High Level Assembler for MVS® & VM & VSE

General Information

Release 3

GC26-4943-02

High Level Assembler for MVS® & VM & VSE

General Information

Release 3

GC26-4943-02

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page vii.

Third Edition (September 1998)

This edition applies to IBM High Level Assembler for MVS & VM & VSE, Release 3, Program Number 5696-234 and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department BWE/H3
P.O.Box 49023
SAN JOSE, CA 95161-9023
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1981, 1998. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1981, 1998

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR

PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

CICS
BookMaster
DFSMS/MVS
Enterprise System/9000
Enterprise Systems Architecture/370
Enterprise Systems Architecture/390
ES/9000
ESA/390
IBM
IBMLink
IMS
MVS
MVS/DFP
MVS/ESA
MVS/XA
OpenEdition
OS/390
OS/2
RETAIN
QMF
S/370
SP
System/370
System/390
VM/ESA
VTAM
3090

Other company, product, and service names may be trademarks or service marks of others.

About this Manual

This book contains general information about IBM High Level Assembler for MVS & VM & VSE, Licensed Program 5696-234, hereafter referred to as High Level Assembler, or simply the assembler.

This book is designed to help you evaluate High Level Assembler for your data processing operation and to plan for its use.

Who Should Use this Manual

High Level Assembler General Information helps data processing managers and technical personnel evaluate High Level Assembler for use in their organization. This manual also provides an introduction to the High Level Assembler Language for system programmers and application programmers.

The assembler language supported by High Level Assembler has functional extensions to the languages supported by Assembler H Version 2 and DOS/VSE Assembler. To fully appreciate the features offered by High Level Assembler you should be familiar with either Assembler H Version 2 or DOS/VSE Assembler.

Organization of this Manual

This manual is organized as follows:

- **Chapter 1, What's New in High Level Assembler Release 3,** gives a summary of the features and enhancements introduced in High Level Assembler Release 3.
- **Chapter 2, Introduction to High Level Assembler,** gives a summary of the main features of the assembler and its purpose.
- **Chapter 3, Assembler Language Extensions,** describes the major extensions to the basic assembler language provided by High Level Assembler, and not available in earlier assemblers.
- **Chapter 4, Macro and Conditional Assembly Language Extensions,** briefly describes some of the features of the macro and conditional assembly language, and the extensions to the macro and conditional assembly language provided by High Level Assembler that were not available in earlier assemblers.
- **Chapter 5, Using Exits to Complement File Processing,** describes the facilities in the assembler to support user-supplied input/output exits, and how these might be used to complement the output produced by High Level Assembler.
- **Chapter 6, Programming and Diagnostic Aids,** describes the many assembly listing and diagnostic features that High Level Assembler provides to help in the development of assembler language programs and the location and analysis of program errors.
- **Chapter 7, Associated Data Architecture,** gives a summary of the Associated Data Architecture, and the associated data file produced by High Level Assembler.

© Copyright IBM Corp. 1981, 1998

- **Chapter 8, Factors Improving Performance,** describes some of the methods used by High Level Assembler to improve performance relative to earlier assemblers.
- **Appendix A, Assembler Options,** lists and describes the assembler options you can specify with High Level Assembler.
- **Appendix B, System Variable Symbols,** lists and describes the system variable symbols provided by High Level Assembler.
- **Appendix C, Hardware and Software Requirements,** provides information about the operating system environments in which High Level Assembler will operate.
- The **Bibliography** lists other IBM publications which may serve as a useful reference to this book.

Throughout this book, we use these indicators to identify platform-specific information:

- Prefix the text with platform-specific text (for example, "Under CMS...")
- Add parenthetical qualifications (for example, "(CMS only)")
- Bracket the text with icons. The following are some of the icons that we use:

  MVS Informs you of information specific to MVS MVS

  CMS Informs you of information specific to CMS CMS

  VSE Informs you of information specific to VSE VSE

MVS is used in this manual to refer to Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™) and to OS/390®.

CMS is used in this manual to refer to Conversational Monitor System on Virtual Machine/Enterprise Systems Architecture (VM/ESA®).

VSE is used in this manual to refer to Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA®).

Hardcopy Publications

General Information is one book in a library of books for High Level Assembler. The following table names the books in the library and shows which books can help you with specific tasks, such as evaluating High Level Assembler.

Task	Publication	Order Number
Evaluation and Planning	General Information	GC26-4943
Installation and Customization	Installation and Customization Guide	SC26-3494
	Programmer's Guide	SC26-4941
	Toolkit Feature Installation Guide	GC26-8711
Application Programming	Programmer's Guide	SC26-4941
	Language Reference	SC26-4940
	General Information	GC26-4943
	Toolkit Feature User's Guide	GC26-8710
	Toolkit Feature IDF User's Guide	GC26-8709
Diagnosis	Installation and Customization Guide	SC26-3494
Warranty	Licensed Program Specifications	GC26-4944

In addition to this General Information book, the following High Level Assembler publications are available:

Installation and Customization Guide
: Contains the information you need to install and customize, and diagnose failures in, the High Level Assembler product.

  The diagnosis section of the book helps users determine if a correction for a similar failure has been documented previously. For problems not documented previously, the book helps users to prepare an APAR. This section is for users who suspect that High Level Assembler is not working correctly because of some defect.

Language Reference
: Presents the rules for writing assembler language source programs to be assembled using High Level Assembler.

Licensed Program Specifications
: Contains a product description and product warranty information for High Level Assembler.

Programmer's Guide
: Describes how to assemble, debug, and run High Level Assembler programs.

Toolkit Feature Installation Guide
: Contains the information you need to install and customize, and diagnose failures in, the High Level Assembler Toolkit Feature.

Toolkit Feature User's Guide
: Describes how to use the High Level Assembler Toolkit Feature.

Toolkit Feature IDF Reference Summary
: Contains a reference summary of the High Level Assembler Interactive Debug Facility.

Toolkit Feature IDF User's Guide
Describes how to use the High Level Assembler Interactive Debug Facility.

Online Publications

The High Level Assembler publications are available in the following softcopy formats:

- *Application Development Collection Kit* CD-ROM, SK2T-1237
- *MVS Collection* CD-ROM, SK2T-0710
- *OS/390 Collection* CD-ROM, SK2T-6700
- *VM/ESA Collection* CD-ROM, SK2T-2067
- *VSE Collection* CD-ROM, SK2T-0060

| For more information about High Level Assembler, see the High Level Assembler
| web site, at

| `http://www.software.ibm.com/ad/hlasm`

Chapter 1. What's New in High Level Assembler Release 3

High Level Assembler Release 3 provides enhancements over High Level Assembler Release 2 in the following areas:

- Installation and customization
- Performance and usability
- Programmer productivity
- Diagnostic information

This chapter describes the major new features provided in High Level Assembler Release 3. Technical changes to this manual are marked in the text by a change bar in the left margin.

Performance and Usability

The following enhancements improve system performance and system usability:

Binary floating-point: The new binary floating-point instructions and data formats are supported. The support includes:

- Accurate conversion of decimal values to binary floating-point representations, with selectable rounding modes.
- Special values, such as infinities and NaNs, may be requested with symbolic forms.
- All new binary and hexadecimal floating-point instructions are supported.

ADATA Register Cross Reference Record: A new ADATA register cross reference record holds details of registers and their usage.

EXIT enhancements. The enhanced EXIT interface allows the user to write one routine that handles multiple exits, without needing elaborate schemes for inter-exit communication. It is also easier to write a single exit that handles multiple I/O types.

ACONTROL: The new ACONTROL statement allows fine-grained controls over certain options within the source program. Its status can be saved and restored with the PUSH and POP instructions.

USING: The USING statement has an additional parameter, the end parameter. When this parameter is supplied, it specifies a range to override the default range.

Register Cross Reference listing: The new section, General Purpose Cross Reference, is added to the assembler listing. This provides extra diagnostic information.

Toolkit Feature: The tools of the optional Toolkit Feature have been enhanced to cater for the additional floating-point registers.

© Copyright IBM Corp. 1981, 1998

Conditional Assembly Language: Enhancements to the conditional assembly language include:

- A new AINSERT statement that allows creation of records to be inserted into the assembler's input stream.
- BYTE and SIGNED internal functions that simplify conditional assembly expressions.
- Five new system variable symbols:

&SYSCLOCK	Provides detailed date and time information.
&SYSMAC	Provides the name of the macro in which it is used, and the names of all macros in the call chain.
&SYSOPT_XOBJECT	Indicates that the XOBJECT option was specified.
&SYSM_SEV	Provides the highest severity codes from MNOTE statements in the most recently called macro.
&SYSM_HSEV	Provides the highest severity codes from MNOTE statements in the entire assembly.

Programmer Productivity

The following enhancements simplify program development and increase programmer productivity:

Assembler Listing: The assembler listing is changed, to improve readability, and to provide more information to the programmer:

- Allows variable record format for the listing file (MVS and CMS).
- Changes the source and object section:
  - Adds the PUSH level to the USING heading.
  - Provides start and length information in the ADDR1 and ADDR2 fields for CSECT, START, LOCTR, and RSECT statements.
  - Provides current and next address information in the ADDR1 and ADDR2 fields for the ORG statement.
  - Provides the value and length information in the ADDR1 and ADDR2 fields for the EQU statement.
  - Provides additional statement type information in the position following the statement number.
- Changes the symbol and cross reference section:
  - Removes leading zeroes on lengths, to accentuate the decimal notation.
  - Changes statement reference information to columnar.
  - Uses the full width, that is, 121 or 133 characters, if specified.
- Provides a new optional section, the cross reference listing of General Purpose register usage.
- Provides a new External Function Statistics table, as part of the Diagnostic Cross Reference and Assembler Summary page.

Diagnostic Information

- Extra warning messages have been provided. These highlight behavior that may lead to unexpected results.

Chapter 2. Introduction to High Level Assembler

High Level Assembler is an IBM licensed program that helps you develop programs and subroutines to provide functions not typically provided by other symbolic languages, such as COBOL, FORTRAN, and PL/I.

Language Compatibility

The assembler language supported by High Level Assembler has functional extensions to the languages supported by Assembler H Version 2 and DOS/VSE Assembler. High Level Assembler uses the same language syntax, function, operation, and structure as these earlier assemblers. The functions provided by the Assembler H Version 2 macro facility are all provided by High Level Assembler.

Migration from Assembler H Version 2 or DOS/VSE Assembler to High Level Assembler requires an analysis of existing assembler language programs to ensure that they do not contain macro instructions with names that conflict with the High Level Assembler symbolic operation codes, or SET symbols with names that conflict with the names of High Level Assembler system variable symbols.

With the exception of these possible conflicts, and with appropriate High Level Assembler option values, assembler language source programs written for Assembler H Version 2 or DOS/VSE Assembler, that assemble without warning or error diagnostic messages, should assemble correctly using High Level Assembler.

High Level Assembler, like its predecessor Assembler H Version 2, can assemble source programs that use the following machine instructions:

- System/370
- System/370 Extended Architecture (370-XA)
- Enterprise Systems Architecture/370 (ESA/370)
- Enterprise Systems Architecture/390 (ESA/390) .

The set of machine instructions that you can use in an assembler source program depend upon which operation code table you use for the assembly.

Highlights of High Level Assembler

High Level Assembler is a functional replacement for Assembler H Version 2 and DOS/VSE Assembler. It offers all the proven facilities provided by these earlier assemblers, and many new facilities designed to improve programmer productivity and simplify assembler language program development and maintenance.

Some of the highlights of High Level Assembler are:

- Extensions to the basic assembler language
- Extensions to the macro and conditional assembly language, including external function calls and built-in functions
- Enhancements to the assembly listing, including a new macro and copy code member cross reference section, and a new section that lists all the unreferenced symbols defined in CSECTs.
- New assembler options

- A new associated data file, the ADATA file, containing both language-dependent and language-independent records that can be used by debugging and other tools
- A DOS operation code table to assist in migration from DOS/VSE Assembler
- The use of 31-bit addressing for most working storage requirements
- An extended object format data set
- Internal performance enhancements and diagnostic capabilities

This book contains a summary of information designed to help you evaluate the High Level Assembler licensed product. For more detailed information, see *High Level Assembler Programmer's Guide* and *High Level Assembler Language Reference*.

