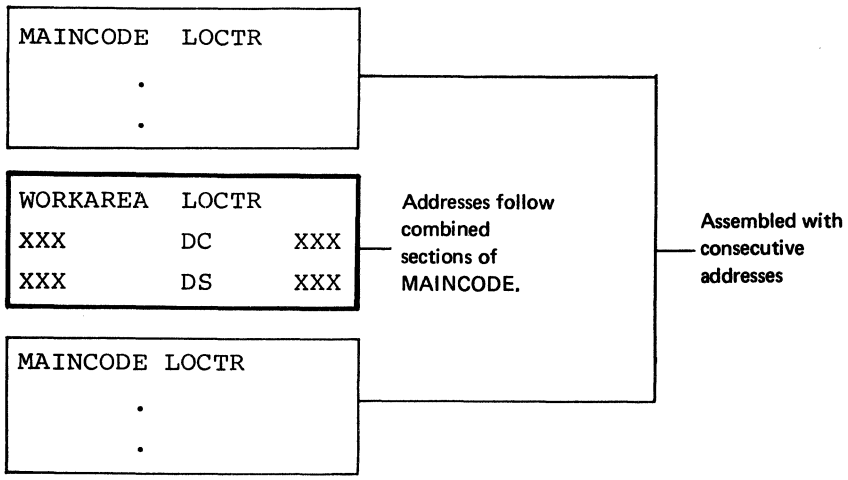


Figure 2. LOCTR Instruction Application

The following rules govern applications of the LOCTR instruction:

- A location counter can be interrupted by a CSECT, DSECT, COM, or another LOCTR instruction.
- A control section name that is defined by the CSECT, COM, DSECT, or START instruction automatically names the first location counter in that section.
- A LOCTR instruction with the same name as a control section resumes the first location counter in that section.
- A LOCTR instruction with the same name as a previous LOCTR instruction forces a return to the control section in which it was first defined and resumes the particular counter involved.
- Resumption of a control section causes resumption of the last active, not necessarily the highest valued, location counter under that control section.
- A control section name defined for the first time is in error if it is identical to a previously defined LOCTR instruction name.
- A LOCTR instruction occurring before the first control section will initiate an unnamed CSECT before the LOCTR instruction is processed.
- LOCTR instructions do not force location counter alignment.

REVISION OF Q-TYPE ADDRESS CONSTANTS

Q-type address constants reserve storage for the offset of an external dummy section. Some restrictions imposed by the language supported by the VS Assembler are relieved under Assembler H:

- DXD or DSECT names referenced in Q-type address constants do not require previous definition.
- If the relocatable symbol in a DXD statement is not used in a Q-type address constant, the DXD symbol is not placed in the external symbol dictionary (ESD). DXD statements without a Q-type address constant reference are not addressable by the program.

NUMBER OF ESD SYMBOLS

The Assembler H does not restrict the number of symbols that can be contained in the external symbol dictionary (ESD). The maximum number of entries depends on the amount of main storage available to the linkage editor.

The assembly of source modules by Assembler H is considerably faster than assembly of the same modules by the VS Assembler. The performance of Assembler H is superior primarily because the Assembler H needs fewer input/output operations and machine instructions to assemble a program. The following figures are system independent and give the ratio between the VS Assembler and Assembler H:

$$\frac{\text{Number of SIO instructions used by XF}}{\text{Number of SIO instructions used by H}} = \text{between 5 and 10}$$

$$\frac{\text{Number of machine instructions used by XF}}{\text{Number of machine instructions used by H}} = \text{between 1.5 and 1.8}$$

The actual ratio for a given program will depend on the source program itself and on the region allocated to the assemblers.

Elapsed Time Measurement

Elapsed time is relevant only when running one job at a time. The figures on elapsed time below serve only to give you an idea of how the different System/360 models affect Assembler H performance relative to the VS Assembler performance.

The following ratios are a conservative measure of elapsed time for an assembler job step when using Assembler H relative to the VS Assembler running under OS on 360 machines. The figures do not apply to very small programs because system overhead for the execution of a job step is constant for both assemblers.

		Model 40	Model 50	Model 65	Model 75	Model 85
$\frac{\text{H step-time}}{\text{XF step-time}}$	=	1:1.5	1:1.9	1:3.5	1:4.0	1:4.5

Factors Influencing Improved Performance

The following list summarizes the factors that influence the number of I/O operations and machine instructions used by Assembler H in comparison to the VS Assembler:

- Logical text stream and tables that are a result of the internal assembly process remain resident in virtual storage, whenever possible, throughout the assembly.
- Two or more assemblies can be performed under the control of one set of job control language (JCL) cards.

- Assembler H edits only the macro definitions that it encounters during a given macro expansion or during conditional assembly of open code, as controlled by AIF and AGO statements.

- Source-text assembly passes are consolidated. The edit and expansion of macro text is done on a demand basis in one pass of the source text, instead of two distinct passes as in the VS Assembler, as shown in Figure 1.

RESIDENT TABLES AND SOURCE TEXT: Performance over the VS Assembler is improved by keeping intermediate text, macro-definition text, dictionaries, and symbol tables in main storage whenever possible. This reduces the I/O time required by assemblers that rely heavily on secondary storage throughout the assembly process. Less I/O reduces system overhead and frees channels and I/O devices for other uses.

Certain portions must remain in virtual storage throughout the assembly process. The symbol table must remain resident, and it has no overflow capacity. Also, all partially filled blocks of text must remain resident.

MULTIPLE ASSEMBLY: Multiple or batch assemblies can be done under the control of a single set of JCL cards. Source decks are placed together with no intervening "/*" card.

Batch assembly improves performance by eliminating job and step overhead for each assembly. It is especially advantageous for processing related assemblies such as a main program and its subroutines.

MACRO-EDITING PROCESS: New methods of macro processing improve performance. The VS Assembler edits all source-program macro definitions and library macro definitions that are contained or referenced in the source program. This often results in editing macro definitions that are never called. Assembler H edits only those macro definitions that are encountered during a given macro expansion or during conditional assembly of open code, as controlled by AIF and AGO statements.

A good example of potential savings by this feature is the process of system generation. During system generation, Assembler H edits only the set of library macro definitions that are expanded, whereas the VS Assembler edits all library macro definitions referenced in the system-generation source stream and the library macro definitions for all inner macro calls (to any level). As a result, Assembler H may edit 25 percent fewer library macro definitions than the VS Assembler.

CONSOLIDATING SOURCE TEXT PASSES: In comparison to the VS Assembler, consolidating assembly source text passes and other new organization procedures reduce by approximately 30 percent the number of internal processor instructions used to handle source text in Assembler H. This is represented in proportionate savings in CPU time. The saving is independent of the size or speed of the system CPU involved; it is a measure of the relative efficiency of the processor.

The Assembler H has many diagnostic features to aid in the location and analysis of program errors. Refinement of macro and conditional assembly diagnostics is particularly significant. This section describes the diagnostic facilities offered by Assembler H.

Diagnostic Extensions in Regular Assembly

ERROR MESSAGES

Assembler H prints error messages in line in the listing and includes at the end of the listing a total of the errors and a table of their line numbers. Certain in-line messages include a copy of the segment of the statement that is in error. Thus, error conditions are spelled out as they occur with direct reference to a specific error. The following illustrates this.

```
                                CSECT
                                •
                                COMM
***ERROR*** UNDEFINED OP CODE- COMM
                                •
                                DS      (*+5)F
***ERROR*** RELOCATABILITY ERROR - (*+5)F
                          1NAME DC      F'O'
***ERROR*** SYMBOL TOO LONG, OR 1ST CHARACTER NOT A LETTER - 1NAME
                                •
                           &C   SETC    'AGO'
                                &C      .X
***ERROR*** OP CODE NOT ALLOWED TO BE GENERATED - AGO
                                •
                                END
```

Diagnostic Messages in Macro Assembly

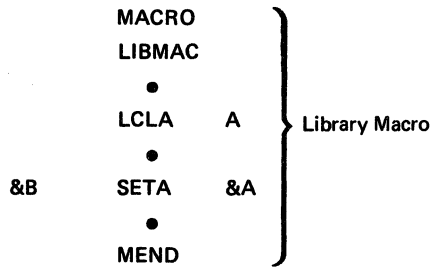
In Assembler H, diagnostic messages printed in macro-generated text are much more descriptive than those in the VS Assembler. In addition, the macro level and the statement number of the macro definition are printed for each programmer macro instruction. The macro level and the first five characters (or fewer) of the macro name are printed for library macro expansions.

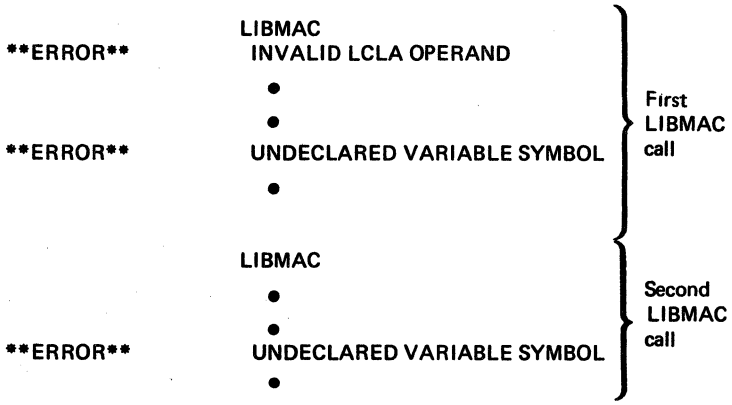
SEQUENCE FIELD IN MACRO-GENERATED TEXT: When a library macro definition is processed as a result of a macro call, the sequence field (columns 73 through 80) of the generated statements contains the level of the macro call in the first two columns, a hyphen in the third column, and the first five letters of the macro-definition name in the remaining five columns. When a line is generated from a source-program macro or a copied library macro, the last five columns contain the line number of the model statement in the definition from which the generated statement is derived. This information can be an important diagnostic aid when analyzing output dealing with macro calls within macro calls.

FORMAT OF MACRO GENERATED TEXT: Whenever possible, a generated statement is printed in the same format as the corresponding macro-definition (model) statement. The starting columns of the operation, operand, and comments fields are preserved unless they are displaced by field substitution, as shown in the following example:

```
Source Statements:        &C   SETC    'ABCDEFGHIJK'
                          &C   LA      1,4
Generated Statement:      ABCDEFGHIJK  LA  1,4
```

ERROR MESSAGES FOR A LIBRARY MACRO DEFINITION: Format errors within a particular library macro definition are listed directly following the first call of that macro. Subsequent calls on the library macro do not result in this type of diagnostic. If the appropriate option of the PRINT instruction is in effect, errors arising in the generated text of a library macro are listed in line within the generated text. The following shows the placement of error messages.





ERROR MESSAGES FOR SOURCE PROGRAM MACRO DEFINITIONS: Macro definitions contained in the source program are printed in the listing, provided that the appropriate PRINT options are in effect. Edit diagnostics are inserted in line in the listing directly following the statement in error. Errors analyzed during macro generation produce messages in line in the generated text.

ERROR MESSAGES IN MACRO-GENERATED TEXT: Diagnostic messages in generated text generally include:

- A description of the error.
- The recovery action.
- The model statement number at which the error occurred.
- A SET symbol name, parameter number, or value string associated with the error.

