GC26-4035-0
File No. S370-21

Program Product

Assembler H Version 2: General Information

Program Number 5668-962

Release 1.0

IBM

GC26-4035-0
File No. S370-21

Program Product

Assembler H Version 2: General Information

Program Number 5668-962

Release 1.0

IBM

This publication was produced using the
IBM Document Composition Facility
(program number 5748-XX9) and
the master was printed on the IBM 3800 Printing Subsystem.

First Edition (December 1981)

This edition applies to Version 2, Release 1.0, of Assembler H, Program Product 5668-962, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest _IBM System/370 and 4300 Processors Bibliography_, GC20-0001, for the editions that are applicable and current.

PREFACE

This publication describes in general terms the Assembler H Version 2, Release 1.0, Program Product 5668-962 (hereafter referred to as Assembler H or, simply, assembler). It contains information about Assembler H extensions to VS Assembler, as well as differences between Assembler H Versions 1 and 2, to help prospective users evaluate and plan for the use of Assembler H Version 2.

ORGANIZATION OF MANUAL

This manual contains the following chapters:

"Chapter 1. Introduction" describes the Assembler H language and tells how to modify it when you add it to your system.

"Chapter 2. Macro and Conditional Assembly Language Extensions" tells about significant enhancements in the macro and conditional assembly language used by Assembler H.

"Chapter 3. Basic Assembler Language Extensions" tells about enhancements to the basic assembler language used by Assembler H.

"Chapter 4. Diagnostic Extensions" tells about the many diagnostic features that Assembler H provides to aid in the location and analysis of program errors.

"Chapter 5. Factors Influencing Improved Performance" discusses the relationship of input/output operations and machine instructions on performance.

ASSEMBLER H READING LIST

The following documentation will be available to users of the program when the product is available:

- _Assembler H Version 2 Application Programming: Guide_
- _Assembler H Version 2 Application Programming: Language Reference_
- _Assembler H Version 2 Installation_
- _Assembler H Version 2 Logic_

SUMMARY OF AMENDMENTS

DECEMBER 1981

VERSION 2, RELEASE 1.0

New features provided by the Assembler H Version 2 Program Product are:

- A program using System/370 Extended Architecture (S/370-XA) machine instructions may be assembled with Assembler H under MVS/Extended Architecture (MVS/XA), OS/VS2 MVS Release 3.8, OS/VS1 Release 7, MVS/SP V1, VM/XA Migration Aid, or VM/System Product (VM/SP). However, a program using Extended Architecture instructions can only be executed on an Extended Architecture mode processor under MVS/XA or MVS/XA guest operating system under VM/XA Migration Aid.
- An AMODE attribute allows specification of the entry point of the addressing mode (24-bit, 31-bit, or either 24- or 31-bit addresses) to be associated with a control section.
- An RMODE attribute allows specification of the residence mode (in the 24-bit addressable range or anywhere) to be associated with a control section.
- New channel command word instructions: CCW1 (format 1) allows 31-bit data addresses; CCW0 (format 0) allows 24-bit data addresses.
- New machine instructions for the Extended Architecture mode processor; in addition, the System/370 (S/370) set of machine instructions has been expanded. A changed installation option allows users to specify whether the S/370, Extended Architecture, or Universal (all inclusive) instruction set will be used for assemblies.
- Three new instruction types are included for the Extended Architecture object code: E, RRE, and SSE.
- An underscore character is allowed in ordinary symbols.
- Operation in the CMS environment of VM/SP and VM/XA Migration Aid.

CONTENTS

FIGURES

CHAPTER 1. INTRODUCTION

Assembler H Version 2 is an IBM program product that runs under MVS/Extended Architecture (MVS/XA), OS/VS2 MVS Release 3.8, OS/VS1 Release 7, MVS/SP V1, VM/XA Migration Aid, and VM/SP. Version 1 of Assembler H made major extensions to the basic assembler language processor, and Version 2 has added new improvements that are listed in the following section.