The Toolkit Feature

The optional High Level Assembler Toolkit Feature provides a powerful and flexible set of tools to improve application recovery and development. The tools include XREF, ASMPUT, the Disassembler, the Interactive Debug Facility, and Enhanced SuperC.

Planning for High Level Assembler

The assembler language and macro language extensions provided by High Level Assembler include functional extensions to those provided by Assembler H Version 2 and the DOS/VSE assembler. The following chapters and appendices help you evaluate these extensions, and plan the installation and customization process. They include:

- A description of the language differences and enhancements that will help you decide if there are any changes you need to make to existing programs.
- A summary of the assembler options to help you decide which ones are appropriate to your installation.
- A summary of the system variable symbols to help you determine if they conflict with symbols already defined in your programs.
- A description of the hardware and software required to install and run High Level Assembler.

Year 2000 Support for High Level Assembler

High Level Assembler is available as an element of OS/390. OS/390 is certified as a Year 2000 ready operating system by the Information Technology Association of America (ITAA).

Chapter 3. Assembler Language Extensions

The instructions, syntax and coding conventions of the assembler language supported by High Level Assembler include functional extensions to those supported by Assembler H Version 2 and DOS/VSE Assembler. This chapter describes the most important of those extensions, and the language differences between High Level Assembler and the earlier assemblers.

Additional Assembler Instructions

The following additional assembler instructions are provided with High Level Assembler:

PROCESS Statement: The *PROCESS statement lets you specify assembler options in the assembler source program. See “Specifying Assembler Options in the Source Program” on page 17.

ACONTROL Instruction: The ACONTROL instruction lets you change many assembler options within a program.

ADATA Instruction: The ADATA instruction allows user records to be written to the associated data file.

AINSERT Instruction: The AINSERT instruction allows creation of records to be inserted into the assembler's input stream.

ALIAS Instruction: The ALIAS instruction lets you replace an external symbol name with a string of up to 64 bytes.

Binary Floating-Point (BFP) instructions: The assembler supports the new binary floating-point instructions and the additional floating-point registers.

CEJECT Instruction: The CEJECT instruction allows page ejects to be done conditionally, under operand control.

CATTR Instruction (MVS and CMS)***:*** You can use the CATTR instruction to establish a program object external class name, and assign binder attributes for the class. This instruction is only valid when you specify the XOBJECT assembler option to produce extended object format modules. See “Extended Object Format Modules (MVS and CMS)” on page 12. By establishing the deferred load attribute, text is not loaded when the program is brought into storage, but is partially loaded, for fast access when it is requested.

EXITCTL Instruction: The EXITCTL instruction allows data to be passed from the assembler source to any of the input/output user exits. See Chapter 5, “Using Exits to Complement File Processing” on page 39.

RSECT Instruction: The previously undocumented instruction, RSECT, which defines a read-only control section, has been documented. See “Read-Only Control Sections” on page 14.

Revised Assembler Instructions

Several assembler instructions used in earlier assemblers have been extended in High Level Assembler.

CNOP Instruction: Symbols in the operand field of a CNOP instruction do not need to be previously defined.

COPY Instruction: Any number of *nestings* (COPY instructions within code that has been brought into your program by another COPY instruction) is permitted. However, recursive COPY instructions are not permitted.

A variable symbol that has been assigned a valid ordinary symbol may be used as the operand of a COPY instruction in open code:

```
 &VAR      SETC 'LIBMEM'
          COPY &VAR
+         COPY LIBMEM                          Generated Statement
```

| ***DC Instruction:*** The DC instruction has been enhanced to cater for the new binary floating-point numbers. As well, the J-type address constant and Q-type address constant have been added, and more characters allowed in the nominal value.

DROP Instruction: The DROP instruction now lets you end the domain of labeled USINGs and labeled dependent USINGs. See “Labeled USINGs and Qualified Symbols” on page 15 and “Dependent USINGs” on page 16.

| ***DXD Instruction:*** The DXD instruction now aligns external dummy sections to the most restrictive alignment of the specified operands (instead of that of the first operand).

EQU Instruction: Symbols appearing in the first operand of the EQU instruction do not need to be previously defined. In the following example, both `WIDTH` and `LENGTH` can be defined later in the source code:

```
Name     Operation      Operand

VAL      EQU            40-WIDTH+LENGTH
```

ISEQ Instruction: Sequence checking of any column on input records is allowed.

OPSYN Instruction: You can code OPSYN instructions anywhere in your source module.

POP Instruction: An additional operand, NOPRINT, can be specified with the POP instruction to cause the assembler to suppress the printing of the specified POP statement. | The operand ACONTROL saves the ACONTROL status.

PRINT Instruction: Seven additional operands can be specified with the PRINT instruction. They are:

MCALL|NOMCALL
: The MCALL operand instructs the assembler to print nested macro call instructions.

The NOMCALL operand suppresses the printing of nested macro call instructions.

MSOURCE|NOMSOURCE
: The MSOURCE operand causes the assembler to print the source statements generated during macro processing, as well as the assembled addresses and generated object code of the statements.

  The NOMSOURCE operand suppresses the printing of the generated source statements, but does not suppress the printing of the assembled addresses and generated object code.

UHEAD|NOUHEAD
: The UHEAD operand causes the assembler to print a summary of active USINGs following the TITLE line on each page of the source and object program section of the assembler listing.

  The NOUHEAD operand suppresses the printing of this summary.

NOPRINT
: The NOPRINT operand causes the assembler to suppress the printing of the PRINT statement that is specified.

The assembler has changed the way generated object code is printed in the assembler listing when the PRINT NOGEN instruction is used. Now the object code for the first generated instruction, or the first 8 bytes of generated data is printed in the *object code* column of the listing on the same line as the macro call instruction. The DC, DS, DXD, and CXD instructions can cause the assembler to generate zeros as alignment data. With PRINT NOGEN the generated alignment data is not printed in the listing.

PUSH Instruction: An additional operand, NOPRINT, can be specified with the PUSH instruction to cause the assembler to suppress the printing of the specified PUSH statement. The operand ACONTROL restores the ACONTROL status.

USING Statements: Labeled USINGs and dependent USINGs provide you with enhanced control over the resolution of symbolic expressions into base-displacement form with specific base registers. Dependent USINGs can be labeled or unlabeled.

The end of range parameter lets you specify a range for the USING statement, rather than accepting the default range. See “Labeled USINGs and Qualified Symbols” on page 15 and “Dependent USINGs” on page 16.

2-Byte Relocatable Address Constants

The assembler now accepts 2 as a valid length modifier for relocatable A-type address constants, such as AL2(*). A 2-byte, relocatable, A-type address constant is processed in the same way as a Y-type relocatable address constant, except that no boundary alignment is provided.

Character Set Support Extensions

High Level Assembler provides support for both standard single-byte characters and double-byte characters.

Standard Character Set

The standard character set used by High Level Assembler is EBCDIC. A subset of the EBCDIC character set can be used to code terms and expressions in assembler language statements.

In addition, all EBCDIC characters can be used in comments and remarks, and anywhere that characters can appear between paired single quotation marks.

Double-Byte Character Set

In addition to the standard EBCDIC set of characters, High Level Assembler accepts double-byte character set (DBCS) data.

When the DBCS option is specified, High Level Assembler accepts double-byte data as follows:

- Double-byte data, optionally mixed with single-byte data, is permitted in:
  - The nominal value of character (C-type) constants and literals
  - The value of character (C-type) self-defining terms
  - The operand of MNOTE, PUNCH and TITLE statements
- Pure double-byte data is supported by:
  - The pure DBCS (G-type) constant and literal
  - The pure DBCS (G-type) self-defining term

Double-byte data in source statements must always be bracketed by the *shift-out* (SO) and *shift-in* (SI) characters to distinguish it from single-byte data.

Double-byte data is supported in the operands of the AREAD and REPRO statements, and in comments and remarks, regardless of the invocation option. Double-byte data assigned to a SETC variable symbol by an AREAD statement contain the SO and SI.

Translation Table

In addition to the standard EBCDIC set of characters, High Level Assembler can use a user-specified translation table to convert the characters contained in character (C-type) data constants (DCs) and literals. High Level Assembler provides a translation table to convert the EBCDIC character set to the ASCII character set. The assembler can also use a translation table supplied by the programmer.

Assembler Language Syntax Extensions

The syntax of the assembler language deals with the structure of individual elements of any instruction statement, and with the order that the elements are presented in that statement. Several syntactical elements of earlier assembler languages are extended in the High Level Assembler language.

Blank Lines

High Level Assembler allows blank lines to be used in the source program. In *open code*, each blank line is treated as equivalent to a SPACE 1 statement. In the body of a *macro definition*, each blank line is treated as equivalent to an ASPACE 1 statement.

Comment Statements

A *macro comment* statement consists of a period in the begin column, followed by an asterisk, followed by any character string. An *open code* comment consists of an asterisk in the begin column followed by any character string.

High Level Assembler allows open code statements to use the macro comment format, and processes them like an open code comment statement.

Mixed-case Input

High Level Assembler allows mixed-case input statements, and maintains the case when it produces the assembler listing. You can use the COMPAT and FOLD assembler options to control how the assembler treats mixed-case input.

Continuation Lines

You are allowed as many as nine continuation lines for most ordinary assembler language statements. However, you are allowed to specify as many continuation lines as you need for the following statements:

- Macro prototype statements
- Macro instruction statements
- The AIF, AGO, SETx, LCLx, and GBLx conditional assembly instructions.

When you specify the FLAG(CONT) assembler option, the assembler issues new warning messages if it suspects that a continuation statement might be incorrect.

Continuation Lines and Double-byte Data

If the assembler is called with the DBCS option, then:

- When an SI occurs in the end column of a continued line, and an SO occurs in the continue column of the next line, the SI and SO are considered redundant and are removed from the statement before the statement is analyzed.
- An extended continuation indicator provides you with a flexible end column on a line-by-line basis so that any alignment of double-byte data in a source statement can be supported.

Continuation Error Warning Messages

The FLAG(CONT) assembler option directs the assembler to issue warning messages for continuation statement errors for macro calls in the following circumstances:

- The operand on the continued record ends with a comma and a continuation statement is present but continuation does not start in the continue column (usually column 16).
- A list of one or more operands ends with a comma, but the continuation column (usually column 72) is blank.

- The continuation record starts in the continue column (usually column 16) but there is no comma present following the operands on the previous record.
- The continued record is full but the continuation record does not start in the continue column (usually column 16).

Symbol Length

High Level Assembler supports three types of symbols:

Ordinary symbols	The format of an ordinary symbol consists of an alphabetic character, followed by a maximum of 62 alphanumeric characters.
Variable symbols	The format of a variable symbol consists of an ampersand (&) followed by an alphabetic character, followed by a maximum of 61 alphanumeric characters.
Sequence symbols	The format of a sequence symbol consists of a period (.) followed by an alphabetic character, followed by a maximum of 61 alphanumeric characters.

External symbols are ordinary symbols used in the name field of START, CSECT, RSECT, COM, DXD, and ALIAS statements, and in the operand field of ENTRY, EXTRN, WXTRN, and ALIAS statements. Symbols used in V-type and Q-type address constants are restricted to 8 characters. You can specify an alias string of up to 64 characters to represent an external symbol.

Underscore

High Level Assembler accepts the underscore character as alphabetic. It is accepted in any position in any symbol name.

Literals

The following changes have been made to previous restrictions on the use of literals:

- Literals can be used as relocatable terms in expressions. They no longer have to be used as a complete operand.
- Literals can be used in RX-format instructions in which an index register is used.

Levels within Expressions

The number of terms or levels of parentheses in an expression is limited by the storage buffer size allocated by the assembler for its evaluation work area.

Extended Object Format Modules (MVS and CMS)

High Level Assembler provides support for extended object format modules. The XOBJECT assembler option instructs the assembler to produce the extended object data set. The following new or modified instructions support the generation of the extended object format records:

- ALIAS
- AMODE

- CATTR

For further details about this facility refer to *DFSMS/MVS Program Management*, SC26-4916.