Macro Trace Facility (MHELP)

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 continuously until superseded by another MHELP statement. MHELP options are:

MACRO CALL TRACE: (MHELP B'1' or MHELP 1). This option provides a one-line trace for each macro call, giving the name of the called macro, its nested depth, and its &SYSNDX (total number of macro calls) value. This trace is provided upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

MACRO BRANCH TRACE: (MHELP B'10', or MHELP 2). This option provides a one-line trace for each AGO and true AIF conditional assembly statement within a macro. It gives the model statement numbers of the "branched from" and "branched to" statements, and the name of the macro in which the branch occurs. This trace option is suppressed for library macros.

MACRO ENTRY DUMP: (MHELP B'10000', or MHELP 16). This option dumps parameter values from the macro dictionary immediately after a macro call is processed.

MACRO EXIT DUMP: (MHELP B'1000', or MHELP 8). This option dumps SET symbol values from the macro dictionary upon encountering a MEND or MEXIT statement.

MACRO AIF DUMP: (MHELP B'100', or MHELP 4). This option dumps SET symbol values from the macro dictionary immediately before each AIF statement that is encountered.

GLOBAL SUPPRESSION: (MHELP B'100000', or MHELP 32). This option suppresses global SET symbols in the two preceding options, MHELP 4 and MHELP 8.

MHELP SUPPRESSION: (MHELP B'10000000', or MHELP 128). This option suppresses all currently active MHELP options.

MHELP CONTROL ON &SYSNDX: MHELP operands are assembled into a signed fullword. See the sample hexadecimal values for MHELP operand in Figure 3.

MHELP OPERAND	Hexadecimal Value &SYSNDX		MHELP	MHELP Effect
		MHELP		
4869	0000	13 05		Macro call trace, Macro AIF dump; &SYSNDX 4869
65536	0001	00 00		No effect
16777232	0100	00 10		Macro entry dump
28678	0000	70 06		Macro branch trace, Macro AIF dump; &SYSNDX 28678

Figure 3. MHELP Control on &SYSNDX

MHELP and &SYSNDX values are determined according to the following rules:

1. If there are any non-zero bits present in the low-order six bits of the MHELP field (see Figure 3), the corresponding options are turned on.

2. &SYSNDX values are set only if the &SYSNDX field has any non-zero bits in it. The value of &SYSNDX is the value of the entire fullword. If the &SYSNDX field contains only zeros, the &SYSNDX value is not set, even if there is a value in the high-order bytes to the left of the &SYSNDX field.

When &SYSNDX (the 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.

When the value of the &SYSNDX reaches its limit, a diagnostic message is issued.

COMBINING OPTIONS: 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.

Abnormal Termination of Assembly

The assembler produces a specially formatted dump whenever an assembly cannot be completed. This may help you in determining the nature of the error. The dump is also useful if the abnormal termination was caused by an error in the assembler itself.

Index

&

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

Part II: For the MFT or MVT User

Preface

This publication is an introduction to the OS Assembler H. It describes:

- Basic structure and features of Assembler H.
- System requirements and the operating environment of Assembler H.
- Additions and extensions to the basic assembler language support by Assembler H.
- Extensions to the macro and conditional assembly facilities supported by Assembler H.
- Estimates of Assembler H performance.
- Additions and extensions to the assembler language diagnostic services supported by Assembler H.

To use this publication effectively, you should be familiar with the OS Introduction, Order Number GC28-6534, or have the equivalent knowledge. 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:

OS Assembler Language, Order Number GC28-6514

Other publications containing pertinent information are the following:

OS Assembler H Language, Order Number GC26-3771.

The Assembler H Language manual tells you the instructions available to program with Assembler H. It is supplemental to the Assembler Language manual above.

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 Job Control Language Reference, Order Number GC28-6704

The Job Control Language manual tells you 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

Figures

Introduction

The OS Assembler H is an assembler language processor with major extensions to the language and performance of OS Assembler F.

Language Compatibility

Any program successfully assembled with no warning or diagnostic message by Assembler F will assemble correctly with Assembler H. If Assembler F restrictions have been used to get a certain result, Assembler H might give a different result. Programs which use features supported only on Assembler H will not assemble correctly with Assembler F.

Language Changes

Many limitations of the F-level assembler language are relaxed or eliminated in the H-level assembler language. Changes and additions to the language fall into the following categories.

- Conditional assembly instructions have expanded functions and flexibility.
- Many assembler instructions have fewer restrictions and allow improved programmer control. New assembler instructions are also added to the language.
- Branch-on-condition register (BCR) extended mnemonic operation codes are added to the assembler instruction set.

Performance

The high-speed assembly capability of Assembler H is a result of the following factors:

- All text processing is performed in main storage if the assembly region is sufficiently large.
- Assembler source-text passes are consolidated. This reduces the number of internal processor instructions used for text handling.
- Multiple, or batch, assemblies can be performed under the control of one set of job control language (JCL) cards.
- Macro definitions referenced in the source text are not automatically edited; they must be called or encountered in the assembly process.

System Requirements

Central Processing Units: Assembler H operates on an IBM System/360 Model 40 or higher, and on an IBM System/370 Model 135 or higher.

Operating Environments: Assembler H operates under OS. It can operate under two types of control program. The following list shows the two control programs and the minimum central processing unit (CPU) size required by each.

- Multiprogramming with a Fixed number of Tasks (MFT) - 384K byte CPU.
- Multiprogramming with a Variable number of Tasks (MVT) - 384K byte CPU.

Main Storage: Recommended minimum region size of the Assembler H is 180K bytes of main storage.

Data Sets: System input and output device requirements are the same as those of Assembler F except that the number of workfiles is reduced to one direct-access storage device. In terms of the IBM 2314 Direct Access Storage Facility, or an equivalent device, cataloged procedures for Assembler H require a maximum of one track on SYS1.PROCLIB, and load modules need approximately 30 tracks on SYS1.LINKLIB, or the user-defined library.

Internal Design

The internal organization of Assembler H is an entirely new design. There are only two source-text passes, as opposed to four for Assembler F. Pass one of the source text by Assembler H edits and expands macros, and builds dictionaries and the symbol table. Pass two completes the assembly and produces the desired output. Figure 1 shows the general processing steps of Assembler F and Assembler H.

RESOLVING SYMBOL ATTRIBUTE REFERENCES

The symbol table is built as symbols are encountered in macro generation and open-code assembly. If an attribute reference is made to a previously undefined symbol, Assembler H proceeds with a forward scan of the source text, called "lookahead" mode. It continues the forward scan until the symbol that initiated the scan is resolved or until the end of the source program is encountered. During the scan, the assembler conditionally places all other symbols that are encountered into the symbol table, so further forward scans are avoided unless a forward reference is made to a symbol at some point beyond the previous forward reference. The symbol attributes established by the forward scan are not fixed, however, and can be overridden when the symbol is placed in the symbol table as a result of regular assembly. If the symbol that initiated the forward scan is not found, a diagnostic message is issued.

INTERNAL TEXT PROCESSING

Assembler F cannot make significantly more efficient use of region sizes larger than its original design criterion of 44K bytes, because it is designed to process intermediate assembly text using external workfiles. Assembler H processes all intermediate assembly text internally in working storage if the assembly region will contain the text. If a source program is sufficiently large, Assembler H will use any additional amount of main storage that can be allocated as a region, up to the maximum region size of a system.

Within the region allocated for an assembly of a source program, the amount of working storage Assembler H uses to perform an operation can dynamically expand to meet the storage requirements of that operation.

When one block of working storage is filled with processed text, processing continues with the allocation of another block from within the region acquired for the assembly.

As blocks of working storage are filled with processed text, they are flagged to indicate whether they can be written out to the external workfile (SYSUT1). Partially filled blocks and those blocks taken up by the symbol table must remain resident at all times during the assembly. Those blocks that can be written out are put on the workfile, but the blocks of text are also retained in working storage and continue to be effective for all assembly purposes. Only when all unallocated workspace within the region is exhausted are the written-out text blocks in working storage reinitialized and overlaid with newly processed text. For continuing assembly purposes, the blocks of text on the workfile must then be accessed.

Note: If all blocks of working storage are allocated and flagged as resident when an operation requires additional workspace, the assembly is terminated. Such assemblies must either be partitioned (that is, broken down into subroutines) or be assembled in a larger main-storage region.

Optional Modifications to Assembler H

There are several changes that can be made to the assembler to tailor it to fit requirements of an installation. These can be set at the time the assembler is added to the system by means of an option-setting routine provided with the assembler. This routine is only used when you wish to change the standard settings, or defaults.

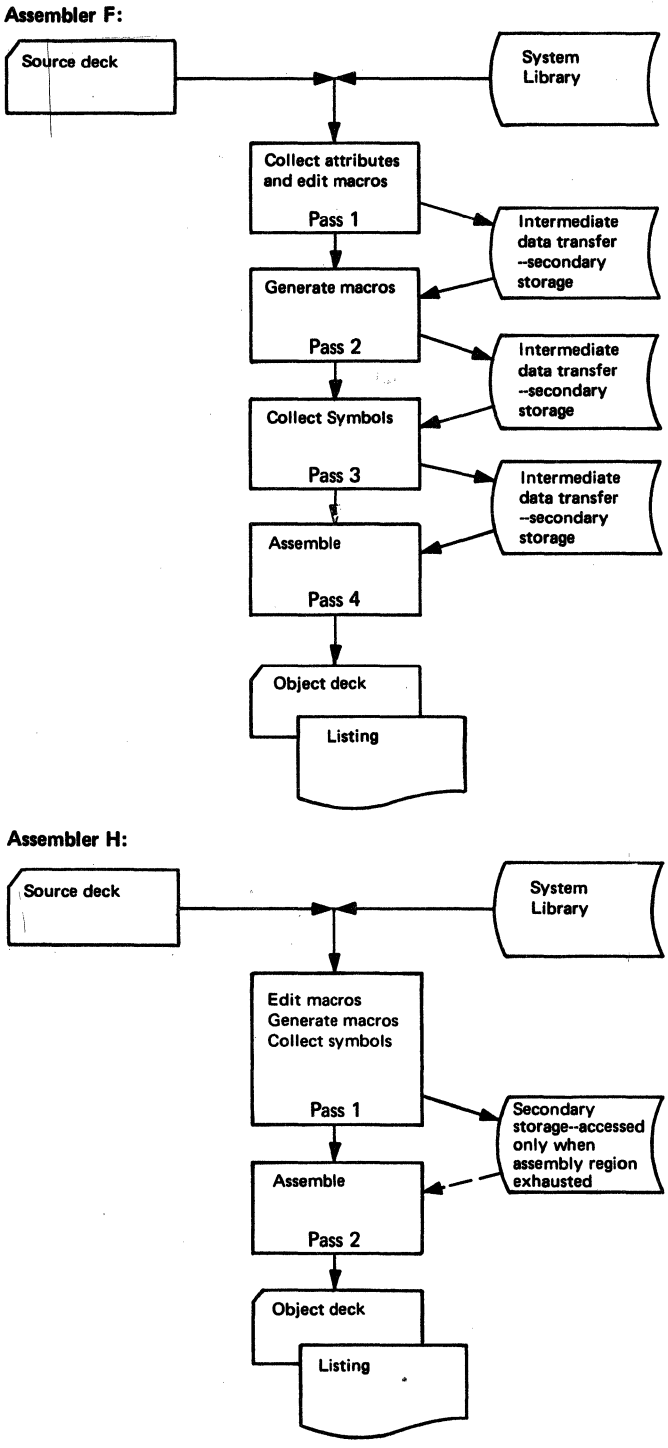


Figure 1. Processing Steps of Assembler H and Assembler F

EXECUTE (EXEC) STATEMENT DEFAULT OPTIONS

The default options in Assembler H are as follows (the alternate option is in parentheses):