Assembler H Version 2 is required for installation of MVS/System Product Version 2 (MVS/SP V2) and Data Facility Product Release 1 (5665-284), and for subsequent maintenance of the MVS/XA operating system. It is also required for installation of service and modifications to VM/XA Migration Aid.

IMPROVEMENTS OVER OS ASSEMBLER H VERSION 1

Assembler H Version 2 (5668-962) enables a program using System/370 Extended Architecture (S/370-XA) machine instructions to be assembled with Assembler H under any of the operating systems mentioned above. However, a program using Extended Architecture instructions can only be executed on an Extended Architecture mode processor under MVS/XA or MVS/XA guest operating system under VM/XA Migration Aid.

Version 2 is a functional replacement for and incorporates all the functions available in OS Assembler H Version 1, Release 5.0 (5734-AS1).

New features in Version 2 are:

- An AMODE attribute allows specification of the entry point of the addressing mode (24-bit, 31-bit, or either 24- or 31-bit addresses) to be associated with a control section.
- An RMODE attribute specification of the residence mode (in the 24-bit addressable range or anywhere) to be associated with a control section.
- New channel command word instructions: CCW1 (format 1) allows 31-bit data addresses; CCW0 (format 0) allows 24-bit data addresses.
- New machine instructions for the Extended Architecture mode processor; in addition, the IBM System/370 (S/370) set of machine instructions has been expanded. You may now specify at installation time whether the Extended Architecture, S/370, or Universal (all inclusive) instruction set will be used for assemblies.
- Three new instruction types are included for the Extended Architecture object code: E, RRE, and SSE.
- An underscore character is allowed in ordinary symbols.
- Operation in the CMS environment of VM/System Product (VM/SP) and VM/Extended Architecture (VM/XA) Migration Aid.

LANGUAGE COMPATIBILITY

The language used by Assembler H Version 2, Release 1.0, has functional extensions to the language supported by VS Assembler and OS Assembler H Version 1, Release 5.0. Programs written for VS Assembler and OS Assembler Version 1, Release 5.0, that were successfully assembled with no warning or diagnostic messages, will be assembled correctly by Assembler H Version 2, Release 1.0.

PERFORMANCE

The high-speed assembly capability of Assembler H relative to VS Assembler 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).
- Conditional assembly instructions can be used to bypass macro definitions included in the source text.

SYSTEM REQUIREMENTS

Operating Environments: Assembler H operates under MVS/XA, OS/VS MVS 3.8, OS/VS1 Release 7, MVS/SP V1, VM/XA Migration Aid, and VM/SP.

Virtual Storage: Assembler H Version 2, Release 1.0, requires a minimum of 200K bytes of main storage.

Machine Requirements: Version 2 is designed to operate on the IBM System/370, 303x, 3081, and 43xx processors supported by the above operating systems.

Note: One 2400 or 3400 series tape unit is required for installation. The 2400 series tape unit is not, however, supported by MVS/XA.

Auxiliary Storage Space: Auxiliary storage space is required for the following data sets:

- System input
- Macro instruction library—either system or private or both
- An intermediate work file, which must be a direct access device (3330/3333, 3340/3344, 3350, 3375, or 3380). Under VM/XA Migration Aid, the intermediate work file must be formatted as a CMS minidisk. Under VM/SP, the intermediate work file, which must be a direct access device (3310, 3370, or one of the devices mentioned above), must also be formatted as a CMS minidisk.
- Print output

Library Space: In terms of the IBM 3350 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 15 tracks on SYS1.LINKLIB or a private link library.

Installation: Version 2 is installed using System Modification Program, Release 4 (SMP4) for installations using OS/VS2 MVS Release 3.8, MVS/XA, MVS/SP V1, or OS/VS1, Release 7. Installation on CMS with VM/SP or VM/XA Migration Aid is accomplished by loading the program from tape with VMFPLC2 (a standard VM/SP utility).