Extended Addressing Support

High Level Assembler provides several instructions for the generation of object modules that exploit extended addressing. These instructions are:

- AMODE
- RMODE
- CCW0
- CCW1

Addressing Mode (AMODE) and Residence Mode (RMODE)

Use the AMODE instruction to specify the addressing mode to be associated with the control sections in the object program. The addressing modes are:

24	24-bit addressing mode
31	31-bit addressing mode
ANY	Either 24-bit or 31-bit addressing mode

Use the RMODE instruction to specify the residence mode to be associated with the control sections in the object program. The residence modes are:

24	Residence mode of 24. The control section must reside below the 16MB line.
ANY	Residence mode of either 24 or 31. The control section can reside above or below the 16MB line.

You can specify the AMODE and RMODE instructions anywhere in the assembly source. If the name field in either instruction is left blank, you must have an unnamed control section in the assembly. These instructions do not initiate an unnamed control section.

Channel Command Words (CCW0 and CCW1)

The CCW0 instruction performs the same function as the CCW instruction, and is used to define and generate a format-0 channel command word that allows a 24-bit data address. The CCW1 instruction result is used to define and generate a format-1 channel command word that allows a 31-bit data address.

The format of the CCW0 and CCW1 instructions, like that of the CCW instruction, consists of a name field, the operation, and an operand (that contains a command code, data address, flags, and data count).

Using EXCP or EXCPVR access methods: If you use the EXCP or EXCPVR access method, only CCW or CCW0 is valid, because EXCP and EXCPVR do not support 31-bit data addresses in channel command words.

Using RMODE ANY: If you use RMODE ANY with CCW or CCW0, an invalid data address in the channel command word can result at execution time.

Programming Sectioning and Linking Controls

High Level Assembler provides several facilities that allow increased control of program organization. These include:

- Read-only control sections
- Multiple location counters
- External dummy sections
- Support for up to 65535 external symbols

Read-Only Control Sections

With the RSECT instruction, you can initiate a read-only executable control section, or continue a previously initiated read-only executable control section.

When a control section is initiated by the RSECT instruction, the assembler automatically checks the control section for non-reentrant code. As the assembler cannot check program logic, the checking is not exhaustive. If the assembler detects non-reentrant code it issues a warning message.

The read-only attribute in the object module shows which control sections are read-only.

Multiple Location Counters

Multiple location counters are defined in a control section by using the LOCTR instruction. The assembler assigns consecutive addresses to the segments of code using one location counter before it assigns addresses to segments of code using 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 1 shows this procedure.

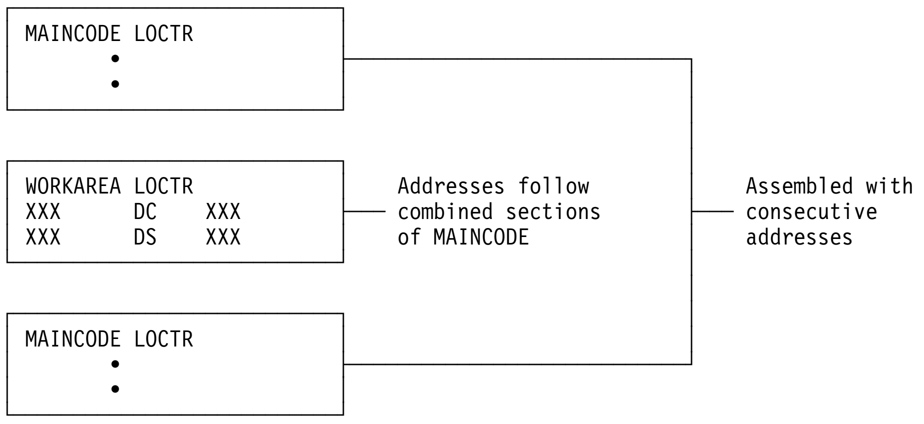
MAINCODE LOCTR
•
•
WORKAREA LOCTR
XXX DC XXX
XXX DS XXX
Addresses follow combined sections of MAINCODE
Assembled with consecutive addresses
MAINCODE LOCTR
•
•

Figure 1. LOCTR Instruction Application

External Dummy Sections

An *external dummy section* is a reference control section that you can use to describe a communication area between two or more object modules that are link-edited together. The assembler generates an external dummy section when you define a Q-type address constant that contains the name of a reference control section specified in a DXD or DSECT instruction.

VSE External dummy sections are only supported by VSE/ESA Version 2 Release 2 and Version 2 Release 3. VSE

Number of External Symbols

The assembler can support up to 65535 independently relocatable items. Such items include control section names, names declared in EXTRNs and so forth. The names of some of these items can appear in the external symbol dictionary (ESD) of the assembler's object module. Note that other products might not be able to handle as many external symbols as the assembler can produce.

Assembler instructions that can produce independently relocatable items and appear in the ESD are:

- START
- CSECT
- RSECT
- COM
- DXD
- EXTRN
- WXTRN
- ALIAS
- CATTR
- V-type address constant
- DSECT if the DSECT name appears in a Q-type address constant

Many instructions can cause the initiation of an unnamed CSECT if they appear before a START or CSECT statement. Unnamed CSECTs appear in the external symbol dictionary with a type of PC.

Addressing Extensions

High Level Assembler extends the means that you can use to establish addressability of a control section with two powerful new facilities:

- Labeled USINGs and qualified symbols
- Dependent USINGs

Labeled USINGs and Qualified Symbols

The format of the assembler USING instruction now lets you code a symbol in the name entry of the instruction. When a valid ordinary symbol, or a variable symbol that has been assigned a valid ordinary symbol, is specified in the name entry of a USING instruction, it is known as the *USING label*, and the USING is known as a labeled USING.

Labeled USINGs provide you with enhanced control over the resolution of symbolic expressions into base-displacement form with specific base registers. The assembler uses a labeled USING when you qualify a symbol with the USING label. You qualify a symbol by prefixing the symbol with the label on the USING followed by a period.

Labeled USING Domains

You can specify the same base register or registers in any number of labeled USING instructions. However, unlike ordinary USING instructions, as long as all the labeled USINGs have unique labels, the assembler considers the domains of all the labeled USINGs to be active and their labels can be used as qualifiers. With ordinary USINGs, when you specify the same base register in a subsequent USING instruction, the domain of the prior USING is ended.

The domain of a labeled USING instruction continues until the end of a source module, except when:

- You specify the label in the operand of a subsequent DROP instruction.
- You specify the same label in a subsequent USING instruction.

Labeled USING Ranges

You can specify the same base address in any number of labeled USING instructions. You can also specify the same base address in an ordinary USING and a labeled USING. However, unlike ordinary USING instructions that have the same base address, if you specify the same base address in an ordinary USING instruction and a labeled USING instruction, the assembler does not treat the USING ranges as coinciding. When you specify an unqualified symbol in an assembler instruction, the assembler uses the base register specified in the ordinary USING to resolve the address into base-displacement form. You can specify an optional parameter on the USING instruction. This option sets the range of the USING, overwriting the default of 4096.

Dependent USINGs

The format of the assembler USING instruction now lets you specify a relocatable expression instead of a base register in the instruction operand. When you specify a relocatable expression, it is known as the *supporting base address*, and the USING is known as a *dependent USING*. If a valid ordinary symbol, or a variable symbol that has been assigned a valid ordinary symbol, is specified in the name entry of a dependent USING instruction, the USING is known as a *labeled dependent USING*.

A dependent USING depends on the presence of one or more corresponding ordinary or labeled USINGs to resolve the symbolic expressions in the dependent USING range.

Dependent USINGs provide you with further control over the resolution of symbolic expressions into base-displacement form. With dependent USINGs you can reduce the number of base registers you need for addressing by using an existing base register to provide addressability to the symbolic address.

Dependent USING Domains

The domain of a dependent USING begins where the dependent USING instruction appears in the source module and continues until the end of the source module, except when:

- You end the domain of the corresponding ordinary USING by specifying the base register or registers from the ordinary USING instruction in a subsequent DROP instruction.
- You end the domain of the corresponding ordinary USING by specifying the same base register or registers from the ordinary USING instruction in a subsequent ordinary USING instruction.
- You end the domain of a labeled dependent USING by specifying the label of the labeled dependent USING in the operand of a subsequent DROP instruction.

Dependent USING Ranges

The range of a dependent USING is 4096 bytes, or as limited by the end operand, beginning at the base address specified in the corresponding ordinary or labeled USING instruction. If the corresponding ordinary or labeled USING assigns more than one base register, the dependent USING range is the composite USING range of the ordinary or labeled USING.

If the dependent USING instruction specifies a supporting base address that is within the range of more than one ordinary USING, the assembler determines which base register to use during base-displacement resolution as follows:

- The assembler computes displacements from the ordinary USING base address that gives the smallest displacement, and uses the corresponding base register.
- If more than one ordinary USING gives the smallest displacement, the assembler uses the higher-numbered register for assembling addresses within the coinciding USING ranges.

Specifying Assembler Options in the Source Program

Process (*PROCESS) statements let you specify selected assembler options in the assembler source program. You can include them in the primary input data set or provide them from a SOURCE user exit.

You can specify a maximum of 10 process statements in one assembly. After processing 10 process statements, the assembler treats the next input record as an ordinary assembler statement; in addition the assembler treats further process statements as comment statements. You cannot continue a process statement from one statement to the next.

When the assembler detects an error in a process statement, it produces one or more warning messages. If the installation default option PESTOP is set, then the assembler stops after it finishes processing any process statements.

| The ACONTROL instruction lets you specify selected assembler options anywhere
| through the assembler source program, rather than at the beginning of the source
| (as provided by *PROCESS statements).

The assembler recognizes the assembler options in the following order of precedence (highest to lowest):

1. Fixed installation defaults
2. Options on the PARM parameter of the JCL EXEC statement under MVS and VSE or the High Level Assembler command under CMS
3. Options on the JCL OPTION statement (VSE only)
4. Options on *PROCESS statements
5. Non-fixed installation defaults

Options specified by the ACONTROL instruction take effect when the specifying ACONTROL instruction is encountered during the assembly. An option specified by an ACONTROL instruction may override an option specified at the start of the assembly.

The assembler lists the options specified in process statements in the *High Level Assembler Option Summary* section of the assembler listing.

Process statements are also shown as comment lines in the *source and object* section of the assembler listing.

IBM-Supplied Default Assembler Options

Figure 2 shows the changes made to the IBM-supplied default assembler options for High Level Assembler Release 3:

Figure 2. Changes to High Level Assembler Default Options

New in Release 3	Previously in Release 2
FLAG(0,ALIGN,CONT,NOIMPLEN, NOPAGE0,RECORD,NOSUBSTR)	FLAG(0,ALIGN,CONT,RECORD,NOSUBSTR)
NOINFO	Not available
RXREF	Not available

See Appendix A, “Assembler Options” on page 71 for a list of all assembler options.

Chapter 4. Macro and Conditional Assembly Language Extensions

The macro and conditional assembly language supported by High Level Assembler provides a number of functional extensions to the macro languages supported by Assembler H Version 2 and DOS/VSE Assembler. This chapter provides an overview of the language, and describes the major extensions.

The Macro Language

The macro language is an extension of the assembler language. It provides a convenient way to generate a preferred sequence of assembler language statements many times in one or more programs. There are two parts to the macro language supported by High Level Assembler:

Macro definition
A named sequence of statements you call with a macro instruction. The name of the macro is the symbolic operation code used in the macro instruction. Macro definitions can appear anywhere in your source module; they can even be nested within other macro definitions. Macros can also be redefined at a later point in your program.

Macro instruction
Calls the macro definition for processing. A macro instruction can pass information to the macro definition which the assembler uses to process the macro.

There are two types of macro definition:

Source macro definition
A macro definition defined in your source program.

Library macro definition
A macro definition that resides in a library data set.

Either type of macro definition can be called from anywhere in the source module by a macro instruction, however a source macro definition must occur before it is first called.

You use a macro prototype statement to define the name of the macro and the symbolic parameters you can pass it from a macro instruction.