DECK	(NODECK)
NOOBJECT	(OBJECT)
LIST	(NOLIST)
XREF(FULL)	(XREF(SHORT), NOXREF)
NORENT	(RENT)
NOTEST	(TEST)
NOBATCH	(BATCH)
ALIGN	(NOALIGN)
ESD	(NOESD)
RLD	(NORLD)
LINECOUNT(55)	(range 1 - 99)
FLAG(0)	(range 0 - 255)
SYSPARM() (empty)	(a character string 0 - 255 characters long)

These default options can be respecified when Assembler H is added to your system. For example, your installation may wish to modify the default options in the EXEC card as follows:

```
NODECK
RENT
LINECOUNT(40)
```

In addition to modifying the default options when adding the assembler to your system, you can delete selected options so that a programmer cannot override the options you select. If you modify the default options as immediately above and delete DECK and NORENT, then NODECK and RENT are the default options and cannot be overridden. Unless an option is deleted, however, a programmer can specify it in the PARM field of his EXEC statement.

DATA DEFINITION (DD) STATEMENT DEFAULT OPTIONS

Assembler H requires the following data set DD names:

SYSIN	
SYSLIB	(if there are library macro references)
SYSLIN	(if OBJECT is specified)
SYSPRINT	(if LIST is specified)
SYSPUNCH	(if DECK is specified)
SYSUT1	

Any of these names can be changed when the assembler is added to your system. For example, your installation may wish to replace SYSUT1 with WORK001, or even SYSUT2.

INSTRUCTION SET OPTIONS

The instruction set, or operation-code table, available to the assembler can be specified optionally. Four instructions sets are available, as shown in Figure 2.

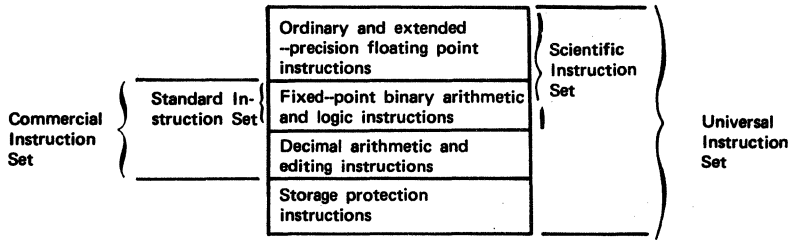


Figure 2. Instruction Set Options

Extensions to Macro and Conditional Assembly Language

The macro and conditional assembly language for Assembler H relaxes many of the language limitations of Assembler F. Many restrictions are altered or eliminated to increase flexibility and extend language functions. All ordering restrictions are removed for conditional assembly statements in macro definitions and in open code.

GENERAL ADVANTAGES IN THE USE OF MACROS

The macro definition is a subroutine that allows you to define the attributes of your data and define your manipulation of that data. Macro definitions are flexible; that is, by means of interpretive testing and substitutions, they can modify themselves internally in response to parameters that are passed to them. The macro definition is accessed, or called, by a macro instruction. When encountered by the assembler, the instruction causes execution of conditional assembly instructions and generation of assembler language instructions contained in the macro definition. By varying the values in the operand field of the macro instruction, you can generate the desired sequence of assembler language instructions. This gives the programmer strong local control of the program environment. For example, by introducing a new data test into the macro definition, you can readily adjust to a change in data type or structure. Using extensible macro language definitions has the following advantages over using basic assembler langauge subroutines:

- Macros give you a logical perspective similar to that of a high-level language. In basic assembler language, logical concepts may become lost in coding sequences that are difficult to understand, and errors may develop in coding and strategy.
- You plan the logic once and do not rethink it every time you need it. When you need it, you simply call it out. In this sense, macros become a language - a language you control and tailor to your local application.
- Macros allow you to code your own debugging aids and to set global switches. These devices can then be called out at will within your program.

Extensions to the Macro Language

The macro instruction and the macro definition have extended capabilities in Assembler H. The extensions improve programming control and coding flexibility. For example, macro definitions can be freely intermixed with open-code statements in the source-program text, and macro definitions can be nested within macro definitions. In addition, macro instructions can be redefined, macro instruction call statements can be generated by substitution, and keyword and positional parameters can be freely combined in macro instruction prototype and call statements.

MACRO DEFINITIONS IN OPEN CODE

In Assembler H, the only restriction on placement of a macro definition is that it must occur in the source text and be edited before it is called. This enables you to organize a source program into a logical text flow. In Assembler F, macro definitions must be grouped at the start of a program, preceding all statements except those pertaining to listing control, and ICTL and ISEQ instructions.

REDEFINITION OF MACRO INSTRUCTIONS

In Assembler H, a macro instruction can be redefined at any point in a source program. When a macro instruction is redefined, the new definition is effective for all subsequent expansions. Macro instructions cannot be redefined in Assembler F.

Once a macro instruction is redefined, its initial definition is lost unless, prior to redefinition, its meaning is assigned to another symbol by an OPSYN statement. For further information concerning OPSYN, refer to the section in this manual entitled "New Assembler Operations".

The initial definition can also be reestablished by returning to a point in the program prior to the first definition, and by reediting the initial definition. Both procedures are demonstrated in Figure 3.

Name	Operation	Operand	Comment
.L	ANOP		
	MACRO		macro header
	AMAC	....	macro prototype
	.		
	.		
	MEND		macro trailer
BMAC	OPSYN	AMAC	BMAC assigned definition of AMAC
	.		
	MACRO		macro header
	AMAC	....	redefine AMAC
	.		
	.		
	.		
	MEND		macro trailer
	.		
	AIF	(T'xxx EQ 'U').L	
	.		
AMAC	OPSYN	BMAC	reestablish first AMAC definition

Figure 3. Redefinition of Macro Instructions

EDITING OPERATION CODES

When conditional assembly statements within macro definitions are edited, the current definitions of their operation codes are fixed for all future expansions of the macro definition. All other model statements in the edited definition use the operation code definitions in effect whenever the macro definition is expanded (generated) as a result of a

macro call (instruction). Figure 4 illustrates this distinction between the effect of redefinition of operation codes on conditional assembly statements and on assembler language model statements within macro definitions.

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
	.		

Figure 4. Redefined Operation Codes in Macro Definitions

In Figure 4, AIF and MVC instructions are used in a macro definition. OPSYN statements are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC. In subsequent generations of the macro involved, AIF is still defined as an AIF operation, and MVC is treated as an MVI operation. In open code following the macro call, the operations of both instructions are derived from their new definitions assigned by the OPSYN statements. If the macro is reedited, the new definitions of AIF and MVC (that is, AGO and MVI) are fixed for any further expansions.

NESTING MACRO DEFINITIONS

In Assembler F, macro definitions can contain inner macro calls, but not inner macro definitions. Assembler H allows both inner macro calls and inner macro definitions. The inner macro definition is edited and bound for a particular assembly if it is encountered when the outer macro is generated. If the outer macro is not called or if the inner macro definition is not encountered in the generation of the outer macro, the inner definition is never edited. Figure 5 illustrates inner macro definition and the editing process.

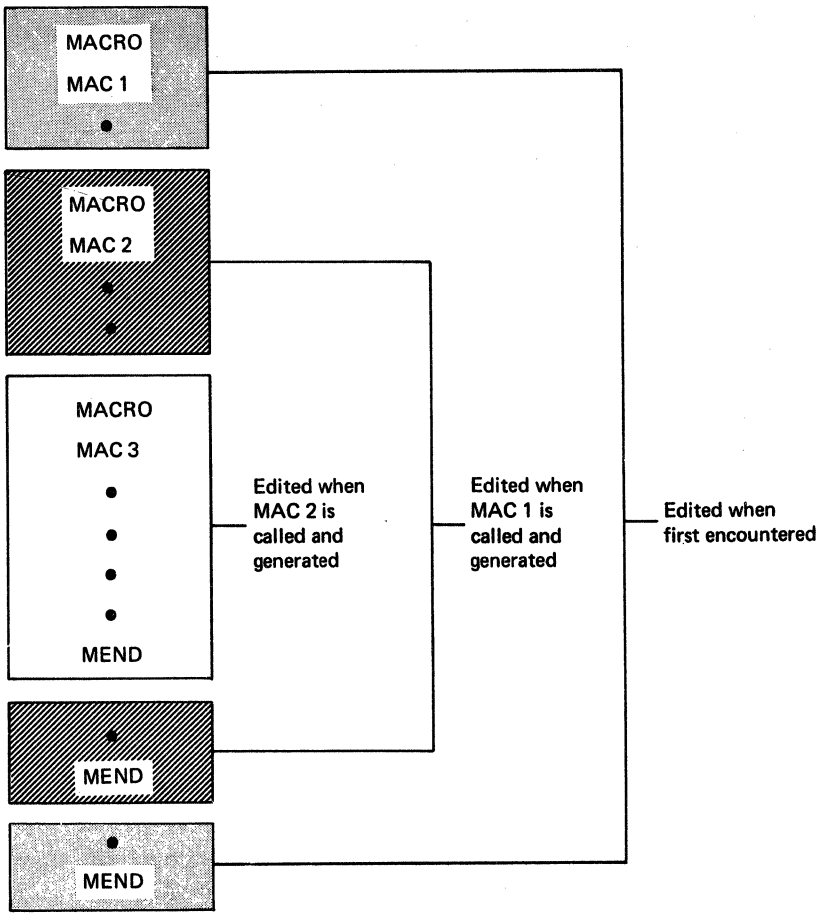


Figure 5. Editing Nested Macro Definitions

In Figure 5, MAC1 is edited, and MAC2 and MAC3 are passed over. When MAC1 is called, MAC2 is edited; and when MAC2 is called, MAC3 is edited. Until the containing (outer) macro definition is called, the contained (inner) macro definition cannot be accessed by a macro call.

MACRO CALLS BY SUBSTITUTION

In Assembler H, macro calls can be created by substitution, either in open code or as an inner macro call. For example, substitution can be performed by including a macro instruction in the operand field of a macro call. This is demonstrated in the following example:

Name	Operation	Operand	Comment
	MACRO		
	MAC	&X	
	.		
	.		
	AIF	..	
	AGO	..	
	.		
	.		
	&X	A,B,C	
	.		
	MEND		
	MAC	MACALL	macro call
	.		

If the statement "&X A,B,C" is encountered in the generation of MAC, MACALL is substituted for &X. If MACALL is a macro name, the macro definition is expanded inline. If MACALL has been made a machine operation through the use of OPSYN, it is processed by the assembler. Otherwise, it is an undefined operation and a diagnostic is issued.