INTERNAL DESIGN

The internal organization of Assembler H provides for only two source text passes, in contrast with more for other assemblers. The first pass of the source text by Assembler H edits and expands macros, and builds dictionaries and the symbol table. The second pass completes the assembly and produces the desired output.

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, called "lookahead" mode, of the source text. 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. This avoids further forward scans 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

If the region allocated to the assembler is large enough to contain the text, Assembler H processes all intermediate assembly text in virtual storage. 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 work file (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 work file, but the blocks of text are also retained in working storage and continue to be effective for all assembly purposes. Only when all unallocated work space 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 are accessed from the work file.

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

<u>MODIFYING ASSEMBLER H WHEN ADDING IT TO YOUR SYSTEM</u>

You can make several optional modifications to tailor Assembler H 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 the modifications you can make to Assembler H.

DEFAULTS FOR ASSEMBLER H 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
TERM	NOTERM
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 string of 1-255 characters)

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

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	
SYSTERM	(if TERM is specified)

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 Assembler H can be specified when the assembler is added to your system. Three instruction sets are available: System/370 (S370), Extended Architecture (EXT), and Universal (UNIV), which incorporates both of preceding sets.

The Extended Architecture object code includes three new instruction types called E, RRE, and SSE. Machine instructions of the new types are invoked by means of mnemonics and specified like other machine instructions. Most of the new instructions for the Extended Architecture have the same instruction types as the S/370 instructions. However, several have one of the new types.

CHAPTER 2. MACRO AND CONDITIONAL ASSEMBLY LANGUAGE EXTENSIONS

Many restrictions imposed by VS Assembler are relaxed or eliminated by Assembler H 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.

MACRO LANGUAGE EXTENSIONS

Additional functions of the macro definition and macro instruction provided 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.

Note: The only restriction Assembler H imposes on the placement of macro definitions is that the macro definition must be encountered before it is called.

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.

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.

Assembler H allows you to 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                    The symbol MAC1 is assigned as the
        .                       name of this macro definition.
        MEND
        .
MAC2    OPSYN        MAC1       MAC2 is assigned as an alias for MAC1
        MACRO
        MAC1                    MAC1 is assigned as the name of this
        .                       macro definition.
        MEND
        .
MAC1    OPSYN        MAC2       MAC1 is assigned to the first defini-
                                tion. 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

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. Figure 1 on page 7 illustrates the editing of inner macro definitions.

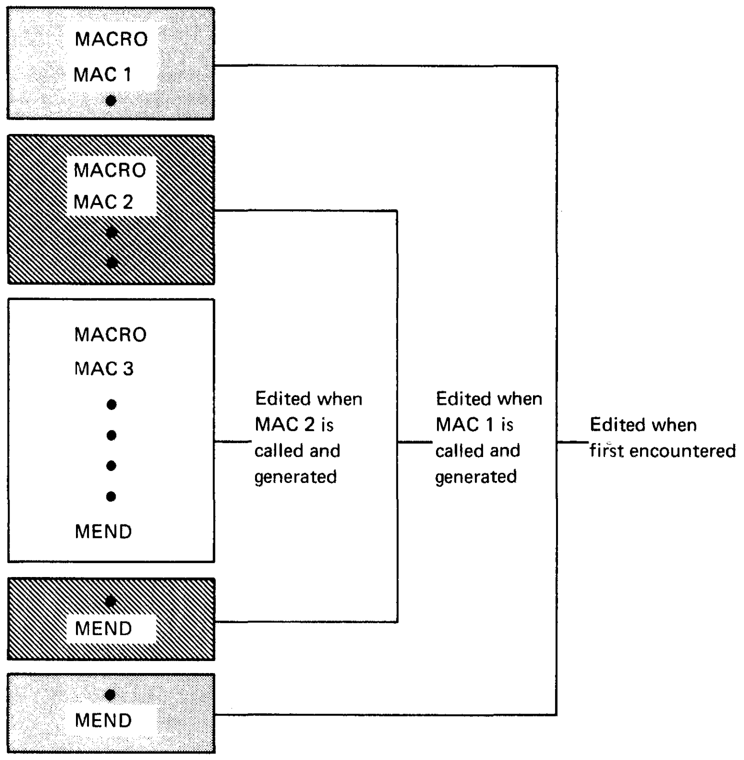

MACRO
MAC 1
MACRO
MAC 2
MACRO
MAC 3
MEND
MEND
MEND
Edited when MAC 2 is called and generated
Edited when MAC 1 is called and generated
Edited when first encountered