General Advantages in Using Macros

The main use of a macro is to insert assembler language statements into your source program each time the macro definition is called by a macro instruction. Values, represented by positional or keyword symbolic parameters, can be passed from the calling macro instruction to the statements within the body of a macro definition. The assembler can use global SET symbols and absolute ordinary symbols created by other macros and by open code.

The assembler assigns attribute values to the ordinary symbols and variable symbols that represent data. By referencing the data attributes of these symbols, or by varying the values assigned to these symbols, you can control the logic of the

© Copyright IBM Corp. 1981, 1998

macro processing, and, in turn, control the sequence and contents of generated statements.

The assembler replaces the macro call with the statements generated from the macro definition. The generated statements are then processed like open code source statements.

Using macros gives you a flexibility similar to that provided by a problem-oriented language. You can use macros to create your own procedural language, tailored to your specific applications.

Assembler Editing of the Macro Definition

The initial processing of a macro definition is called *editing*. In editing, the assembler checks the syntax of the instructions and converts the source statements to an edited version 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. This is why a macro must always be edited, and consequently be defined, before it can be called by a macro instruction.

| VSE “Reading Edited Macros (VSE only)” on page 41 describes how you can
| use a LIBRARY exit to allow High Level Assembler to read edited macros.
| VSE

Macro Language Extensions

Extensions to the macro language include the following:

- Macro redefinition facilities
- Inner macro definitions
- Multilevel sublists in macros
- DBCS language support
- AREAD instruction to read source stream input
- Instructions to control the listing of macro definitions

Redefining Macros

You can redefine a macro definition 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.

You can save the function of the original macro definition by using the OPSYN instruction before you redefine the macro. If you want to reestablish the initial function of the operation code, you can include another OPSYN instruction to redefine it. The following example shows this:

```
Name      Operation     Operand       Comment

          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
                                      new macro definition.
          ⋮
          MEND
          ⋮
MAC1      OPSYN         MAC2          MAC1 is assigned to the first
                                      definition. The second
                                      definition is lost.
```

You can issue a conditional assembly branch (AGO or AIF) to a point before the initial definition of the macro and reestablish a previous source macro definition. Then that definition will be edited and effective for subsequent macro instructions calling it.

See “Redefining Conditional Assembly Instructions” on page 35.

Inner Macro Definitions

High Level Assembler 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 generation of the outer macro. 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 3 on page 22 shows the editing of inner macro definitions.

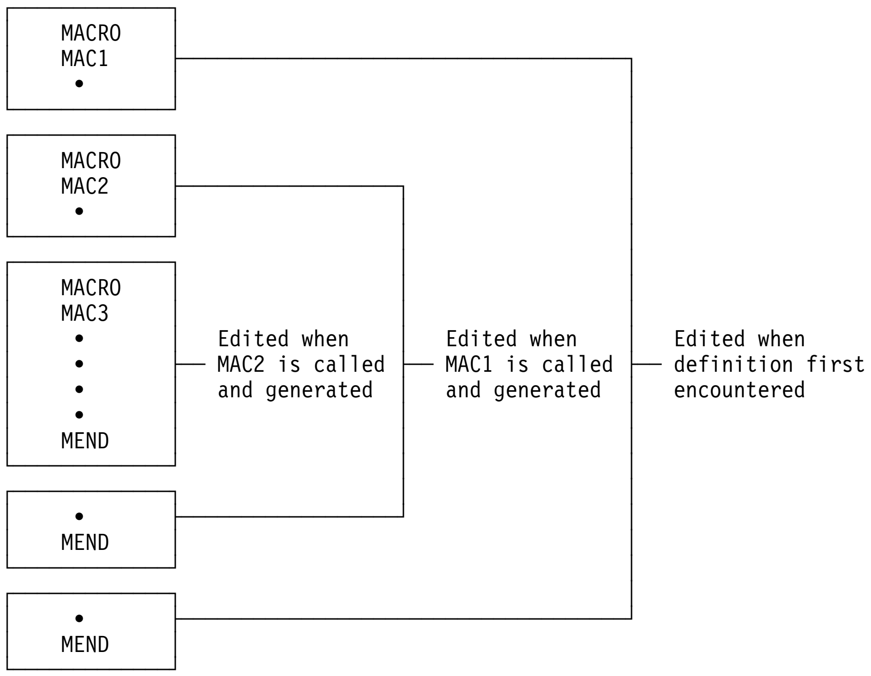
MACRO
MAC1
MACRO
MAC2
MACRO
MAC3
MEND
MEND
MEND
Edited when MAC2 is called and generated
Edited when MAC1 is called and generated
Edited when definition first encountered

Figure 3. Editing Inner Macro Definitions

First MAC1 is edited, and MAC2 and MAC3 are not. When MAC1 is called, MAC2 is edited (unless its definition is bypassed by an AIF or AGO branch); when MAC2 is called, MAC3 is edited. No macro can be called until it has 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.

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. Figure 4 shows the value of selected elements if &P is the first positional parameter and the value assigned to it in a macro instruction is (A,(B,(C)),D).

Figure 4. Multilevel Sublists

Selected Elements from &P	Selected Elements from &SYSLIST	Value of Selected Element
&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

Sublists may also be assigned to SETC symbols and used in macro instruction operands. However, if you specify the COMPAT(SYSLIST) assembler option, the assembler treats sublists in SETC symbols as character strings, not sublists, when used in the operand of macro instructions.

Macro Instruction Name Entries

You can write a name field parameter on the macro prototype statement. You can then assign a value to this parameter from the name entry in the calling macro (instruction). Unlike in earlier assemblers, the name entry need not be a valid symbol.

The name entry of a macro instruction can be used to:

- Pass values into the called macro definition.
- Provide a conditional assembly label (sequence symbol) so that you can branch to the macro instruction during conditional assembly.

DBCS Language Support

Double-byte data is supported by the macro language with the following:

- The addition of a pure DBCS (G-type) self-defining term.
- Double-byte data is permitted in the operands of the MNOTE, PUNCH and TITLE statements.
- The REPRO statement exactly reproduces the record that follows it, whether it contains double-byte data or not.
- Double-byte data can be used in the macro language, wherever quoted EBCDIC character strings can be used.
- When a *shift-in* (SI) code is placed in the end column of a continued line, and a *shift-out* (SO) code is placed in the continue column of the next line, the SI and SO are considered redundant and are removed from the statement before it is analyzed.
- Redundant SI/SO pairs are removed when double-byte data is concatenated with double-byte data.
- An extended continuation indicator provides the ability to:

– Extend the end column to the left on a line-by-line basis, so that any alignment of double-byte data in a source statement can be supported.

– Preserve the readability of a macro-generated statement on a DBCS device by splitting double-byte data across listing lines with correct SO/SI bracketing.

Source Stream Input—AREAD

The AREAD assembler operation permits a macro to read a record directly from the source stream into a SETC variable symbol. 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 5 shows how the instruction is used:

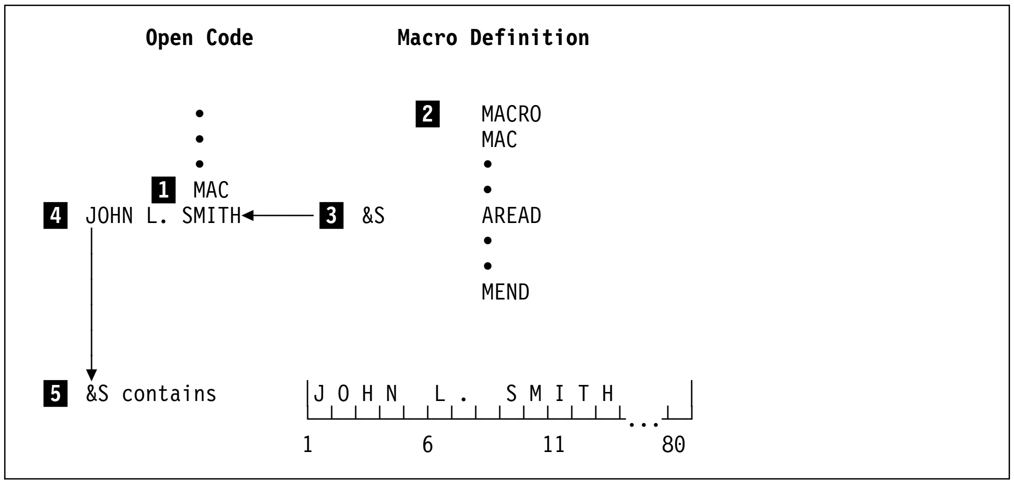

Figure 5. AREAD Assembler Operation

The assembler processes the instructions in Figure 5 as follows:

The macro instruction `MAC` (1) causes the macro `MAC` (2) to be called. When the AREAD instruction (3) is encountered, the next sequential record (4) following the macro instruction is read and assigned to the SETC symbol `&S` (5) .

Repeated AREAD statements read successive records.

When macro instructions are nested, the records read by AREAD must always follow the outermost macro instruction regardless of the level of nesting in which the AREAD instruction is found.

If the macro instruction is found in code brought in by the COPY instruction (copy code), the records read by the AREAD instruction can also be in the copy code. If no more records exist in the copy code, subsequent records are read from the ordinary input stream.

Records that are read in by the AREAD instruction are not checked by the assembler. Therefore, no diagnostic is issued if your AREAD statements read records that are meant to be part of your source program. For example, if an AREAD statement is processed immediately before the END instruction, the END instruction is lost to the assembler.

AREAD Listing Options

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

AREAD Clock Functions

You can specify the CLOCKB or CLOCKD operand in the AREAD instruction to obtain the local time. The time is assigned to the SETC symbol you code in the name field of the AREAD instruction. The CLOCKB operand obtains the time in hundredths of a second. The CLOCKD operand obtains the time in the format *HHMMSSTH*.

Macro Input/Output Capability

The AREAD facility complements the PUNCH facility to provide macros with direct input/output (I/O) capability. The total I/O 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 records
Conditional I/O:	SET symbol values provided by external functions, using the SETAF and SETCF conditional assembly instructions.

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

Source Stream Insertion—AINSERT

The AINSERT assembler operation inserts statements into the input stream. The statements are stored in an internal buffer until the macro generator is completed. Then the internal buffer is used to provide the next statements. An operand controls the sequence of insertion of statements within the buffer. Statements can be inserted at the front or back of the queue, though they are removed only from the front of the queue.

Macro Definition Listing Control—ASPACE and AEJECT

You can use the ASPACE and AEJECT instructions to control the listing of your macro definitions. The ASPACE instruction is similar to the SPACE instruction, but instead of controlling the listing of your open code, you can use it to insert one or more blank lines in your macro definition listing. Similarly, the AEJECT instruction is like the EJECT instruction, but you can use it to stop printing the macro definition on the current page and continue printing on the next page.

Other Macro Language Extensions

High Level Assembler provides the following extensions to some earlier macro languages:

- You can insert blank lines in macro definitions provided they are not continuation lines. See also “Blank Lines” on page 11.
- Macro names, variable symbols (including the ampersand), and sequence symbols (including the period), can be a maximum of 63 alphanumeric characters.
- You can insert comments (both ordinary and internal macro types) between the macro header and the prototype and, for library macros, before the macro header. These comments are not printed with the macro generation.
- You can use a macro definition to redefine any machine or assembler instruction. When you subsequently use the machine or assembler instruction the assembler interprets it as a macro call.
- You can include any instruction, except ICTL and *PROCESS statements, in a macro definition.
- You no longer need to precede the SET symbol name with an ampersand in LCLx and GBLx instructions, except for created SET symbols.
- The SETA and SETB instructions now allow you to use predefined absolute symbols in arithmetic expressions. You can also use predefined absolute symbols in a SETC operand.

Conditional Assembly Language Extensions

Extensions to the conditional assembly language provides you with a flexible and powerful tool to increase your productivity, and simplify your coding needs. These include:

- New instructions that support external function calls
- New built-in functions
- Extensions to existing instructions
- Extensions to SET symbol usage
- New system variable symbols
- New data attributes

External Function Calls

You can use the new SETAF and SETCF instructions to call your own routines to provide values for SET symbols. The routines, which are called external functions, can be written in any programming language that conforms to standard OS linkage conventions. The format of the SETAF and SETCF instructions is the same as a SETx instruction, except that the first operand of SETAF is a character string.