The following operation codes cannot 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	

ARBITRARY LANGUAGE INPUT - AREAD

An entirely new concept is introduced by a new macro language instruction that reads source-deck data cards directly into macro-generated text. The AREAD assembler operation permits a macro instruction to "read cards" directly from the source-text input stream by putting 80-character images into SETC symbols; that is, it permits statements in an arbitrary language to occur in the assembler input stream, and to be read and processed by a macro at assembly time. AREAD cannot be used in open code. The format of AREAD is:

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

If neither option is selected, the card is printed following the macro call which contains the AREAD statement, and the printed card is assigned the next sequential statement number. If the NOSTMT option is selected, the card is printed, but no statement number is assigned. If the NOPRINT option is selected, the card is not printed, and no statement number is assigned. The name field can contain any SETC variable (optionally subscripted). If the variable is not previously declared, the AREAD statement serves as a default SETC declaration. In this sense, AREAD has the logical effect of the statement:

Name	Operation	Operand
&X	SETC	"card image"

If there are no more source cards (that is, end-of-file on SYSIN), then the AREAD assigns a null string value. Processing continues normally, and null assignments can continue with additional AREAD statements in the macro definition. Note that it is always source cards that are read. Thus, even though AREAD can occur in an inner macro, the cards read are the source cards immediately following the outermost macro call. If control passes through a given section of code repeatedly because of AIF and AGO statements, then the same source cards are available to be reread.

The AREAD statement assigns to the variable symbol the 80-character image of the next source card in the source text stream or in code generated by a COPY statement. Repeated AREAD statements read successive cards. Cards so read are printed in the assembly listing (unless operand is NOPRINT) with statement numbers (unless operand is NOSTMT), but are otherwise unprocessed by the assembler, in much the same fashion as the card following a REPRO instruction. For example:

Name	Operation	Operand	Comment
	MACRO		
	READER	&N	macro prototype
.LOOP	ANOP		
&K	SETA	&K+1	
&CARD(&K)	AREAD		read a card record
	AIF	(&K LT &N).LOOP	
	MEND		
	READER	2	macro call
A			
B			
	END		

This example has the effect of assigning to &CARD(1) an "A" followed by 79 blanks, and to &CARD(2) a "B" followed by 79 blanks. Upon exit from the macro, the next card processed by the assembler is the END card.

AREAD I/O Capability: The AREAD facility complements the PUNCH Facility to provide macros with direct I/O capability. The total I/O capability of macros can be described as follows:

Implied Input: Parameter values and global SET symbol values that are passed to the macro.

Implied Output: Generated statements are passed to the assembler, and global values are set by the macro.

Direct Input: AREAD.

Direct Output: MNOTE for printed messages, PUNCH for punched cards.

For example, the AREAD and PUNCH statements can be used to write card conversion programs. The following macro interchanges the left and right halves of the input cards.

```
Name      Operation     Operand
          MACRO
          SWAP
.LOOP     ANOP
&CARD     AREAD
          AIF           ('&CARD'EQ'').MEND
&CARD     SETC          '&CARD'(41,40).'&CARD'(1,40)
          PUNCH         '&CARD'
          AGO           .LOOP
.MEND     MEND
```

AREAD Within COPY Code: If the AREAD statement occurs in a macro called within COPY code, then the cards are read from the COPY file. Procedures at the end of a file are automatic for AREAD on COPY files; AREAD continues reading from the outer COPY file in case of nested COPY, or from the source stream.

FORMAT CHANGES IN MACRO STATEMENTS

Within the operand field of macro prototype and macro call statements, keyword and positional format dependencies are relaxed in Assembler H. All macro prototypes are mixed mode. Mixed mode in Assembler F requires all positional parameters to precede keyword parameters. In Assembler H, however, keyword and positional parameters can be freely intermixed in the operand field, as shown in Figure 6.

```
Name      Operation     Operand                 Comment
          MACRO
&NAME     MASTER        &A,&KEY1=,&B,           intermixed parameter types.          X
                        &KEY2=,&KEY3=,          alternate coding format.             X
                        &C,                     allows open-ended list of            X
                        &KEY4=,                 positional parameters that           X
                        &D,                     can easily be inserted without       X
                        &E                      reworking the entire operand field.  X
          XXX                                   any instruction.
          MEND
```

Figure 6. Mixed Parameters in Macro Prototype

Positional parameters within operands containing keyword and positional parameters can be accessed by a system variable, &SYSLIST(n), where "n" is a subscript corresponding to the number of that particular positional parameter that you wish to access. &SYSLIST evaluates the overall operand field by passing over the field from left to right and ignoring any keyword parameters it encounters.

Example: If the operands of a macro call are: A,KEY=B,C
Then &SYSLIST(2)=C

A maximum of 240 positional and keyword parameters is permitted in a prototype statement. There is no limit to the number of positional parameters you can put in the operand of a macro call.

For example, you could have 20 positional parameters in a macro prototype and have 400 positional parameters in the corresponding macro call. You could then reference any one of the parameters in the macro call by &SYSLIST(n), where "n" is 1 to 400.

Multi-Level Sublists In Macro Calls And Prototypes: Multi-level sublists (sublists within sublists) are permitted in macro-instruction operands and in keyword default values in the prototype statement, as shown in the following sample prototypes:

```
MAC1        (A,B,(W,X,(R,S,T),Y,Z),C,D)
MAC2        &KEY=(1,12,(8,4),64)
```

The depth of this nesting is limited only by the constraint that the total length of an individual operand cannot exceed 255 characters.

N'&SYSLIST with an n-element subscript array gives the number of operands in the indicated n-th level sublist. N' of an operand name with an n-element subscript array gives the number of operands in the indicated (n+1)th level sublist. For example, if &P is the first positional parameter and the value assigned to it 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)
&P92,2,1)   = &SYSLIST(1,2,2,1)    = C
&P(2,2,2)   = &SYSLIST(1,2,2,2)    = null

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
```

OTHER REVISIONS

The following rules apply to further language extensions of Assembler H relative to Assembler F.

- Macro names, variable symbols (including the ampersand), and sequence symbols (including the period) can be a maximum of 63 alphameric characters. The first character, excluding ampersands and periods, must be alphabetic.
- Macro definitions can be copied from a library by issuing the statement: COPY XX (where XX is the macro name). The library definition must include a MEND statement.
- Comments (both '*' and '.*' types) can be inserted between the macro header and the prototype. Any such comments are discarded by the macro-edit phase and are not printed with the generation of the macro.
- Any mnemonic operation code of the IBM System/360 and 370 Standard Instruction Set or any conditional assembly operation code can be used as a macro instruction. When any of the operation codes is redefined as a macro instruction, subsequent use is interpreted as a macro call.
- Any instruction, except ICTL, is permitted within a macro definition.
- An equals sign can be embedded in a positional or keyword parameter in a macro call. The positional operand will be accepted and handled

properly; however, a warning message will appear if the equals sign is preceded by an alphanumeric string that appears to be a keyword. For example:

Name	Operation	Operand	Comment
&NAME	MACRO MAC . . MEND . MAC .	&A=,&B=,&C A=X=Y,D=T	macro prototype macro call

The first expression in the macro call is a keyword parameter with an embedded equals sign (X=Y), and the second expression is a positional parameter also containing an embedded equals sign (D=T). The second expression is evaluated as a positional parameter because there is no corresponding keyword parameter ('&D=') in the prototype statement in the macro definition.

- MNOTE statements are permitted in open code with all of the substitution options.
- An implicit or explicit SET variable declaration must be encountered before the corresponding SET variable is used in the operand field. Otherwise there is no ordering restriction on statements following the prototype statement. LCLx, GBLx, and ACTR instructions can appear anywhere in the definition.
- An ACTR count is halved when an error is detected during macro expansion. This rapidly shortens any loops that may be caused by repeated erroneous statements.

Extensions to Conditional Assembly Instructions

The flexibility of the AIF, AGO, SETA, SETB, and SETC instructions is increased in Assembler H. In Assembler H, multiple AIF statements can be merged in one AIF statement, the AGO statement has an expanded interpretive function, and a single SETx instruction (SETx is either SETA, SETB, or SETC) can assign values to more than one element of a SET symbol array. Format and ordering restrictions are also revised, and new system variable symbols are introduced. In addition, generated statements have new functions, and the availability of symbol attributes is increased.

EXTENDED AGO STATEMENTS

In Assembler H, one AGO instruction can contain computed branch sequence information. The extended AGO statement has the following format:

Operation	Operand
AGO	(k).S1,.S2....,.Sn

Where "k" is a SETA arithmetic expression. If the value of "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. The statement is exactly equivalent to the following sequence of AIF instructions:

Operation	Operand
AIF	(arithmetic expression EQ 1).S1
AIF	(arithmetic expression EQ 2).S2
.	
.	
.	
AIF	(arithmetic expression EQ n).Sn

EXTENDED AIF STATEMENTS

The AIF statement in Assembler H can include a string of logical expressions and related sequence symbols. There is no limit on the number of expressions and symbols that you can use in an extended AIF statement. The format is:

Operation	Operand	Column 72
AIF	(logical expression).S1, (logical expression).S2, ...,(logical expression).Sn	X X

This is equivalent to "n" successive AIF statements. The branch is taken to the first sequence symbol (scanning left to right) that corresponds to a true logical expression. If none of the logical expressions is true, control passes to the next sequential instruction.

EXTENDED SET STATEMENTS

The SETA, SETB, and SETC statements are used in IBM System/360 assemblers to assign arithmetic, binary, and character values, respectively, to SET variable symbols. In Assembler H, the SET statement can be used to assign lists, or arrays, of values to subscripted SET symbols. In Assembler F, each element of a list requires a single SET statement to assign its value to a symbol. For example, a list of 100 SETx values requires 100 SETx coded statements. In Assembler H, such a list can be coded in one extended SETx statement. The extended SETx statement has the following format:

Name	Operation	Operand
&SYM(k)	SETx	X1,X2,,X4,.....,Xn

The form of the name and operation fields is the same as that used in Assembler F for assignment of a dimensioned variable SET symbol: &SYM is a dimensioned SET symbol, "k" is a SETA arithmetic expression, and SETx is SETA, SETB, or SETC.

Each of the operands ("Xn") has the form of an ordinary SETx operand, or it may be omitted. Whenever an operand is omitted, the corresponding element of the dimensioned variable SET symbol (&SYM) is left unchanged.

When none of the operands is omitted, the extended SETx statement is equivalent to the sequence of statements:

```
&SYM(k)          SETx    X1
&SYM(k+1)        SETx    X2
.
.
.
&SYM(k+n-1)      SETx    Xn
```

Following are examples of the use of extended SETx statements:

a.
```
&X(3)        SETA    3,,5,,7
```

This is equivalent to the sequence:

```
&X(3)        SETA    3
&X(5)        SETA    5
&X(7)        SETA    7
```

b.
```
&X(1)        SETA    1,&X(1)+1,&X(2)+1
```

This is equivalent to the sequence:

```
&X(1)        SETA    1
&X(2)        SETA    2
&X(3)        SETA    3
```

c.
```
&Y(1)        SETC    '',,''
```

This sets &Y(1) and &Y(3) to null values and leaves &Y(2) unchanged.

SET SYMBOL FORMAT AND DEFINITION CHANGES

Extensions in Assembler H to SETx statements, and local and global definition statements, are discussed in the following list.

- Dimensioned SET symbol list sizes are effectively unlimited. Dimensioned SET symbols, declared implicitly (see below) or explicitly, are open ended. The declared limit can be exceeded without being flagged as an error. In Assembler F, the sizes are limited to 2500 elements.

- Global (GBLx) and local (LCLx) definitions of SET symbols can occur anywhere in open code or in macros.

- In Assembler H, global and local declarations are processed at generation time in the assembly process, not edit time as in Assembler F. Either a macro definition or open code can contain more than one declaration for a given SET symbol, as long as only one is encountered during a given macro expansion or conditional assembly of open code, as controlled by AIF and AGO statements.

- A SET symbol that has not been declared in a LCLx or GBLx statement is implicitly declared by appearing in the name field of a SETx

statement. The declaration defaults to LCLx, with the type determined by the SETx operator, and the dimensionality is determined by the occurrence of a subscript in the name field. Any explicit declaration encountered thereafter is flagged as a duplicate declaration. Undeclared SET symbols are not valid in Assembler F, and error messages are generated when encountered.

- SETC values, substring values, and character-relation terms can be up to 255 characters long. A SETC value can be the same size as a macro parameter. With its 8-character limit, Assembler F may require a controlled loop to accomplish what is typically done by one statement by Assembler H.

- No fixed limit is placed on local and global symbol dictionary sizes. The dictionaries will grow as long as space is available in the region provided for the assembly.

- Any SETC expression can be preceded by a duplication factor that is a SETA expression enclosed in parentheses. For example:

```
&C      SETC    (2)'XYZ'.(&J*2)'ABC'
```

  If the value of &J is 2, the above statement is equivalent to

```
&C      SETC    'XYZXYZ'.'ABCABCABCABC'
```

- SETA variables can be used anywhere a SETB value is permitted. Its Boolean value is 0 if its arithmetic value is zero and 1 if its arithmetic value is nonzero. For example:

```
&A      SETA    0
&B      SETA    1
&C      SETA    47
```

Given these SETA declarations, SETB statements produce the following results:

```
        &X      SETB    (&A)
.*              &X=0
        &Y      SETB    (&B)
.*              &Y=1
        &Z      SETB    (&C)
.*              &Z=1
```

CREATED SET SYMBOLS

Created SET symbols are produced in open code and macro definitions by substitution at generation time in the assembly process. Created SET symbols cannot be used in Assembler F because the actual SET symbol names are discarded by the macro edit phase, and are not available at generation time. A created SET symbol has the form &(e) where "e" represents a sequence of one or more of the following:

- Variable symbols, optionally subscripted and optionally followed by a period for concatenation.

- Strings of alphameric characters.

After substitution and concatenation, "e" must consist of a string of 1 to 62 alphameric characters, the first being alphabetic. This string is then used as the name of a SETx variable. For example:

```
&Y(1)          SETC        'A1','A2','A3','A4'
&(&Y(3))       SETA        5
```

These statements have an effect similar to: &A3 SETA 5.

Created SET symbols can be used wherever ordinary SET symbols are permitted, including declarations; they can even be nested in other created SET symbols. The following nested variable could generate a valid created SET symbol:

```
&(&(&X(&(&Y))))
```

The created SET symbol can be thought of in one sense as a form of indirect addressing. Thus, in the first example above, &X is a variable whose value is the name of the variable to be updated. With nested created SET symbols, you can get such 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" attribute of the symbol substituted for &SYM.

A related application is illustrated in the following macro definition. This macro is designed to push an item into the specified push-down stack. A new stack is created for each new stack name given as a parameter in the macro call. Note that &LIST becomes as long as required.

```
                               MACRO
                               PUSH        &STAK,&ITEM
                               GBLA        &(&STAK)(1),&(&STAK.SIZE)
&(&STAK.SIZE)                  SETA        &(&STAK.SIZE)+1
&(&STAK)(&(&STAK.SIZE))        SETA        &ITEM
                               MEND
```

The macro call "PUSH LIST,25" is logically equivalent to:

```
                               GBLA        &LIST(1),&LISTSIZE
&LISTSIZE                      SETA        &LISTSIZE+1
&LIST(&LISTSIZE)               SETA        25
```

Created SET symbols also enable you to get some of the effect of multidimensional arrays by creating a separate name item for each element of the array. For example, a three-dimensional array of the form &X(&I,&J,&K) can be addressed as &(X&I.$&J.$&K). Then &X(2,3,4) would be represented as a reference to the symbol &X2$3$4.

Note that what is being created here is a SET symbol. Both creation and recognition occur at macro expansion time. In contrast, parameters are recognized and encoded (fixed) at macro edit time. Consequently, if a created SET symbol name happens to coincide with a parameter name, the fact is ignored, and there is no interaction between the two.

GENERATED COMMENTS

You can generate a comment field by embedding blanks in a generated operand field. Anything following the blank becomes the comment. For example:

Name	Operation	Operand	Comment
	MACRO		
	MAC1	&A	macro prototype
&C	SETC	' '.'&A'(2,K'&A-2)	
	LA	2,10&C	
	MEND		
	.		
	.		
	MAC1	'GENERATED COMMENT'	macro call
+	LA	2,10 GENERATED COMMENT	

In addition, variable symbols in a comment can be evaluated and replaced in certain special cases, for example, when the operation code takes no operand or an optional operand, or when the operand and comments-field format coincides with macro-call alternate format.

AVAILABILITY OF ATTRIBUTE REFERENCES

In Assembler H, attributes of symbols produced by macro expansion or open-code substitution are available immediately after the defining statement is generated. In Assembler F, attributes of symbols in generated statements are never accessible; the attributes of symbols are fixed when a macro is edited. In Assembler H, only syntactic validity is checked during the macro edit.

Forward Attribute References: If an attribute reference is made to an as yet undefined symbol, or if an open-code AIF or AGO instruction effects a forward branch to a previously undefined sequence symbol, the source text is scanned forward until the referenced symbol is encountered in the name field of an open-code statement. This operation is terminated when the symbol is found, or when the end of source text is reached. Attribute entries are made in the symbol table for this symbol if it is found, and for all other previously undefined symbols encountered during the forward scan. Attributes established for a given symbol by the forward scan may be overridden by the first occurrence of the symbol in the name field of a statement that the assembler is actually processing.

Attribute references in Assembler H always involve a direct reference to the symbol table. As a result, attributes are always available for parameter values, regardless of the manner in which the values are passed to the parameter. Assembler F cannot obtain attributes for parameters of inner macro calls unless the parameter value is passed directly from an outer call parameter. Figure 7 demonstrates attribute references to symbols in generated code.

The type (T') attribute of &R is not available in Assembler F because &R is generated within a macro, MAC1. In Assembler H, both type-attribute references are recognized as valid and return valid types. The improved availability of attributes in Assembler H results in many instances in which Assembler H returns true attributes of symbols where Assembler F would return U (undefined) or M (macro).

New Symbol Attribute: The defined (D') attribute is introduced in Assembler H. It can be used on conditional-assembly statements to determine if a given ordinary symbol has been defined at a prior point in the source program. If the symbol has been defined, the value of the D' attribute is one; if it has not been defined, the value is zero. By testing a symbol for the D' attribute, you can avoid a forward scan of the source text.

K' Attribute for SET Symbols: K' can be applied to SETx variables to determine the length of the named item in characters (after conversion to a character-string value in the case of SETA and SETB variables). For example:

```
&A          SETA        0100
```

then K'&A equals 3 (the leading zero is lost in the conversion process). In Assembler F, K' can be used only to refer to parameter names.

```
Name    Operation    Operand      Comment
        MACRO
        MAC1         &P           macro prototype
        .
        .
&A      SETC         'ABC'
        .
        MAC2         &P,&A        inner macro call
        .
        .
        MEND
        .
        MACRO
        MAC2         &Q,&R        macro prototype
        .
        .
&C1     SETC         T'&Q
&C2     SETC         T'&R
        .
        MEND
        MAC1         XYZ          macro call
        .
```

Figure 7. Attribute References to Generated Code

N' Attributes for SET Symbols: N' can be applied to SETx variables to determine the highest subscript value that has been used on the left side of a SETx instruction. For example, if the only occurrences of the SET symbol &A were:

```
Name      Operation    Operand
&A(1)     SETA         0
&A(2)     SETA         0
&A(3)     SETA         &A(2)
&A(10)    SETA         0
```

then N'&A would be 10.

N' is zero if the SET symbol has not been assigned a value. In Assembler F, the N' attribute of a symbol can only refer to parameter names.

Attribute References to SETC Variables: Assembler H permits T', D', L', S', and I' attribute references to SETC variables in open code and in macro definitions. In Assembler F, you can only refer to the attributes of symbolic parameters in macros, and to the attributes of ordinary symbols in open code.

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 character in column 72. Comments are inserted optionally between the blank and column 72. The last line terminates the series with a blank in column 72.

In Assembler F, alternate format can be used only in macro prototype and macro call statements. In Assembler H, the extended AGO, extended AIF, GBLx, LCLx, and extended SETx statements can also be written in alternate format, as shown in the following examples:

```
Name      Operation     Operand                  Comment
          AGO           (&A).S1,                 comment      X
                        .S2,.S3,                              X
                        .S4
          AIF           (&L1).S1,                comment      X
                        (&L2).S2,                             X
                        (&L3).S3
          GBLA          &A1,                                  X
                        &B(5)
          LCLC          L1,                      comment      X
                        L2,L3,                   comment      X
                        L4                       comment
&B(1)     SETB          0,                       comment      X
                        (&A NE 3),               comment      X
                        ('SC' EQ 'XYZ')
```

New System Variable Symbols

System variable symbols are local variable symbols that are assigned values by the assembler when they are encountered. Four new system variable symbols are provided in Assembler H for use in macro definitions.

&SYSLOC: &SYSLOC is identical in function to &SYSECT, except that its value is the character string that represents the location counter (as controlled by the LOCTR statement) that is in effect at the time the macro is called. &SYSECT gets the value of the character string that represents the current CSECT, DSECT, or START section. If no LOCTR statement is in effect, the value of &SYSLOC is the same as the value of &SYSECT. &SYSLOC can be used only in macro definitions. The LOCTR instruction is described in the section of this manual entitled "Changes to Programming Sectioning and Linking Controls".

For example, when the following statements occur in a source program, &SYSLOC will have the character string value XYZ during expansion of MAC1.

Name	Operation	Operand
XYZ	LOCTR	
	MACRO	
	MAC1	
&C	SETC	'&SYSLOC'
	MEND	
	MAC1	

&SYSTIME: The value of &SYSTIME is the time of assembly as it appears in the heading of the assembly listing. The value remains constant throughout the assembly. The value of &SYSTIME is the 5-character string "hh.mm" (hours.minutes). &SYSTIME can be used in macro definitions or open code.

&SYSDATE: The value of &SYSDATE is the date of assembly, exactly as it appears in the heading of the assembly listing. The value of &SYSDATE is the 8-character string "mm/dd/yy" (month/day/year). &SYSDATE can be used in macro definitions or open code.

&SYSPARM: The system variable &SYSPARM has the character-string value specified by the SYSPARM parameter in the PARM field of the EXEC statement in a JCL card. &SYSPARM can be used in macro definitions or open code.

Changes to Basic Assembler Language

This section discusses the changes and extensions to the basic assembler language. These include several new assembler operation codes, extended mnemonics for branch instructions, and language syntax changes. Also, program sectioning and linking controls have been expanded.

New Assembler Operations

New assembler operations extend programming controls and flexibility in several areas of assembler-language programming. A pair of new instructions allows you to protect your USING status and your PRINT status from being lost because of changes made by autonomous macros and subroutines that are used in your program. Another new instruction allows you to change or delete operation code definitions at any point in a source program.

RETENTION OF PRINT AND USING STATUS

Two new assembler instructions, PUSH and POP, are introduced to save and restore the PRINT status or USING status of a segment of an assembly. When entering a macro, a subroutine, or code generated by a COPY instruction, unknown conditions may cause the altering of your PRINT status or USING status before control is returned to your routine. To avoid this, you can save your PRINT or USING status by issuing one of the following instructions:

Operation	Operand
PUSH	USING
PUSH	PRINT
PUSH	PRINT,USING
PUSH	USING,PRINT

The PUSH instruction does not alter the PRINT or USING status; it stores the status. When control is returned to your program, you can restore your USING and PRINT status by issuing a POP instruction. The format of the POP instruction is the same as that of PUSH. A POP instruction restores the condition retained by the last PUSH operation. Sequence symbols can be used in the name field of PUSH and POP statements.

CHANGING OPERATION CODE DEFINITIONS

The new assembler operation OPSYN adds or deletes entries in the operation code table. OPSYN is also supported by Assembler F. In Assembler F, the OPSYN statement must appear in the source program before any machine-operation instructions, macro instructions, or any macro definitions. These restrictions are eliminated in Assembler H. The OPSYN statement has the following format:

Name	Operation	Operand
opname1	OPSYN	[opname2]

The following rules apply to the application of OPSYN:

- If no operand is present, the operation code in the name field is deleted from the operation code table.
- If an operand is present, the operation code in the name field is defined (or redefined) as equivalent to the operation code in the operation field. The newly-defined operation code acquires all attributes of its prototype. For example, if the prototype is a macro instruction, then the new operation code is also a macro instruction, calling on the same macro definition.

Revised Assembler Operations

Several assembler operations used in Assembler F are extended in Assembler H. The following operations are changed:

Operation	General Function
EQU	EQU assigns the length, value, and relocatability attributes of an expression in the operand to the symbol in the name field.
DROP	DROP causes the release of base registers that are established by USING statements.
COPY	COPY obtains source-language coding from a library and inserts the copied code in the source program immediately after the COPY statement is encountered.
PRINT	PRINT controls the printing options that effect the assembly listing.
ORG	ORG alters the location counter for the current control section.
CNOP	CNOP aligns an instruction at a specific halfword boundary.
Literal	Literals introduce data into a program. They are simply constants preceded by an equals sign.

The Assembler H changes to the assembler operations that are listed here are discussed under the headings that follow.

EQU INSTRUCTION EXTENSIONS

- With Assembler F, all symbol attributes are set by the assembler. The function of the EQU statement is extended in Assembler H to allow a wider range of potential attribute references and to permit direct programmer control of attributes. The extended EQU statement allows a

wide range of T' and L' attributes to be assigned to symbols used as EQU statement names.

- A second operand can be used in an EQU statement to define explicitly the length attribute of the symbol in the name field, and a third operand can be used to override the normal type attribute assigned to the EQU statement name. The extended EQU statement permits you to establish your own attributes for your source data. This is a very powerful tool in macro-language implementations.
- Previous definition of symbols used in the first operand of the EQU statement is not required. Symbols used in the second and third operand fields, however, must be previously defined.
- 32-bit positive or negative values are kept for EQU statements, and printed in the assembly and cross-reference listing.
- Complexly relocatable first operands are allowed in EQU statements.
- The format of the EQU instruction statement is as follows:

Name	Operation	Operand
A SET variable or ordinary symbol	EQU	Expression 1 [{,Expression 2[,Expression 3] / ,,Expression 3}]

Expression 1 can be any relocatable or absolute expression. Expression 2 can be an absolute expression with a value range of 1 through 65536. Expression 3 can be an absolute expression with a value range of 0 through 255.

Note: Operands aligned within brackets, [], are optional. Operand stacked within braces,{}, must have one operand chosen.

DROP INSTRUCTION EXTENSION

- A DROP instruction with a blank operand can be issued. This causes the release of all currently active base registers. In Assembler F, all base registers to be released have to be specified in the operand field.

COPY INSTRUCTION EXTENSIONS

- COPY can occur within COPY code. If an internal COPY refers to code that is being copied by an external COPY, the internal COPY is passed over, and a diagnostic message is issued.
- Copied code can contain MACRO and MEND instructions. A COPY containing a MACRO statement must contain a corresponding MEND statement.
- An END statement can occur in copied code, and it can be produced by a copied macro definition when the copied macro is generated. An END model statement in a macro definition is only effective when the

macro is generated; it is not effective when the macro is edited. If an END statement is encountered in copied macro-generated code, or any other copied code, the assembly is terminated. If BATCH mode is in effect, the next assembly begins with the next card in the source-text input stream, and the copy or macro nest is abandoned.

PRINT INSTRUCTION EXTENSIONS

- The PRINT instruction is effective within macro definitions. In Assembler F, PRINT status could be altered only in open code.
- The PRINT statement is always printed, except when PARM=NOLIST, regardless of print options in effect.

CNOP INSTRUCTION EXTENSIONS

- The label field of a CNOP can contain a symbol. If the CNOP instruction does not fall on a halfword boundary, the value of the symbol is the first halfword address greater than the location-counter value before operation of the CNOP. If the CNOP is on a halfword boundary, the value of the symbol is the current location-counter setting. If an "*" is used as a term in the operand field, it will have the same value as the symbol.
- Symbols in the operand field of CNOP do not require previous definition.

ORG INSTRUCTION EXTENSIONS

- A symbol can be used in the name field of an ORG statement. The symbol is assigned the value of the location counter prior to the operation of the ORG. If an "*" is used as a term in the operand field, it will have the same value as the symbol.
- The location-counter values, both before and after the operation of the ORG, appear in the assembly listing.
- An ORG statement cannot specify a location outside or before the beginning of the LOCTR in which it appears. This disallows a negative value; the value is no longer truncated to a high positive value.
- An ORG statement with a blank operand causes the current location counter to be reset to its highest previous setting within the CSECT or LOCTR.

LITERAL INSTRUCTION EXTENSIONS

- Modifier expressions can be used in literals. Any symbol occurring in a duplication factor or length modifier expression must be defined prior to the literal reference. For example, in the following statement, the use of L'SYM requires previous definition of SYM.

```
MVC SYM,=CL(L'SYM)'$'
```

- Q- and S-type address constants can be used in literals. This increases the power of literals serving to make coding easier and to improve diagnostic capability by allowing the programmer to code a constant within the logic sequence in which it is used. With S-type address constants, the address decomposition in terms of base register and displacement is based on the USING statements in effect at the beginning of the associated literal pool.

Assembler Language Syntax Extensions

The syntax of an assembler language deals with the structure of individual elements of an instruction statement and with the order in which the elements are presented in the statement. Several important syntactical elements of the F-level assembler language are extended in the H-level assembler language.

CONTINUATION LINES

A maximum of 9 continuation lines can be used in an ordinary assembly statement in Assembler H. This allows a total of 10 lines per statement - 9 continuation lines (with a character in column 72) plus 1 final line (with a blank in column 72). The Assembler F limit is a total of 3 lines per ordinary assembly statement. In Assembler F and Assembler H, there is no limit on continuation lines in a macro or conditional-assembly statement.

SYMBOL LENGTH

In Assembler H, a maximum of 63 characters can be used for a valid symbol. This limit includes the ampersand for variable symbols and the period for sequence symbols. The corresponding limit in Assembler F is 8 characters.

External assembly symbols are restricted to 8 characters as in Assembler F. Restricted symbols are those used in the name fields of START, CSECT, COM, and DXD statements, and in the operand fields of EXTRN, WXTRN, and ENTRY statements. Symbols used in V-type and Q-type address constants are also restricted to 8 characters.

TREATMENT OF SIGNED VALUES

Unlike Assembler F, Assembler H can use unary + and - operators in expressions. This allows greater freedom in formulating arithmetic expressions. A unary operator can precede a:

- Term
- Left Parenthesis
- Unary Operator

A unary operator can begin an expression or can follow a:

- Left Parenthesis

- Unary Operator
- Binary Operator

The following expressions are valid in Assembler H, but not in Assembler F:

- -1
- 1+-10
- A-+B
- A--B
- A*-B

Unary operations are performed before binary operations, and an expression cannot contain two terms or two binary operators in succession. Two successive minus signs are equivalent to a plus sign.

SYMBOL VALUES

Self-defining terms are restricted to three-byte values in Assembler F. In Assembler H, four-byte values can be used for self-defining terms. The value of a symbol can lie in the range -2^{31} through $2^{31}-1$. For example, the following equate statements assign fullword negative values to the symbols in the name fields:

```
SYMBOL          EQU         X'FFFFFFFF'
SYM2            EQU         C'ABCD'
SYM3            EQU         -1
```

Note: In Assembler H, a self-defining term with a 1 in the sign bit is treated as a negative number in expression evaluation. Thus, the values of SYMBOL and SYM3 in the example are equivalent.

LEVELS WITHIN EXPRESSIONS

Extending the freedom in expression evaluation, any number of terms or levels of parentheses can be used in an expression. The artificial limits established in Assembler F restrict expressions to 16 terms, or 5 levels of parentheses.

CHARACTER VARIABLES USED IN ARITHMETIC EXPRESSIONS

A SETC variable can be treated as an arithmetic term if its character-string value represents any valid self-defining term. A null value is treated as zero. This facility is available in Assembler F, but the characters of the self-defining term are restricted to 1 through 9.

The expanded facility in Assembler H allows you to associate numeric values with EBCDIC or hexadecimal characters which can be used in such applications as indexing, code conversion, translation, or sorting.

For example, the following set of instructions converts a hexadecimal value in X'F3' into a decimal value in &VAL:

```
&X      SETC        'X''F3'''
&VAL    SETA        &X
```

Mnemonic Operation Code Extensions

Assembler H provides extended mnemonic operation codes for conditional branch instructions in the RR (register-to-register) format, that is, BCR (branch-on-condition-register) instructions. This instruction set complements the existing set of conditional branches used in Assembler F for RX (register-to-storage) operations.

The following extended mnemonic operation codes for the BCR instruction are effective in Assembler H:

Extended Code		Meaning	Machine Instruction
	Used After Compare Instructions		
BHR	R2	Branch on High	BCR 2,R2
BLR	R2	Branch on Low	BCR 4,R2
BER	R2	Branch on Equal	BCR 8,R2
BNHR	R2	Branch on Not High	BCR 13,R2
BNLR	R2	Branch on Not Low	BCR 11,R2
BNER	R2	Branch on Not Equal	BCR 7,R2
	Used After Arithmetic Instructions		
BOR	R2	Branch on Overflow	BCR 1,R2
BPR	R2	Branch on Plus	BCR 2,R2
BMR	R2	Branch on Minus	BCR 4,R2
BZR	R2	Branch on Zero	BCR 8,R2
BNPR	R2	Branch on Not Plus	BCR 13,R2
BNMR	R2	Branch on Not Minus	BCR 11,R2
BNZR	R2	Branch on Not Zero	BCR 7,R2
	Used After Test Under Mask Instructions		
BOR	R2	Branch if Ones	BCR 1,R2
BMR	R2	Branch if Mixed	BCR 4,R2
BZR	R2	Branch if Zeros	BCR 8,R2
BNOR	R2	Branch if Not Ones	BCR 14,R2

Changes to Program Sectioning and Linking Controls

Operations controlling program sectioning and linking are extended in Assembler H to allow increased freedom of program organization. A new instruction is introduced, and several instructions in the F-level assembler language are revised.

USE OF MULTIPLE LOCATION COUNTERS

The assembler instruction LOCTR allows multiple location counters to be defined within a control section during the assembly. The format of this new instruction is:

Name	Operation	Operand
Any ordinary or variable symbol	LOCTR	Blank

The assembler assigns consecutive addresses to all segments of a location counter in a control section before it continues address assignment with the first segment of the next location counter. By using the LOCTR instruction, you can cause your program object-code structure to differ from a logical order appearing in the listing. You can code sections of a program as independent logical and sequential units. For example, you can code work areas and constants within the section of code that requires them, without branching around them. Figure 8 illustrates this procedure.

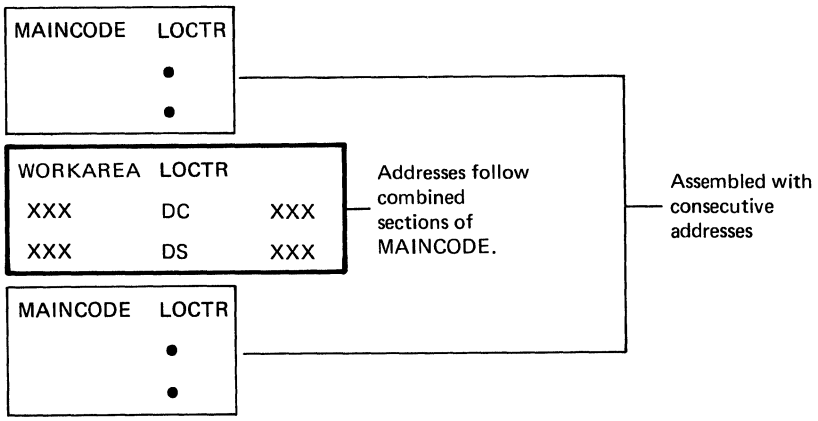


Figure 8. LOCTR Instruction Application

The following rules govern applications of the LOCTR instruction:

- A location counter can be interrupted by a CSECT, DSECT, COM, or another LOCTR instruction.
- A control-section name that is defined by the CSECT, COM, DSECT, or START instruction automatically names the first location counter in that section.
- A LOCTR instruction with the same name as a control section resumes the first location counter in that section.
- A LOCTR instruction with the same name as a previous LOCTR instruction forces a return to the control section in which it was first defined and resumes the particular counter involved.
- Resumption of a control section causes resumption of the last active, not necessarily the highest valued, location counter under that control section.
- A control section name defined for the first time is in error if it is identical to a previously defined LOCTR instruction name.
- A LOCTR instruction occurring before the first control section will initiate an unnamed CSECT before the LOCTR instruction is processed.
- LOCTR instructions do not force location counter alignment.

LINKING COMMON STORAGE AREAS

In Assembler F, the COM assembler instruction cannot be labeled, and it reserves one common area of storage that can be referenced by independent assemblies. In Assembler H, COM statements can be labeled by inserting a symbol reference in the name field. This allows multiple labeled common sections, instead of the one continuous unnamed section of Assembler F, and enhances the programmer's ability to link independent assemblies. The revised format is:

Name	Operation	Operand
Any symbol or blank	COM	Blank

When a problem cannot be handled by a high-level language, or can be more efficiently handled by assembler language, an assembler language subroutine can be used. Variables can be passed between the subroutine program and the higher-level language problem program by storing them in a common area defined by the COM instruction.

In Assembler H, the following rules apply to the use of the common sections:

- A storage area is reserved for each named common section. An unnamed common section is treated the same as in Assembler F. Within one assembly, when the same COM instruction name is used on one or more COM instructions, COM instructions subsequent to the first result in resumption and expansion of the initial common section. The resulting common section is a sum of these sections.
- Within multiple independent assemblies, duplication of a COM instruction name results in a reserved common storage area equal to the largest common section by that name.
- A common section can be interrupted by a CSECT instruction, a DSECT instruction, a COM instruction with a different name, or a LOCTR instruction that is defined prior to initiation of the common section in control at that point in the assembly.

REVISION OF Q-TYPE ADDRESS CONSTANTS

Q-type address constants reserve storage for the offset of an external dummy section. In Assembler H, they are relieved of several F-level language restrictions:

- DXD or DSECT names referenced in Q-type address constants no longer require previous definition.
- Q-type address constants can be generated in literals.
- DXD instructions define external dummy sections. If the relocatable symbol in a DXD statement is not used in a Q-type address constant, the DXD symbol is not placed in the external symbol dictionary (ESD). DXD statements without a Q-type address constant are not addressable by the program.

OTHER REVISIONS

Two additional extensions that relate to program sectioning considerations are developed in Assembler H. Their discussions follow.

ESD and ENTRY Symbols: The number of ESD and ENTRY symbols is not restricted by Assembler H. The maximum number of entries is dependent on the amount of main storage space available to the linkage editor.

Dummy Control Sections (DSECTs): Unnamed DSECTs are valid in Assembler H. The general format of the DSECT statement is:

Name	Operation	Operand
Any symbol, or blank	DSECT	Not used; can contain a comment

In Assembler F, an unnamed DSECT generates a diagnostic message. The Assembler H option frees the programmer of the necessity of inventing a DSECT name when it is not needed. If the name field contains a sequence symbol, the DSECT is unnamed.

Performance Improvements

The assembly of source programs by Assembler H is 1.5 to 10 times faster than assembly of the same programs by Assembler F, depending on the source program, and the system configuration and environment.

Assembly time is usually measured in terms improvised for accounting purposes, and these terms can vary with every installation. For conceptual purposes, assembly time can be defined in the following terms:

- Elapsed Time: The time involved in the processing of a program from source-deck input to object-deck output.
- CPU Time: The time required to do a job, exclusive of I/O requirements and wait time.
- Weighted Time: The CPU time required to do a job plus a fixed amount (for example, 25 milliseconds) for each start I/O operation and each wait that is incurred pending completion of an I/O operation.

Elapsed Time Measurement

This measure of time required to assemble a given program assumes no interference from any other program. Elapsed time is most significant in an operating system environment where the entire system is dedicated to each job step. The following ratios are a conservative measure of elapsed time under Assembler H relative to elapsed time under Assembler F. They do not necessarily apply to programs of fewer than 400 statements with no macros.

CPU	Model 40	Model 50	Model 65	Model 75	Model 85 and up
H step-time / F step-time	1:1.5	1:1.9	1:3.5	1:4.0	1:4.5

CPU Time Measurement

This is a measure of net CPU time, disregarding time spent for starting or completion of I/O. Such a measure may be used in scientific centers where a typical job may involve reading in a few cards of data, performing computation for several minutes or hours with no I/O, and eventually printing out only 5 lines of output. In such a situation where I/O usage is discounted, performance for assemblies under Assembler H is not markedly improved over similar assemblies under Assembler F. However, internal organization of the Assembler H processor reduces the net CPU assembly time by approximately 30 percent over Assembler F.

Using a test case involving a moderate use of macros, the following figures are a measure of the CPU time. The ratio remains constant for all CPUs that support Assembler H.

Assembler	F	H
IBM System/360 Model 40	61 seconds	44 seconds

Weighted Time Measurement

Most installations have a mixture of jobs, some involving light I/O usage and others involving heavy I/O usage. Under MFT or MVT, this I/O is essentially free; that is, another application is being charged for CPU time while your I/O is being performed. Therefore, many accounting routines keep count of start I/O and wait operations and charge a fixed amount of time for each operation. This approach recognizes that I/O usage, just as CPU time, is a resource. Weighted time may also include an extra charge, usually a percentage, for region sizes over and above a standard region size established by an installation. In a weighted-time environment, the ratios shown under "Elapsed Time Measurement" are improved.

Factors Influencing Improved Performance

The following list summarizes the factors that influence the improved timing performance of Assembler H in comparison to Assembler F:

- Logical text stream and tables that are a result of the internal assembly process remain resident in main storage, whenever possible, throughout the assembly.
- Two or more assemblies can be performed under the control of one set of job control language (JCL) cards.
- Assembler H edits only the macro definitions that it encounters during a given macro expansion or during conditional assembly of open code, as controlled by AIF and AGO statements.
- Source-text assembly passes are consolidated. The edit and expansion of macro text is done on a demand basis in one pass of the source text, instead of two distinct passes as in Assembler F, as shown in Figure 1.

Resident Tables and Source Text: Performance over lower-level assembly-language processors is improved by keeping intermediate text, macro-definition text, dictionaries, and symbol tables in main storage whenever possible. This reduces the I/O time required by assemblers that rely heavily on secondary storage throughout the assembly process. Less I/O reduces system overhead and frees channels and I/O devices for other uses.

Certain portions must remain in main storage throughout the assembly process. The symbol table must remain resident, and it has no overflow capacity. Also, all partially filled blocks of text must remain resident.

Multiple Assembly: Multiple or batch assemblies can be done under the control of a single set of JCL cards. Source decks are placed together with no intervening "/*" card. The EXEC card must declare the following:

```
//STEPONE       EXEC        ASMHC,PARM=BATCH
```

Batch assembly improves performance by eliminating job and step overhead for each assembly. It is especially advantageous for processing related assemblies such as a main program and its subroutines.

Macro-Editing Process: New methods of macro processing improve performance. Assembler F edits all source program macro definitions and library macro definitions that are contained or referenced in the source program. This often results in editing macro definitions that are never called. Assembler H edits only those macro definitions that are encountered during a given macro expansion or during conditional assembly of open code, as controlled by AIF and AGO statements.

A good example of potential savings by this feature is the process of system generation. During system generation, Assembler H edits only the set of library macro definitions that are expanded, whereas Assembler F edits all library macro definitions referenced in the system-generation source stream and the library macro definitions for all inner macro calls (to any level). As a result, Assembler H may edit 25 percent fewer library macro definitions than Assembler F.

Consolidating Source Text Passes: In comparison to Assembler F, consolidating assembly source text passes and other new organization procedures reduces by approximately 30 percent the number of internal processor instructions used to handle source text in Assembler H. This is represented in a proportionate saving in CPU time. The saving is independent of the size of speed of the system CPU involved; it is a measure of the relative efficiency of the processor.

Extensions to Listing Controls and Diagnostics

Many diagnostic features are added to Assembler H to aid the location and analysis of program errors. Refinement of macro and conditional-assembly diagnostics is particularly significant. Also, new options governing program listings and object decks are introduced.

Diagnostic Changes in Regular Assembly

New assembler options are available in Assembler H to improve control of source-program diagnostics. Certain diagnostics can be optionally called out or suppressed. Also, new diagnostic information is automatically printed out in the listing, and error messages are generally improved over those of Assembler F.

Literal Cross-Reference Table: Literals are cross-referenced, enabling you to locate the address of references to literal symbols within a program. A literal reference that is repeated throughout a program can thereby be easily identified, located, and changed when necessary. If you wish to change a literal reference that is made repeatedly in a program, this eliminates an exhaustive and error-prone visual scan of the source program.

Suppressing Cross Reference Entries for Non-referenced Symbols: By specifying the option XREF(SHORT), it is possible to suppress the listing in the cross reference table of symbols that are defined but not referenced in the program.

Dictionary Listing Options: The Relocation Dictionary (RLD) and External Symbol Dictionary (ESD) printouts can be suppressed by means of PARM options in the EXEC statement: PARM='NOESD', PARM='NORLD', or PARM='NOESD,NORLD'.

Option to Check Reentrant Status of Program Modules: A reentrant module can be executed by more than one task at a time. It is not modified by itself or by any other module during execution. You can confirm the reentrant status of your modules by declaring in the EXEC statement: PARM='RENT'. This causes every detected occurrence of non-reentrant coding to be flagged in line in your program listing. This option is available in Assembler F; however, the diagnostic message in Assembler F comes at the end of the program listing and cannot be traced to a particular statement. Without this option, extensive testing is the only method of testing a module's reentrant status.

Error Message Suppression: Messages below a specified severity level can be optionally suppressed by declaring in the EXEC statement: PARM='FLAG(n)' (where "n" is the selected severity level). If you are not concerned with warning and error messages in a specific assembly, utilizing this option provides a cleaner listing.

Identification of Object Decks: The name field of a TITLE statement can be blank or can contain 1 to 8 nonblank characters. The contents of this name field are punched into all the output cards that make up an object deck, except for those cards produced by PUNCH or REPRO statements. The name is punched into the cards starting in column 73, and any remaining columns are used for deck sequencing. In Assembler F, the name field can contain 1 to 4 nonblank characters that are punched in columns 73-76.

Printout of Counter Settings (ORG): The ORG instruction sets a location counter value. In the H Assembler, the setting of the counter prior to an ORG instruction and the counter setting generated by the instruction are printed in the listing. Thus, you can analyze the program object code and have a direct reference to the location of program segments that are intermixed as a result of ORG counter settings.

When a section is resumed by a CSECT, DSECT, COM, or LOCTR, the resumed location counter is also printed in the object code.

Error Messages: Assembler H prints error messages in line in the listing and includes at the end of the listing a total of the errors and a table of their line numbers. Certain in line messages include a copy of that segment of the statement that is in error. Thus, error conditions are spelled out as they occur with direct reference to a specific error. Figure 9 illustrates this.

```
                                CSECT
                                •
                                COMM
*** ERROR *** UNDEFINED OP CODE - COMM
                                •
                                DS      (*+5)F
*** ERROR *** RELOCATABILITY ERROR -- (*+5)F
                          1NAME DC      F 'O'
*** ERROR *** SYMBOL TOO LONG ,OR 1ST CHARACTER NOT A LETTER -- 1NAME
                                •
                          &C    SETC    'AGO'
                                &C      .X
*** ERROR *** OP CODE NOT ALLOWED TO BE GENERATED -- AGO
                                •
                                END
```

Figure 9. Sample Diagnostic Messages

Diagnostic Messages in Macro Assembly

In Assembler H, diagnostic messages printed in macro-generated text are much more descriptive than those in Assembler F. In addition, the macro level and the statement number of the macro definition are printed for each programmer macro instruction. The macro level and the first five characters (or fewer) of the macro name are printed for library macro expansions.

Sequence Field in Macro-Generated Text: When a library macro definition is processed as a result of a macro call, the sequence field (columns 73 through 80) of the generated statements consists of the level of the macro call in the first two columns, and the first five letters of the macro definition name in the remaining five columns. When a line is

generated from a source program macro or a copied library macro, the last five columns contain the line number of the model statement in the definition from which the generated statement is derived. This information can be an important diagnostic aid when analyzing output dealing with macro calls within macro calls.

Format of Macro Generated Text: Whenever possible, a generated statement is printed in the same format as the corresponding macro definition (model) statement. The starting columns of the operation, operand, and comments fields are preserved unless they are displaced by field substitution, as shown in the following example:

```
Source Statements:      &C  SETC              'ABCDEFGHIJK'
                        &C  LA                1,4
Generated Statement:    ABCDEFGHIJK     LA    1,4
```

Error Messages for a Library Macro Definition: Format errors within a particular library macro definition are listed directly following the first call of that macro. Subsequent calls on the library macro do not result in this type of diagnostic. If the appropriate PRINT option is in effect, errors arising in the generated text of a library macro are listed in line within the generated text. Figure 10 shows the placement of error messages.

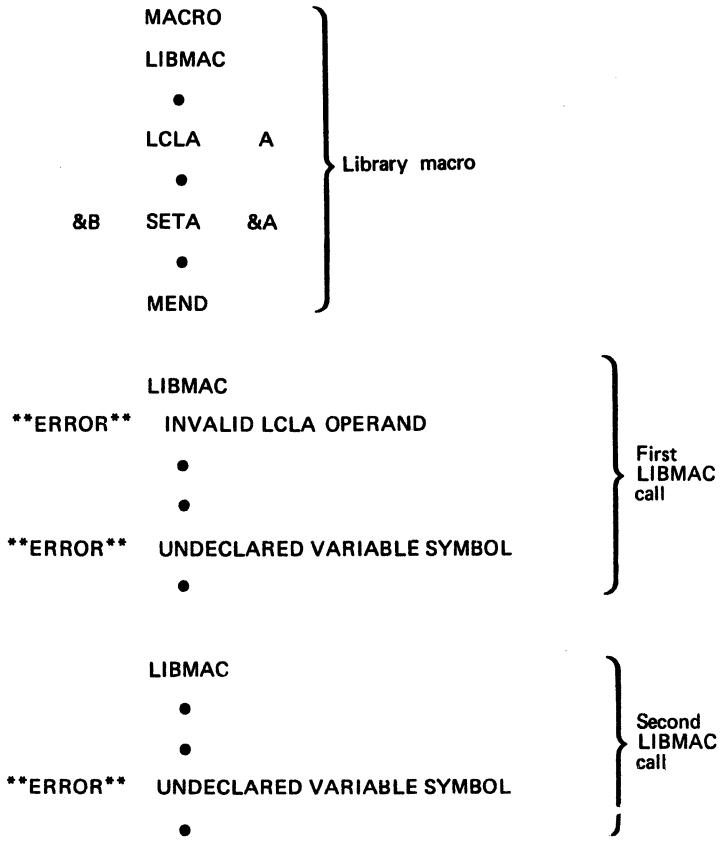


Figure 10. Library Macro Definition Diagnostics

Error Messages for Source Program Macro Definitions: Macro definitions contained in the source program are printed in the listing, provided that the appropriate PRINT options are in effect. Edit diagnostics are inserted in line in the listing directly following the statement in error. Errors analyzed during macro generation produce messages in line in the generated text.

Error Messages in Macro-Generated Text: Diagnostic messages in generated text generally include:

- A description of the error.
- The recovery action.
- The model statement number at which the error occurred.
- A SET symbol name, parameter number, or value string associated with the error.

Macro Trace Facility - MHELP

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 continuously until superceded by another MHELP statement. MHELP options are:

Macro Call Trace: (MHELP B'1' or MHELP 1). This option provides a one-line trace for each macro call, giving the name of the called macro, its nested depth, and its &SYSNDX (total number of macro calls) value. Note: This trace is provided upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

Macro Branch Trace: (MHELP B'10', or MHELP 2). This option provides a one-line trace for each AGO and true AIF conditional-assembly statement within a macro. It gives the model-statement numbers of the "branched from" and "branched to" statements, and the name of the macro in which the branch occurs. This trace option is suppressed for library macros.

Macro Entry Dump: (MHELP B'10000', or MHELP 16). This option dumps parameter values from the macro dictionary immediately after a macro call is processed.

Macro Exit Dump: (MHELP B'1000', or MHELP 8). This option dumps SET symbol values from the macro dictionary upon encountering a MEND or MEXIT statement.

Macro AIF Dump: (MHELP B'100', or MHELP 4). This option dumps SET symbol values from the macro dictionary immediately before each AIF statement that is encountered.

Global Suppression: (MHELP B'100000', or MHELP 32). This option suppresses global SET symbols in the two preceding options, MHELP 4 and MHELP 8.

MHELP Suppression: (MHELP B'10000000', or MHELP 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 256-65536 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 diagnostic message "ACTR EXCEEDED -- &SYSNDX" is issued.

Combining Options: 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.

Abnormal Assembly Termination Processing: The assembler provides a specially formatted dump whenever an assembly cannot be completed.

When the abnormal termination is caused by an unprocessable assembly, the diagnostic dump can aid you in debugging or correcting the problem. In other cases, such as program checks in the assembler itself, the dump provides information useful for assembler maintenance.

Index

&

A

B

C

M

N

O

P

Q

R

S

T

U

V

W

X

OS Assembler H Gen. Info. Man. (S360-21 (OS)) Printed in U.S.A. GC26-3758-3

GC26-3758-3

OS Assembler H
Gen. Info. Man.

GC26-3758-3

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such request, please contact your IBM representative or the IBM Branch Office serving your locality.*

CUT ALONG DOTTED LINE

Reply requested:
Yes ☐
No ☐

Name: _____
Job Title: _____
Address: _____
_____ Zip _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

GC26-3758-3

Your comments, please . . .

Your answers to the questions on the back of this form, together with your comments, will help us to produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Note: Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Fold Fold

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 813 L
1133 Westchester Avenue
White Plains, New York 10604

Fold Fold

IBM®

CUT OR FOLD ALONG LINE

OS Assembler H Gen. Info. Man. (S360-21 (OS)) Printed in U.S.A. GC26-3758-3

OS Assembler H
Gen. Info. Man.

GC26-3758-3

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such request, please contact your IBM representative or the IBM Branch Office serving your locality.*

CUT ALONG DOTTED LINE

Reply requested:
Yes ☐
No ☐

Name: _____
Job Title: _____
Address: _____
_____ Zip _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

GC26-3758-3

Your comments, please . . .

Your answers to the questions on the back of this form, together with your comments, will help us to produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Note: Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Fold Fold

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 813 L
1133 Westchester Avenue
White Plains, New York 10604

Fold Fold

IBM®

CUT OR FOLD ALONG LINE

OS Assembler H Gen. Info. Man. (S360-21 (OS)) Printed in U.S.A. GC26-3758-3

GC26-3758-3

OS Assembler H Gen. Info. Man. (S360-21 (OS)) Printed in U.S.A. GC26-3758-3