Figure 1. Editing 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); 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

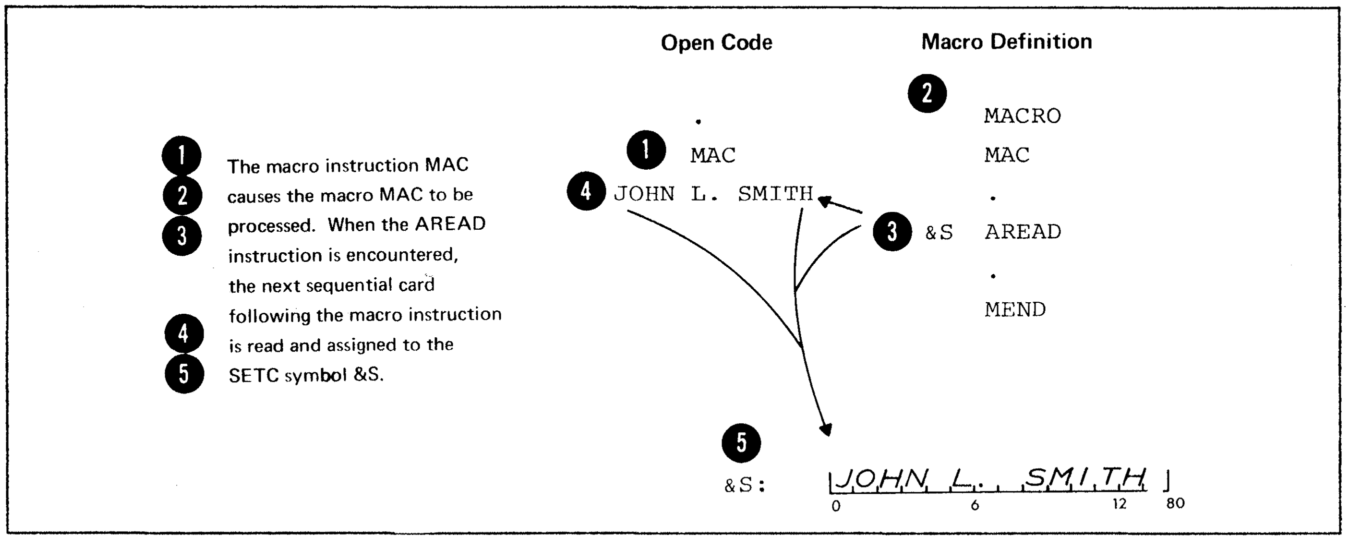
Macro instruction operation codes can be generated by substitution, either in open code or inside macro definitions. Consider the following example:

```
  Name    Operation    Operand

          MACRO
          MAC          &X
          .
          .
          &X           A,B,C   Inner macro instruction
          .
          MEND
          MACRO
          MACALL
          .
          MEND
          MAC          MACALL  Outer macro instruction
```

ARBITRARY LANGUAGE INPUT—AREAD

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. Figure 2 illustrates how the instruction is used:



Open Code
Macro Definition
1 2 3 The macro instruction MAC causes the macro MAC to be processed. When the AREAD instruction is encountered, the next sequential card following the macro instruction
4 5 is read and assigned to the SETC symbol &S.
1 MAC
4 JOHN L. SMITH
2 MACRO
MAC
3 &S AREAD
MEND
5 &S: JOHN L. SMITH
0 6 12 80

Figure 2. AREAD Assembler Operation

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 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 input/output capability. The total input/output capability of macros is 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
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

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 LANGUAGE EXTENSIONS

The following rules apply to other 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 must be alphabetic, not an ampersand or a period.
- Comments (both '*' and '.*' types) can be inserted between the macro header and the prototype and, for library macros, before the macro header. 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 S/370, Extended Architecture, and Universal instruction sets, or any assembler operation code, can be defined as a macro instruction. When one 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.