The assembler calls the external function load module, and passes it the address of an external function parameter list. Each differently named external function called in the same assembly is provided with a separate parameter list.

SETAF Instruction: You use the SETAF instruction to pass parameters containing arithmetic values to the external function module. The symbol in the name field of the instruction is assigned the fullword integer value returned by the external function module.

SETCF Instruction: You use the SETCF instruction to pass parameters containing character values to the external function module. The symbol in the name field of the instruction is assigned the character string value returned by the external function module. The length of the returned character string can be from 0 to 255 bytes.

Built-In Functions

The assembler provides you with new *built-in functions* that you can use in SETx instructions to perform logical, arithmetic, and character string operations on SETA, SETB and SETC expressions:

AND	Logical AND on two arithmetic expressions
BYTE	Converts an arithmetic expression to a single character.
DOUBLE	Double any quotes and ampersands in a SETC variable
FIND	Return the offset of the first character in a SETC variable or character string, found in another SETC variable or character string
INDEX	Return the offset of one SETC variable or character string, found in another SETC variable or character string
LOWER	Convert a SETC variable to lowercase
NOT	Logical NOT on an arithmetic expression
OR	Logical OR on two arithmetic expressions
SIGNED	Convert arithmetic expression to a string, representing signed value
SLA	Shift SETA variable left arithmetic
SLL	Shift SETA variable left logical
SRA	Shift SETA variable right arithmetic
SRL	Shift SETA variable right logical
UPPER	Convert a SETC variable to uppercase
XOR	Logical exclusive OR on two arithmetic expressions or on two logical expressions

In the following examples, assume that &B is set to the arithmetic value +10, and &C is set to the arithmetic value +2:

```
Name      Operation      Operand       Comment

&Z        SETA           (&C SLA 2)    Shift Left Arithmetic 2 bits
&Y        SETA           (&C SRA &B)   Shift Right Arithmetic 10 bits
&X        SETA           (&B SLL &C)   Shift Left Logical 2 bits
&W        SETA           (&B AND &C)   Logical AND
&V        SETA           (NOT &B)      Logical NOT
&zeroes   SETA           0
&ones     SETA           (NOT &zeroes) Logical NOT
```

These statements have the following effect:

&Z contains the arithmetic value +8

&Y contains the arithmetic value 0

&X contains the arithmetic value +40

&W contains the arithmetic value +2

&V contains the arithmetic value −11

&ones contains the value −1, or X'FFFFFFFF'. This is an example of how to create a mask of all one bits.

In the following examples assume that &E is set to the character value ‘EIGHT’, &T is set to the character value ‘twentyEIGHT’, and &D is set to the character value ‘&S'TUV’

```
Name      Operation      Operand

&Z        SETC           (UPPER '&T'(1,2))
&Y        SETC           (LOWER '&E')
&X        SETC           (DOUBLE '&D')
&W        SETA           ('&T' INDEX '&EIGHT')
&V        SETA           ('&E' FIND 'GT')
```

These statements have the following effect:

&Z contains the character value 'TW'

&Y contains the character value 'eight'

&X contains the character value '&&S''TUV'

&W contains the value 7

&V contains the value 3 (the G in GT matches the G in EIGHT)

AIF Instruction

The AIF instruction can include a string of logical expressions and related sequence symbols that is equivalent to multiple AIF instructions. This form of the AIF instruction is described as an *extended* AIF instruction. There is no limit to the number of expressions and symbols that you can use in an extended AIF instruction.

AGO Instruction

An AGO instruction lets you make branches according to the value of an arithmetic expression in the operand. This form of the AGO instruction is described as a *computed* AGO instruction.

Extended Continuation Statements

For the following statements, the assembler allows as many continuation statements as are needed:

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

SET Symbols and SETx Statements

The most powerful element of the conditional assembly language is SET symbol support. SET symbols are variable symbols that provide you with arithmetic, binary, or character data, and whose values you can set at conditional assembly time with the SETA, SETB, and SETC instructions, respectively. This section discusses some of the major features of this support, and the extensions High Level Assembler provides.

SET Symbol Definition

When you define a SET symbol, you determine its scope. The scope of the SET symbol is that part of a program for which the SET symbol has been declared. A SET symbol can be defined as having local scope or global scope.

If you declare a SET symbol to have local scope, you can use it only in the statements that are part of:

- The macro definition in which it was defined, or
- Open code, if it was defined in open code

If you declare a SET symbol to have global scope, you can use it in the statements that are part of:

- The same macro definition
- A different macro definition
- Open code

To help you with SET symbol definition, High Level Assembler provides the following facilities:

- A SET symbol is declared implicitly when it appears in the name field of a SETx instruction, and it has not been declared in a LCLx or GBLx instruction. It is assigned as having local scope. If the assembler subsequently encounters any local scope explicit declaration of the symbol, the symbol is flagged as a duplicate declaration. A SET symbol is declared as an array if the name field of the SETx instruction contains a subscript. See “Array Processing with SET Symbols” on page 31.

- Global and local SET symbol declarations are processed at conditional assembly time. Both a macro definition and open code can contain more than one declaration for a given SET symbol, as long as only one is encountered during a given macro generation or conditional assembly of open code.
- A SET symbol can be defined as an array of values by specifying a subscript when you declare it, 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, as shown in the following example:

```
Name      Operation     Operand

          LCLA          &J(50)
&J(45)    SETA          415
&J(89)    SETA          38
```

Created SET Symbols

You can create SET symbols 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 alphanumeric characters
- Predefined symbols with absolute values
- Other 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

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

`&X` is a variable whose value is the name of the variable to be updated.

These statements have an effect identical to:

```
&A3       SETA          5
```

You can use created SET symbols wherever ordinary SET symbols are permitted, including declarations; they can even be nested in other created SET symbols.

The created SET symbol can be thought of as a form of indirect addressing. With nested created SET symbols, you can use 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 `I`-th element of the symbol substituted for `&SYM`. Note that the value of `&SYM` need not be the name of a valid symbol; thus created SET symbols may have arbitrary *names*.

Created SET symbols also allow you to achieve 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)` is 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, the names of parameters are recognized and encoded (fixed) at macro-edit time. If a created SET symbol name happens to coincide with a parameter name, the coincidence is ignored and there is no interaction between the two.

Array Processing with SET 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(exp) SETx           X1,X2,,X4,...,Xn
```

where:

`&SYM`	is a dimensioned SET symbol
`exp`	is a SETA arithmetic expression
`SETx`	is SETA, SETB, or SETC

An operand can be omitted by specifying two commas without intervening blanks. Whenever an operand is omitted, the corresponding element of the dimensioned variable SET symbol (`&SYM`) is left unchanged.

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. This includes the G-type self-defining term. A null value is treated as zero. This use of the SETC variable lets you associate numeric values with EBCDIC, DBCS, 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     SETA           &X
```

Using Ordinary Symbols in SETx Statements

In addition to variable symbols, self-defining terms, and attribute references, predefined symbols that have absolute values can be used in SETA and SETB statements. You can use this facility to do arithmetic or logical operations on expressions whose values are unknown at coding time, or are difficult to calculate. For example, the following code could be used to assign the length of a CSECT to a SETA symbol:

```
Name      Operation      Operand

BEGIN     CSECT
           ⋮
CSECTLEN  EQU           *-BEGIN
&CSCTLEN  SETA          CSECTLEN
```

Similarly, in addition to character expressions and type attribute references, predefined symbols that have absolute values can be used in SETC statements.

For example, the following code could be used to assign a string of fifty spaces to a SETC symbol:

```
Name      Operation     Operand

FIFTY     EQU           50
          ⋮
&SPACES   SETC          (FIFTY)' '
```

Substring Length Value

You can specify an asterisk as the second subscript value of the substring notation. This indicates that the length of the extracted string is equal to the length of the character string, less the number of characters before the starting character.

The following examples show how the substring notation can be used:

```
Name      Operation     Operand                Comment

&Z        SETC          'Astring'(2,3)         length specified
&Y        SETC          'Astring'(2,*)         length not specified
&X        SETC          (UPPER '&Y'(3,*))  length not specified
```

These statements have the following effect:

&Z contains the character value 'str'

&Y contains the character value 'string'

&X contains the character value 'RING'.

See "Built-In Functions" on page 27 for an explanation of the UPPER built-in function.

Attribute References

Data such as instructions, constants, and areas have characteristics called data attributes. The assembler assigns attribute values to the ordinary symbols and variable symbols that represent the data.

You can determine up to eight attributes of symbols you define in your program by means of an attribute reference. By testing attributes in conditional assembly instructions, you can control the conditional assembly logic.

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

Figure 6 shows the data attributes.

Figure 6 (Page 1 of 2). Data Attributes

Attribute	Purpose	Notation
Type	Gives a letter that identifies the type of data represented by an ordinary symbol, a macro instruction operand, a SET symbol, and a literal	T'
Length	Gives the number of bytes occupied by the data that is named by the symbol, or literal, specified in the attribute reference	L'

Figure 6 (Page 2 of 2). Data Attributes

Attribute	Purpose	Notation
Scaling	Refers to the position of the decimal point in decimal, fixed-point, and floating-point constants	`S'`
Integer	Is a function of the length and scaling attributes of decimal, fixed-point, and floating-point constants	`I'`
Count	Gives the number of characters that would be required to represent the current value of the SET symbol or the system variable symbol. It also gives the number of characters that constitute the macro operand instruction.	`K'`
Number	Gives the number of sublist entries in a macro instruction operand sublist	`N'`
Defined	Indicates whether the symbol referenced has been defined prior to the attribute reference	`D'`
Operation Code	Indicates whether a given operation code has been defined prior to the attribute reference	`O'`

Where Attribute References Can Be Used

References to the type (T'), length (L'), scaling (S'), and integer (I') attributes of ordinary symbols and SETC symbols can be used in:

- Conditional assembly instructions
- Any assembler instruction that accepts an absolute expression as an operand
- Any machine instruction

For example:

```
Name      Operation      Operand            Comment

&TYPE     SETC           T'PACKED           Type
LENGTH    LA             2,L'PACKED         Length
ADTYPE    LA             2,T'PACKED         Value of Type (C'P')
&SCALE    SETA           S'PACKED           Scaling
INTEGER   DC             AL1(I'PACKED)      Integer
          :
PACKED    DC             P'123.45'          Referenced Symbol
```

Attribute references to the count (K') and number (N') attributes, however, can only be used in conditional assembly instructions.

Attribute References and SETC Variables

The symbol referenced by an attribute reference of length (L'), type (T'), scaling (S'), integer (I'), and defined (D'), 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

Attribute References and Literals

In addition to symbols, you can reference literals with the type, length, scaling, and integer attribute references. In some cases you can only reference a literal that has been defined prior to the reference. For example:

```
Name      Operation     Operand             Comment

LENGTH    LA            2,L'=C'ABCXYZ'      Length attribute has value 6
TYPE      EQU           T'=F'1000'          Type attribute has value 'F'
          ⋮
INTEGER   DC            AL2(I'=P'123.45')   Integer attribute has value 3
```

Type Attribute of a CNOP label

The type attribute (T') of a CNOP label has been changed to ‘I’. In Assembler H Version 2 the attribute value was ‘J’.

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. See “Forward Attribute-Reference Scan” on page 35.

Operation Code Attribute (O')

The operation code attribute (O') can be used in conditional assembly statements to determine if a given operation code has been defined prior to the attribute reference. The following letters are used for the operation code attribute value:

A	Assembler operation codes
E	Extended mnemonic operation codes
M	Macro operation codes
O	Machine operation codes
S	Macro found in SYSLIB (MVS and CMS) or library (by Librarian on VSE)
U	Undefined operation codes

If an operation code is redefined using the OPSYN instruction the attribute value represents the new operation code type. If the operation code is deleted using the OPSYN instruction the attribute value is ‘U’.

The following example checks to see if the macro MYMAC is defined. If not, the MYMAC macro instruction is bypassed. This example prevents the assembly from failing when the macro is not available.

```
Name      Operation     Operand

&CHECKIT  SETC          O'MYMAC
          AIF           ('&CHECKIT' EQ 'U').NOMAC
          MYMAC
.NOMAC    ANOP
          ⋮
DATAAREA  DC            F'0'
```

Number Attributes for SET Symbols

The number attribute (N') 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 definitions 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 been defined but not assigned any value, regardless of whether it is subscripted or not. The number attribute is always 1 for a SET symbol that is not subscripted and when the SET symbol has been assigned a value.

The number attribute also applies to the operands of macro instructions.

Forward Attribute-Reference Scan

If you make an attribute reference to an undeclared symbol, the assembler scans the source code either until it finds the symbol in the name field of a statement in open code, or until it reaches the end of the source module. The assembler makes an entry in the symbol table for the symbol, as well as for any other previously undefined symbols it encounters during the scan. 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 can result from a forward scan, even though the statement is later found to be in error and therefore not accepted by the assembler.

You must be careful with the contents of any AREAD input in your source module. If an AREAD input record is encountered before the symbol definition, and the record has the same format as an assembler language statement, and the name field contains the symbol referred to in the attribute reference, then the forward scan will attempt to evaluate that record instead.

Redefining Conditional Assembly Instructions

You can use the OPSYN instruction to redefine conditional assembly instructions anywhere in your source module. A redefinition of a conditional assembly instruction affects only macro definitions occurring after the OPSYN instruction. The original definition of a conditional assembly instruction is always used during the processing of subsequent calls to a macro that was defined before the OPSYN instruction.

An OPSYN instruction that redefines the operation code of an assembler or machine instruction generated from a macro instruction is effective immediately, even if the definition of the macro was made prior to the OPSYN instruction. Consider the following example:

```
Name     Operation      Operand        Comment

          MACRO                        Macro header
          MACRDEF        ...            Macro prototype
          AIF            ...
          MVC            ...
          ⋮
          MEND                          Macro trailer
          ⋮
 AIF      OPSYN          AGO            Assign AGO properties to AIF
 MVC      OPSYN          MVI            Assign MVI properties to MVC
          ⋮
          MACRDEF        ...            Macro call
                                        (AIF interpreted as AIF instruct-
                                        ion; generated AIFs not printed)
+         MVC            ...            Interpreted as MVI instruction
          ⋮
                                        Open code started at this point
          AIF            ...            Interpreted as AGO instruction
          MVC            ...            Interpreted as MVI instruction
```

In this example, AIF and MVC instructions are used in a macro definition. AIF is a conditional assembly instruction, and MVC is a machine instruction. OPSYN statements assign the properties of AGO to AIF and assign the properties of MVI to MVC. In subsequent calls of the macro MACRDEF, AIF is still defined, and used as an AIF operation, but the generated 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 used for further generations.

This description does not apply to nested macro definitions because the assembler does not edit inner macro definitions until it encounters them during the generation of its outer macro. An OPSYN statement placed before the outer macro instruction can affect conditional assembly statements in the inner macro definition.

System Variable Symbols

System variable symbols are read-only, local-scope or global-scope variable symbols whose values are determined and assigned only by the assembler. System variable symbols that have local scope are assigned a read-only value each time a macro definition is called by a macro instruction. You can only refer to local-scope system variable symbols inside macro definitions. System variable symbols that have global scope are assigned a read-only value for the whole assembly. You can refer to global-scope system variable symbols in open code and in macro definitions.

The format of the following two system variables has changed since Assembler H Version 2:

- &SYSLIST treats parenthesized sublists in SETC symbols as sublists when passed to a macro definition in the operand of a macro instruction. The COMPAT(SYSLIST) assembler option can be used to treat sublists in the same way as Assembler H Version 2, that is, parenthesized sublists are treated as character strings, not sublists.

- &SYSPARM can now be up to 255 characters long, subject to restrictions imposed by job control language.

Some of the new system variable symbols introduced with High Level Assembler supplement the data provided by system variables available in previous assemblers.

&SYSCLOCK: &SYSCLOCK provides the date and time the macro is generated.

&SYSDATE and &SYSDATC: &SYSDATE provides the date in the format *MM/DD/YY* without the century digits, and the year digits are in the lowest-order positions.

The new variable symbol &SYSDATC provides the date with the century, and the year digits in the highest-order positions. Its format is *YYYYMMDD*.

&SYSECT and &SYSSTYP: All previous assemblers have supported the &SYSECT variable to hold the name of the enclosing control section at the time a macro was invoked. This allows a macro that needs to change control sections to resume the original control section on exit from the macro. However, there was no capability to determine what *type* of control section to resume.

The &SYSSTYP variable provides the type of the control section named by &SYSECT. This permits a macro to restore the correct previous control section environment on exit.

&SYSMAC: Retrieves the name of any macro called between opencode and the current nesting level.

&SYSM_HSEV: Provides the highest MNOTE severity code for the assembly so far.

&SYSM_SEV: Provides the highest MNOTE severity code for the macro most recently called from this macro or open code.

&SYSOPT_XOBJECT: Determines if the XOBJECT assembler option was specified.

&SYSNDX and &SYSNEST: All previous assemblers have supported the &SYSNDX variable symbol, which is incremented by one for every macro invocation in the program. This permits macros to generate unique ordinary symbols if they are needed as local labels. Occasionally, in recursively nested macro calls, the value of the &SYSNDX variable was used to determine either the depth of nesting, or to determine when control had returned to a particular level.

Alternatively, the programmer could define a global variable symbol, and in each macro insert statements to increment that variable on entry and decrement it on exit. This technique is both cumbersome (because it requires extra coding in every macro) and unreliable (because not every macro called in a program is likely to be under the programmer's control).

High Level Assembler provides the &SYSNEST variable to keep track of the level of macro-call nesting in the program. The value of &SYSNEST is incremented globally on each macro entry, and decremented on each exit.

&SYSTIME and the AREAD Statement

The &SYSTIME variable symbol is provided by High Level Assembler and Assembler H, but not by earlier assemblers. It provides the local time of the start of the assembly in *HH.MM* format. This time stamp may not have sufficient accuracy or resolution for some applications.

High Level Assembler provides an extension to the AREAD statement, discussed in more detail in "AREAD Clock Functions" on page 25, that may be useful if a more accurate time stamp is required.

Appendix B, "System Variable Symbols" on page 75 describes all the system variable symbols.

Chapter 5. Using Exits to Complement File Processing

The High Level Assembler EXIT option lets you provide an exit module that can replace or complement the assembler's data set input/output processing. This chapter describes the exits available to you and how to use them.

User Exit Types

You can select up to seven exit types during an assembly on MVS and CMS, or six on VSE:

Exit Type	**Exit Processing**
SOURCE	Use this exit to replace or complement the assembler's primary input file processing. It can read primary input records instead of the assembler, or it can monitor and optionally modify the records read by the assembler before they are processed. You can also use the SOURCE exit to provide additional primary input records.
LIBRARY	Use this exit to replace or complement the assembler's MACRO and COPY library processing. It can read MACRO and COPY library records instead of the assembler, or it can monitor and optionally modify the records read by the assembler before they are processed. You can also use the LIBRARY exit to provide additional MACRO and COPY source records.
LISTING	Use this exit to replace or complement the assembler's listing output processing. It can write the listing records provided by the assembler, or it can monitor and optionally modify the records before they are written by the assembler. You can also use the LISTING exit to provide additional listing records.
OBJECT	(MVS and CMS) Use this exit to replace or complement the assembler's object module output processing. It can write object module records provided by the assembler, or monitor and optionally modify the records before they are written by the assembler. You can also use the OBJECT exit to provide additional object module records. The OBJECT exit is the same as the PUNCH exit, except that you use it when you specify the OBJECT assembler option to write object records to SYSLIN.
PUNCH	On MVS and CMS, the PUNCH exit is the same as the OBJECT exit, except that you use it when you specify the DECK assembler option to write object records to SYSPUNCH. On VSE, use this exit to replace or complement the assembler's object module output processing. It can write object module records provided by the assembler, or monitor and optionally modify the records before they are written by the assembler. You can also use the PUNCH exit to provide additional object module records.
ADATA	Use this exit to monitor the assembler's associated data output processing. The ADATA exit cannot modify the records, discard records, or provide additional records.

TERM Use this exit to replace or complement the assembler's terminal output processing. It can write the terminal records provided by the assembler, or it can monitor and optionally modify the records before they are written by the assembler. You can also use the TERM exit to provide additional terminal output records.

How to Supply a User Exit to the Assembler

You must supply a user exit as a module that is available in the standard module search order.

You may write an exit in any language that allows it to be loaded once and called many times at the module entry point, and conforms to standard OS Linkage conventions.

On entry to the exit module, Register 1 points to an Exit Parameter list supplied by the assembler. The Exit Parameter list has a pointer to an Exit-Specific Information block that contains specific information for each exit type. High Level Assembler provides you with a macro, called ASMAXITP, which lets you map the Exit Parameter list and the Exit Specific Information block.

You specify the name of the exit module in the EXIT assembler option. You can also pass up to 64 characters of data to the exit, by supplying them as a suboption of the EXIT option. The assembler passes the data to your exit during assembler initialization.

Passing Data to I/O Exits from the Assembler Source

You can use the EXITCTL instruction to pass data from the assembler source to any of the exits. The assembler maintains four signed, fullword, exit-control parameters for each type of exit. You use the EXITCTL instruction to set or modify the contents of the four fullwords during the assembly, by specifying the following values in the operand fields:

- A decimal self-defining term with a value in the range -2^{31} to $+2^{31}-1$.
- An expression in the form `*±`*n*, where `*` is the current value of the corresponding exit-control parameter to which *n*, a decimal self-defining term, is added or from which *n* is subtracted. The value of the result of adding *n* to or subtracting *n* from the current exit-control parameter value must be in the range -2^{31} to $+2^{31}-1$.

If a value is omitted, the corresponding exit-control parameter retains its current value.

The assembler initializes all exit-control parameters to binary zeros.

Statistics

The assembler writes the exit usage statistics to the *Diagnostic Cross Reference and Assembler Summary* section of the assembler listing.

Disabling an Exit

A return code of 16 allows an EXIT to disable itself. The EXIT is not called again during this assembly, or any following assemblies if the BATCH option is being used.

Communication between Exits

The Common User field in the Request information block provides a mechanism by which all exits can communicate and share information.

Reading Edited Macros (VSE only)

An E-Deck refers to a macro source book of type E that can be used as the name of a macro definition to process in a macro instruction. E-Decks are stored in edited format, however High Level Assembler requires library macros to be stored in source statement format. You can use the LIBRARY exit to analyze a macro definition, and, in the case of an E-Deck, call the VSE/ESA ESERV program to change, line by line, the E-Deck definition back into source statement format.

See the section titled *Using the High Level Assembler Library Exit for Processing E-Decks* in the *IBM VSE/ESA Guide to System Functions* manual. This section describes how to set up a LIBRARY exit and use it to process E-Decks.

Sample Exits provided with High Level Assembler (MVS and CMS)

The following sample exits are provided with High Level Assembler:

ADATA Exit: The ADATA exit handles the details of interfaces to the assembler, and provides ADATA records to any of a number of *filter* routines that inspect the records to extract the information they require. This lets you add or modify a filter routine without impacting either the exit or the other filter routines.

The design of the exit:

- Supports multiple simultaneous filter routines.
- Simplifies the ADATA-record interface for each filter, because you do not need to be concerned about the complex details of interacting directly with the assembler.
- Supports filter routines written in high-level languages.

There are three components that make up the functional ADATA exit:

1. The exit routine, ASMAXADT, which the assembler invokes.
2. A table of filter-routine names, contained in a *Filter Management Table* (FMT), module ASMAXFMT. The exit routine loads the FMT.
3. The filter routines. The exit loads these as directed by the FMT.

No filter routines are provided with High Level Assembler. Appendix I, “Sample ADATA User Exit” in the *High Level Assembler Programmer's Guide, SC26-4941*, describes the exit and the input format of the filter routines.