CONDITIONAL ASSEMBLY INSTRUCTION EXTENSIONS

The flexibility of the AIF, AGO, SETA, SETB, and SETC instructions is increased 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 (meaning SETA, SETA, 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.

AIF STATEMENTS

The AIF statement 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.

AGO STATEMENTS

One AGO statement 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 "Kth" 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
```

SETX STATEMENTS

The SETA, SETB, and SETC statements are used to assign arithmetic, binary, and character values, respectively, to SET variable symbols. You can use the SET statement to assign lists or arrays of values to subscripted SET symbols. For example, a list of 100 SETx values 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 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

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

- Global and local SET symbol declarations are processed at generation time in the assembly process. 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.

- 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. All such SET symbol arrays are open-ended; the subscript value specified in the declaration does not limit the size of the array. This is shown in the following example:

```
Name      Operation    Operand

          LCLA         &J(50)
&J(45)    SETA         415      Allowed under both assemblers
&J(89)    SETA         38       Allowed only under Assembler H
```

CREATED SET SYMBOLS

SET symbols may 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 pushdown 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. This allows you to associate numeric values with EBCDIC or hexadecimal characters, and 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

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

Forward Attribute Reference

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, and if it appears before the "true" symbol, the forward scan will attempt to evaluate that card instead.

Attribute Reference 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

Note: You can specify the underscore character in ordinary symbols; that is, symbols which can be used in the name and operand fields of machine and assembler instructions. It must not appear in an external 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.

Defined Attribute (D')

The defined attribute (D') 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.

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. Optionally, comments are inserted between the blank and column 72. The last line terminates the series with a blank in column 72.

The extended AGO and 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')
```