LISTING Exit: Use the LISTING exit to suppress the *High Level Assembler Options Summary* section, or the *Diagnostic Cross Reference and Assembler Summary* section, or both from the assembler listing. The exit can also direct the assembler to print the options summary at the end of the assembler listing. You specify keywords as suboptions of the EXIT option to control how the assembler processes these sections of the listing.

The LISTING exit is called ASMAXPRT.

Appendix J, "Sample LISTING User Exit" in the *High Level Assembler Programmer's Guide, SC26-4941* describes the exit and the keywords you can use to select the print options.

SOURCE Exit: Use the SOURCE exit to read variable-length source data sets. Each record that is read is passed to the assembler as an 80-byte source statement. If any record in the input data set is longer than 71 characters the remaining part of the record is converted into continuation records.

The exit also reads a data set with a fixed record length of 80 bytes.

Appendix K, "Sample SOURCE User Exit" in the *High Level Assembler Programmer's Guide, SC26-4941*, describes this exit.

Chapter 6. Programming and Diagnostic Aids

High Level Assembler has many assembler listing and diagnostic features to aid program development and to simplify the location and analysis of program errors. You can also produce terminal output to assist in diagnosing assembly errors. This chapter describes these features.

Assembler Listings

High Level Assembler produces a comprehensive assembler listing that provides information about a program and its assembly. Each section of the assembler listing is clear and easily readable. The following assembler options are used to control the format and which sections of the listing to produce:

ASA (MVS and CMS) Allows you to use American National Standard printer control characters, instead of machine printer control characters.

DXREF Produces the *DSECT Cross Reference* section.

ESD Produces the *External Symbol Dictionary* section.

EXIT(PRTEXIT(*mod3***))**
Supplies a listing exit to replace or complement the assembler's listing output processing.

FOLD Instructs the assembler to print the assembler listing in uppercase characters, except for quoted strings and comments.

LANGUAGE
Produces error diagnostic messages in the following languages:

- English mixed case (EN)
- English uppercase (UE)
- German (DE)
- Japanese (JP)
- Spanish (ES)

When you select either of the English languages, the assembler listing headings are produced in the same case as the diagnostic messages.

When you select either the German language or the Spanish language, the assembler listing headings are produced in mixed case English.

When you select the Japanese language, the assembler listing headings are produced in uppercase English.

The assembler uses the installation default language for messages produced in CMS by the High Level Assembler command.

LINECOUNT
Specifies how many lines should be printed on each page, including the title and heading lines.

LIST Controls the format of the *Source and Object* section of the listing. NOLIST suppresses the entire listing.

MXREF Produces one, or both, of the *Macro and Copy Code Source Summary* and *Macro and Copy Code Cross Reference* sections.

© Copyright IBM Corp. 1981, 1998

PCONTROL
Controls what statements are printed in the listing, and overrides some PRINT instructions.

RLD Produces the *Relocation Dictionary* section.

RXREF Produces the *General Purpose Register Cross Reference* section.

USING(MAP)
Produces the *Using Map* section.

XREF Produces one, or both, of the *Ordinary Symbol and Literal Cross Reference* and the *Unreferenced Symbols Defined in CSECTs* sections.

Option Summary

High Level Assembler provides a summary of the options current for the assembly, including:

- A list of the overriding parameters specified when the assembler was called
- The options specified on *PROCESS statements
- In-line error diagnostic messages for any overriding parameters and *PROCESS statements in error

You cannot suppress the option summary unless you suppress the entire listing, or you supply a user exit to control which lines are printed.

On MVS and CMS, High Level Assembler provides a sample LISTING exit that allows you to suppress the option summary or print it at the end of the listing. See the description of the sample listing exit on page 42.

Figure 7 shows an example of the *High Level Assembler Option Summary*. The example includes assembler options that have been specified in the invocation parameters and in *PROCESS statements. It also shows the *PROCESS statements in the *Source and Object* section of the listing.

```
                                          High Level Assembler Option Summary                                   Page    1
                                                                                                 1          2
                                                                                          HLASM R3.0  1998/09/25  11.38
  Overriding Parameters-  NOOBJECT,language(en),size(4meg),xref(short,unrefs)
  Process Statements-     ALIGN
                          noDBCS
                          MXREF(FULL),noLIBMAC
                          FLAG(0)
                          noFOLD,LANGUAGE(ue)
                          NORA2
                          NODBCS
                          XREF(FULL)

    3
** ASMA400W Error in invocation parameter - size(4meg)
** ASMA422N Option LANGUAGE is not valid on a *PROCESS statement.
** ASMA437N Attempt to override invocation parameter in *PROCESS statement. Suboption FULL of XREF option ignored.

  Options for this Assembly
   NOADATA
     ALIGN
   NOASA
     BATCH
   NOCOMPAT
   NODBCS
   NODECK
     DXREF
     ESD
   NOEXIT
     FLAG(0,ALIGN,CONT,NOIMPLEN,PAGE0,RECORD,NOSUBSTR)
   NOFOLD
   NOINFO
     LANGUAGE(EN)
   NOLIBMAC
     LINECOUNT(60)
     LIST(121)
     MXREF(FULL)
   NOOBJECT
     OPTABLE(UNI)
   NOPCONTROL
   NOPESTOP
   NOPROFILE
   NORA2
   NORENT
     RLD
     SIZE(MAX)
     SYSPARM()
   NOTERM
   NOTEST
   NOTRANSLATE
     USING(NOLIMIT,MAP,WARN(15))
   NOXOBJECT
     XREF(SHORT,UNREFS)

   4
  No Overriding DD Names
                                                  :
                                                                                                 Page    3
  Active Usings: None
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                            HLASM R3.0  1998/09/25  11.38
                                              5
                                      1 *PROCESS ALIGN                                                    00001000
                                      2 *process noDBCS          any text here is a comment               00002000
                                      3 *process MXREF(FULL),noLIBMAC                                     00003000
                                      4 *PROCESS FLAG(0)                                                  00004000
                                      5 *process noFOLD,LANGUAGE(ue)                                      00005000
                                      6 *PROCESS NORA2                                                    00006000
                                      7 *PROCESS NODBCS                                                   00007000
                                      8 *PROCESS XREF(FULL)                                               00008000
000000                00000 00000     9 A        CSECT                                                    00009000
                R:F  00000           10          USING *,15                                               00010000
```

*Figure 7. Option Summary with options from *PROCESS statements*

The highlighted numbers in the example are:

1 Shows the product description at the top of each page of the assembler listing. (You can use the TITLE instruction to generate individual headings for each page of the source and object program listing.)

2 Shows the date and the time of the assembly.

3 Error diagnostic messages for overriding parameters and *PROCESS statements are shown immediately following the list of *PROCESS statement options.

4 On MVS and CMS, if the assembler has been called by a program and any standard (default) ddnames have been overridden, both the default ddnames and the overriding ddnames are listed. Otherwise, this statement appears:

```
No Overriding DD Names
```

5 The *PROCESS statements are written as comment statements in the Source and Object section of the listing.

External Symbol Dictionary

Figure 8 shows the external symbol dictionary (ESD) information passed to the linkage editor or loader, or DFSMS/MVS Binder linkage editor in the object module.

```
SAMP01                                        External Symbol Dictionary                                        Page    2
1         2                        3                     4
Symbol   Type   Id      Address  Length     LD ID  Flags Alias-of                            HLASM R3.0  1998/09/25  11.38
SAMP01    SD 00000001
B_PRV     ED 00000002                    00000001
B_TEXT    ED 00000003 00000000 000000E4  00000001   00
SAMP01    LD 00000004 00000000           00000003   00
ENTRY1    LD 00000005 00000000           00000003   00
KL_INST   SD 00000006
B_PRV     ED 00000007                    00000006
B_TEXT    ED 00000008 00000000 00000000  00000006   00
KL_INST   CM 00000009 00000000           00000008   00
          SD 0000000A
B_PRV     ED 0000000B                    0000000A
B_TEXT    ED 0000000C 000000E8 00000000  0000000A   00
Date0001  ER 0000000D                    0000000A       RCNVDTE
RCNVTME   ER 0000000E                    0000000A
```

Figure 8. External Symbol Dictionary

1 Shows the name of every external dummy section, control section, entry point, external symbol, and class.

2 Indicates whether the symbol is the name of a label definition, external reference, unnamed control section definition, common control section definition, external dummy section, weak external reference, or external definition.

3 Shows the length of the control section.

4 When you define a symbol in an ALIAS instruction, this field shows the external symbol name of which the symbol is an alias.

You can suppress this section of the listing by specifying the NOESD assembler option.

Source and Object

On MVS and CMS, the assembler can produce two formats of the source and object section: a 121-character format and a 133-character format. To select one, you must specify either the LIST(121) assembler option or the LIST(133) assembler option. Both sections show the source statements of the module, and the object code of the assembled statements.

The 133-character format shows the location counter, and the first and second operand addresses (ADDR1 and ADDR2) as 8-byte fields in support of 31-bit addresses. This format is required when producing the extended object file; see “Extended Object Format Modules (MVS and CMS)” on page 12. The 133-character format also contains the first eight characters of the macro name in the identification-sequence field for statements generated by macros.

Figure 9 on page 47 shows an example of the *Source and Object* section of the listing. This section shows the source statements of the module, and the object code of the assembled statements.

The fixed heading line printed on each page of the source and object section of the assembler listing indicates if the control section, at the time of the page eject, is a COM section, a DSECT or an RSECT.

High Level Assembler lets you write your program and print the assembler listing headings in mixed-case.

121-Character Listing Format

Figure 9 shows an example of the source and object section in 121-character format, and in mixed-case.

```
SAMP01   Sample Listing Description                                                                           Page    3
  Active Usings: None
   1       2                          3          4
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R3.0  1998/09/25  11.38
000000              00000 000E0     2 Samp01   Csect                                                          00002000
                                    3          Sav   (14,12)                Save caller's registers           00003000
5 ** ASMA057E Undefined operation code - SAV
6 ** ASMA435I Record 3 in FIG6     ASSEMBLE A1 on volume: ADISK
                                               ⋮
                                   23 Entry1   SAMPMAC  Parm1=YES                                             00023000
000000 18CF                        24+Entry1   LR 12,15                                                       01-SAMPM
                   7
                R:C  00000         25+         USING Entry1,12              Ordinary Using                    01-SAMPM
000002 0000 0000          00000    26+         LA Savearea,10                                                 01-SAMPM
** ASMA044E Undefined symbol - Savearea
** ASMA029E Incorrect register specification
8 ** ASMA435I Record 5 in TEST     MACLIB   A1(SAMPMAC) on volume: ADISK
000006 50D0 A004          00004    27+         ST 13,4(,10)                                                   01-SAMPM
00000A 50A0 D008          00008    28+         ST 10,8(,13)                                                   01-SAMPM
00000E 18DA                        29+         LR 13,10                                                       01-SAMPM
              R:A35  00010         30+         USING *,10,3,5               Ordinary Using,Multiple Base      01-SAMPM
                                                                                                                 9
** ASMA303W Multiple address resolutions may result from this USING and the USING on statement number 25
** ASMA435I Record 9 in TEST     MACLIB   A1(SAMPMAC) on volume: ADISK
                                               ⋮
                                   42+         DROP 10,3,5                  Drop Multiple Registers           01-SAMPM
                                   43          COPY  SAMPLE                                                   00024000
                                   44=*        Line from member SAMPLE                                        00001000
              C 02A  00000 0002A   45          Using IHADCB,INDCB           Establish DCB addressability      00025000
              C 07A  00000 0007A   46 ODCB     Using IHADCB,OUTDCB                                            00026000
                                   47          push  using                                                    00027000
                             10
                R:2  00000         48 PlistIn  Using Plist,2                Establish Plist addressability    00028000
                R:3  00000         49 PlistOut Using Plist,3                                                  00029000
SAMP01   Sample Listing Description                                                                           Page    4
11  Active Usings (1):Entry1(X'1000'),R12  IHADCB(X'FD6'),R12+X'2A'  PlistIn.Plist(X'1000'),R2
  PlistOut.Plist(X'1000'),R3  ODCB.IHADCB(X'F86'),R12+X'7A'
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R3.0  1998/08/04 19.16
000010 1851                        50 ?Branch  LR    R5,R1                  Save Plist pointer                00030000
** ASMA147E Symbol too long, or first character not a letter - ?Branch
** ASMA435I Record 30 in FIG6     ASSEMBLE A1 on volume: ADISK
000012 5825 0000          00000    51          L     R2,0(R5)               R2 = address of request list      00031000
000016 47F0 C022          00022    52          B     Open                                                     00032000
                                               ⋮
                                  697          End                                                            00050000
0000D0 00000001                   698                =f'1'
0000D4 00000000                   699                =v(Rcnvdte)
0000D8 00000000                   700                =v(Rcnvtme)
0000DC 00000002                   701                =f'2'
```

Figure 9. Source and Object listing section—121 format

1 Shows, in hexadecimal notation, the assembled address of the object code.

2 Shows, in hexadecimal notation, the object code generated by assembly of the statement. The object code of machine instructions is printed in full. Only 8 bytes of object code are printed for assembled constants, unless the PRINT DATA instruction or the PCONTROL(DATA) assembler option is specified, in which case all the object code is printed.