SYSTEM VARIABLE SYMBOLS

System variable symbols are local variable symbols that are assigned values by the assembler when they are encountered. &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, DESECT, 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 "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
```

CHAPTER 3. BASIC ASSEMBLER LANGUAGE EXTENSIONS

This chapter covers the extensions to VS Assembler for Assembler H.

REVISED ASSEMBLER OPERATIONS

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

OPSYN INSTRUCTION EXTENSION

You can code OPSYN instructions anywhere in your source module.

EQU INSTRUCTION EXTENSION

Symbols appearing in the first operand of the EQU instruction need not have been previously defined. 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

Any number of "nestings"—COPY instructions within code that has been brought into your program by another COPY instruction—is permitted.

CNOP INSTRUCTION EXTENSION

There is no restriction that symbols in the operand field of a CNOP instruction must have been previously defined.

ISEQ INSTRUCTION EXTENSION

Sequence checking of any column on input cards is allowed.

NEW ASSEMBLER OPERATIONS

New assembler operations have been added to Assembler H. The new operations are described in the following sections.

ADDRESS MODE (AMODE) AND RESIDENCE MODE (RMODE)

The AMODE and RMODE assembler attributes are included only in the Extended Architecture and Universal instruction sets (not in the S/370 instruction set). AMODE sets the entry point of the addressing mode, and RMODE, that of the residence mode.

The format of the AMODE attribute contains a name field (which associates the addressing mode with a control section), the operation (AMODE), and one of three operands (24, 31, or ANY). 24 specifies that an addressing mode within the 24-bit range is associated with the control section; 31, an addressing mode within the 31-bit range; and ANY, an addressing mode within either the 24- or the 31-bit range.

The format of the RMODE attribute contains a name field (which associates the residence mode with a control section), the operation (RMODE), and one of two operands (24 or ANY). 24 specifies that a residence mode within the 24-bit range is associated with the control section, and ANY, a residence mode of ANY.

Any field of these attributes may be generated by a macro, or by substitution in open code.

AMODE and RMODE attributes may be specified anywhere in the assembly. If the name field in either attribute is left blank, there must be an unnamed control section in the assembly. These attributes do not initiate an unnamed control section.

CHANNEL COMMAND WORDS (CCW1 AND CCW0)

Two new assembler instructions, CCW1 and CCW0, are included only in the Extended Architecture and Universal instruction sets. CCW1 results in a Format 1 channel command word that allows a 31-bit data address; CCW0 results in a Format 0 channel command word that allows a 24-bit data address.

Notes:

1. The CCW instruction, which is included in all instruction sets, also results in a Format 0 channel command word. The format of the CCW1 and CCW0 instructions, like that of a CCW instruction, contains a name field, the operation, and an operand (which has a command code, data address, flags, and data count).

2. If EXCP or EXCPVR is used, only CCW or CCW0 is valid, because EXCP and EXCPVR do not support 31-bit data addresses in channel command words.

3. If RMODE ANY is used with CCW or CCW0, an invalid data address in the channel command word may result at execution time. See the Assembler H Version 2 Application Programming: Guide for other examples of restrictions concerning the use of RMODE ANY.

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 VS Assembler language are extended in the Assembler H language.

CONTINUATION LINES

Assembler H allows as many as nine continuation lines in an ordinary assembler language statement.

The alternate format, which allows as many continuation lines as needed, is allowed for certain instructions. These include macro prototype statements and macro instruction statements, as well as AIF, AGO, SETx, LCLx, and GBLx instructions.

SYMBOL LENGTH

A maximum of 63 characters can be used for a symbol. This limit includes the ampersand for variable symbols and the period for sequence symbols.

Because the linkage editor does not accept symbols longer than 8 characters, external symbols are limited to 8 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 8 characters.

LEVELS WITHIN EXPRESSIONS

Any number of terms of levels of parentheses in an expression is allowed.

ANOTHER LANGUAGE EXTENSION

The following language extension has been made to Version 2 of Assembler H.

UNDERSCORE IN SYMBOLS

You can specify the underscore character in ordinary symbols, that is, symbols which can be used in the name and operand fields of machine and assembler instructions.

The format of an ordinary symbol contains a one-byte alphabetic character followed by up to 62 alphameric or underscore characters. The underscore character must not appear in an external symbol. See the Assembler H Version 2 Application Programming: Guide for a list of symbol fields in which it must not appear.

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 available, and several others have been 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 3 on page 21 illustrates this procedure.

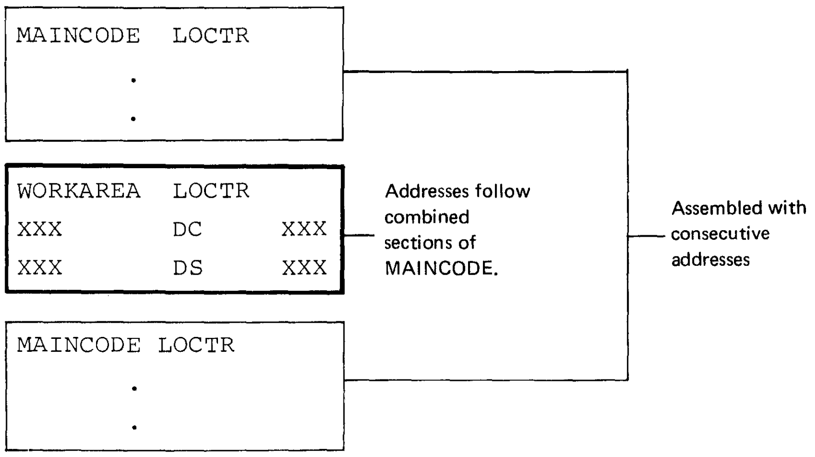

MAINCODE LOCTR
.
.
WORKAREA LOCTR
XXX DC XXX
XXX DS XXX
MAINCODE LOCTR
.
.
Addresses follow combined sections of MAINCODE.
Assembled with consecutive addresses

Figure 3. 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 have been 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

There is no restriction on 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.

CHAPTER 4. DIAGNOSTICS EXTENSIONS

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 chapter describes these diagnostic features.

DIAGNOSTIC EXTENSIONS IN REGULAR ASSEMBLY

ERROR MESSAGES

Assembler H prints in-line error messages 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 example illustrates this.

```
                         CSECT
                         .
                         COMM
***ERROR***UNDEFINED OP CODE - COMM
                         .
                         DS    (*+5)F
***ERROR***RELOCATABILITY ERROR - (*+5)F
                   1NAME DC    F'0'
***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

Diagnostic messages printed in macro-generated text are 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 example shows the placement of error messages.

```
                            MACRO        ┐
                            LIBMAC       │
                            .            │
                            LCLA    A    │   Library Macro
                            .            │
                      &B    SETA    &A   │
                            .            │
                            MEND         ┘

                            LIBMAC                          ┐
  **ERROR**                  INVALID LCLA OPERAND           │   First
                            .                               │   LIBMAC
  **ERROR**                  UNDECLARED VARIABLE SYMBOL     │   Call
                            .                               ┘

                            LIBMAC                          ┐
                            .                               │   Second
  **ERROR**                  UNDECLARED VARIABLE SYMBOL     │   LIBMAC
                            .                               │   Call
                                                            ┘
```

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. In-line edit diagnostics are inserted in the listing directly following the statement in error. Errors analyzed during macro generation produce in-line messages 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 the MHELP operand (Figure 4 on page 26).

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

1. If there are any nonzero bits present in the low-order 6 bits of the MHELP field (see Figure 3 on page 21), the corresponding options are turned on.

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

   When &SYSNDX (the total number of macro calls) exceeds the value of the fullword which contains the MHELP operand value,

MHELP OPERAND	Hexadecimal Value				MHELP Effect
		&SYSNDX	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
286¯8	0000	70	06		Macro branch trace, Macro AIF dump; &SYSNDX 28678

Figure 4. MHELP Control on &SYSNDX

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.

Note: Multiple options can be obtained by coding 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.

CHAPTER 5. FACTORS INFLUENCING IMPROVED PERFORMANCE

This chapter gives some of the factors used by Assembler H that cause performance to be improved relative to VS Assembler. The following list summarizes the factors that influence the number of input/output operations and machine instructions used by Assembler H.

- 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 are done on a demand basis in one pass of the source text.

RESIDENT TABLES AND SOURCE TEXT: Performance 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 input/output reduces system overhead and frees channels and input/output 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: 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; as a result, Assembler H may edit up to 25% fewer library macro definitions than previous assemblers.

CONSOLIDATING SOURCE TEXT PASSES: Consolidating assembly source text passes and other new organization procedures reduces the number of internal processor instructions used to handle source text in Assembler H. This is represented in proportionate savings in processor time. The saving is independent of the size or speed of the system processor involved; it is a measure of the relative efficiency of the processor.

INDEX

Assembler H Version 2:
General Information
GC26-4035-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. 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.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ________________

Previous TNL ________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

GC26-4035-0

Reader's Comment Form

Fold and tape | **Please do not staple** | Fold and tape

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

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | **Please do not staple** | Fold and tape

IBM®

Assembler H Version 2: General Information (File No. S370-21) Printed in U.S.A. GC26-4035-0

Assembler H Version 2:
General Information
GC26-4035-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. 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.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ________________________

Previous TNL ____________________

Previous TNL ____________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

GC26-4035-0

Reader's Comment Form

Fold and tape

Please do not staple

Fold and tape

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

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

old and tape

Please do not staple

Fold and tape

IBM®

GC26-4035-0

Assembler H Version 2: General Information (File No. S370-21) Printed in U.S.A. GC26-4035-0