3 Shows the statement number. If you specify the PCONTROL(GEN) assembler option, or if you specify the PRINT GEN instruction before a macro instruction, the statements generated by the macro instruction is printed. A plus sign (+) suffixes the statement numbers of generated statements.

4 Shows the source statement.

5 Displays the error diagnostic messages immediately following the source statement in error. Many error diagnostic messages include the segment of the statement that is in error. You can use the FLAG assembler option to control the level of diagnostic messages displayed in your listing.

6 Displays the informational message, ASMA435I, that describes the origin of the source statement in error. This message is only printed when you specify the FLAG(RECORD) assembler option.

7 The *Addr1* and *Addr2* columns show the first and second operand addresses in the USING instructions. The base registers on an ordinary USING instruction are printed, right justified in the *Object Code* columns, preceded by the characters `R:`.

8 Displays the informational message, ASMA435I, that describes the origin of the source statement in error. Conditional assembly statements and comment statements contribute to the record count of macro definitions.

9 The macro name in the identification-sequence field is truncated after the first five characters.

10 The *Addr1* and *Addr2* columns show the first and second operand addresses in the USING instructions. The register and resolved base displacement for a dependent USING instruction are printed in the *Object Code* columns, as `register displacement`. The base address is shown in the *Addr1* column, and the explicit base displacement is shown in the *Addr2* column.

11 Shows active USINGs.

| In this example, the first is an ordinary USING, the second a dependent
| USING, the third a labeled dependent USING, and the last two are labeled
| USINGS.

133-Character Listing Format

| Figure 10 shows an example of the *Source and Object* section when the same
| assembly is run with assembler option LIST(133), and is followed by a description
| of its differences with Figure 9 on page 47:

```
SAMP01   Sample Listing Description                                                                             Page    3
  Active Usings: None
  1
  Loc    Object Code      Addr1    Addr2    Stmt  Source Statement                          HLASM R3.0  1998/09/25  11.38
00000000                00000000 000000E0      2 Samp01   Csect                                                    00002000
                                               3          Sav   (14,12)                Save caller's registers     00003000
** ASMA057E Undefined operation code - SAV
** ASMA435I Record 3 in FIG8     ASSEMBLE A1 on volume: ADISK
                                                          :
                                              23 Entry1   SAMPMAC  Parm1=YES                                       00023000
00000000 18CF                                 24+Entry1   LR 12,15                                                 01-SAMPMAC
                               2
                    R:C 00000000              25+         USING Entry1,12              Ordinary Using              01-SAMPMAC
00000002 0000 0000               00000000     26+         LA Savearea,10                                           01-SAMPMAC
** ASMA044E Undefined symbol - Savearea
** ASMA029E Incorrect register specification
** ASMA435I Record 5 in TEST     MACLIB   A1(SAMPMAC) on volume: ADISK
00000006 50D0 A004               00000004     27+         ST 13,4(,10)                                             01-SAMPMAC
0000000A 50A0 D008               00000008     28+         ST 10,8(,13)                                             01-SAMPMAC
0000000E 18DA                                 29+         LR 13,10                                                 01-SAMPMAC
                                                                                                                 3
                  R:A35 00000010              30+         USING *,10,3,5               Ordinary Using,Multiple Base    01-SAMPMAC
** ASMA303W Multiple address resolutions may result from this USING and the USING on statement number 25
** ASMA435I Record 9 in TEST     MACLIB   A1(SAMPMAC) on volume: ADISK
                                                          :
                                              42+         DROP 10,3,5                  Drop Multiple Registers     01-SAMPMAC
                                              43          COPY  SAMPLE                                             00024000
                                              44=*        Line from member SAMPLE                                  00001000
                  C 02A 00000000 0000002A     45          Using IHADCB,INDCB           Establish DCB addressability    00025000
                  C 07A 00000000 0000007A     46 ODCB     Using IHADCB,OUTDCB                                      00026000
                                              47          push  using                                              00027000
                    R:2 00000000              48 PlistIn  Using Plist,2                Establish Plist addressability  00028000
                    R:3 00000000              49 PlistOut Using Plist,3                                            00029000
SAMP01   Sample Listing Description                                                                             Page    4
  Active Usings (1):Entry1(X'1000'),R12  IHADCB(X'FD6'),R12+X'2A'  PlistIn.Plist(X'1000'),R2
  PlistOut.Plist(X'1000'),R3  ODCB.IHADCB(X'F86'),R12+X'7A'
  Loc    Object Code      Addr1    Addr2    Stmt  Source Statement                          HLASM R3.0  1998/08/04 17.23
00000010 1851                                 50 ?Branch  LR    R5,R1                  Save Plist pointer          00030000
** ASMA147E Symbol too long, or first character not a letter - ?Branch
** ASMA435I Record 30 in FIG8     ASSEMBLE A1 on volume: ADISK
00000012 5825 0000               00000000     51          L     R2,0(R5)               R2 = address of request list    00031000
00000016 47F0 C022               00000022     52          B     Open                                               00032000
                                                          :
                                             697          End                                                      00050000
000000D0 00000001                            698                =f'1'
000000D4 00000000                            699                =v(Rcnvdte)
000000D8 00000000                            700                =v(Rcnvtme)
000000DC 00000002                            701                =f'2'
```

Figure 10. Source and Object listing section—133 format

1 The assembled address of the object code occupies 8 characters.

2 The Addr1 and Addr2 columns show 8-character operand addresses.

3 The first 8 characters of the macro name are shown in the identification-sequence field.

Relocation Dictionary

Figure 11 shows an example of the *Relocation Dictionary* section of the listing, which contains information passed to the linkage editor, or DFSMS/MVS Binder, in the object module. The entries describe the address constants in the assembled program that are affected by relocation.

```
SAMP01                                           Relocation Dictionary                                          Page   17
     1         2         3        4
   Pos.Id   Rel.Id   Flags    Address                                                   HLASM R3.0  1998/09/25  11.38
  00000001 00000001    0C    000000E0
  00000001 00000004    1C    000000DC
  00000001 00000005    1C    000000E4
  00000001 00000006    1C    000000E8
  00000001 00000007    1C    000000EC
```

Figure 11. Relocation Dictionary

1 Indicates the ESD ID assigned to the ESD entry for the control section in which the address constant is defined.

2 Indicates the ESD ID assigned to the ESD entry for the control section to which this address constant refers.

3 Indicates the type of address constant.

4 Shows the assembled address of the address constant.

You can suppress this section of the listing by specifying the NORLD assembler option.

Ordinary Symbol and Literal Cross Reference

Figure 12 shows an example of the *Ordinary Symbol and Literal Cross Reference* section of the listing. It shows a list of symbols and literals defined in your program. This is a useful tool for checking the logic of your program. It helps you see if your data references and branches are correct.

```
                             Ordinary Symbol and Literal Cross Reference                               Page   20
1         2         3        4    5 6     7    8
Symbol   Length   Value     Id    R Type   Defn References                                  HLASM R3.0  1998/09/25  11.38
ASMAXINV      1 00000000 00000001     J     152  170U  185U  190U
AXPABSREC
              4 00000404 FFFFFFFC     F     781  473
AXPCEND       2 00000408 FFFFFFFC     H     782  789
AXPDSN      255 00000200 FFFFFFFC     C     776  267M
AXPERRL       4 0000002C FFFFFFFD     F     761  279M  508M
                    ⋮
fl12nd        1 00000080 FFFFFFFF A   U     622  414   416
FullStatement
             80 00000000 FFFFFFFA     C    1371  343M  344   344M  344   461M  464M
IHADCB        1 00000000 FFFFFFFB     J     799  189U  203U  249M  879   928   999  1124  1131  1147  1154  1167  1265
                                                 1271  1298  1317  1318  1324  1365  1366
IOError       2 00000490 00000001     H     489  252   569
jfcb        176 00000058 FFFFFFFF     X     599  256   600
jix           1 00000000 00000000 C   U     214  215
                    ⋮
WA                       00000001 A   U     202  203U  206D  206   207
WORKAREA      1 00000000 FFFFFFFF     J     595  202U  213U  638
```

Figure 12. Ordinary Symbol and Literal Cross Reference

1 Each symbol or literal. Symbols are shown in the form in which they are defined, either in the name entry of a machine or assembler instruction, or in the operand of an EXTRN or WXTRN instruction. Symbols defined using mixed-case letters are shown in mixed-case letters, unless the FOLD assembler option was specified.

2 The byte length of the field represented by the symbol, in decimal notation.

3 Shows the hexadecimal address that the symbol or literal represents, or the hexadecimal value to which the symbol is equated.

4 Shows the ESD ID assigned to the ESD entry for the control section in which the symbol or literal is defined.

5 Symbols `fl12nd` and `WA` are absolute symbols and are flagged "A" in the R column. Symbol `jix` is the result of a complex relocatable expression and is flagged "C" in the R column. Symbol `IOerror` is simply relocatable and is not flagged. (Column title R is an abbreviation for "Relocatability Type".)

6 Indicates the type attribute of the symbol or literal.

7 Indicates the number of the statement in which the symbol or literal was defined.

8 Shows the statement numbers of the statements in which the symbol or literal appears as an operand. Additional indicators are suffixed to statement numbers as follows:

B The statement contains a branch instruction, and the symbol is used as the branch-target operand.

D The statement contains a DROP instruction, and the symbol is used in the instruction operand.

M The statement caused the field named by the symbol to be modified.

U The statement contains a USING instruction, and the symbol is used in one of the instruction operands.

X The statement contains an EX machine instruction and the symbol, in the second operand, is the symbolic address of the target instruction.

You can suppress this section of the listing by specifying the NOXREF assembler option. You can also suppress all symbols not referenced in the assembly by specifying the XREF(SHORT) assembler option.

Unreferenced Symbols Defined in CSECTs

Figure 13 shows an example of the *Unreferenced Symbols Defined in CSECTs* section of the listing. This section contains a list of symbols defined in CSECTs in your program that are not referenced. It helps you remove unnecessary labels and data definitions, and reduce the size of your program. Use the XREF(UNREFS) assembler option to produce this section.

```
SAMP01                           Unreferenced Symbols Defined in CSECTs                                  Page   19
  1       2
  Defn  Symbol                                                                         HLASM R3.0  1998/09/25  11.38
    47  ODCB
    49  PlistIn
    50  PlistOut
     7  R0
    10  R3
    16  Unreferenced_Long_Symbol
```

Figure 13. Unreferenced Symbols Defined in CSECTS

1 Shows the statement number that defines the symbol.

2 Shows the symbol name.

General Purpose Register Cross Reference

Figure 14 shows an example of the *General Purpose Register Cross Reference* section of the listing. It lists the registers, and the lines where they are referenced. This helps find all references to registers, particularly those generated by macros that do not use symbolic names, or references using symbolic names than the common R0, R1, and so on.

```
                                        General Purpose Register Cross Reference
Register  References (M=modified, B=branch, U=USING, D=DROP, N=index)
    1       2
    0(0)  115
    1(1)  118  120  121  122  124  126  127  128  130  131  133  135  136  137
    2(2)   36   37   38   39   40   41   42   43   44M  45   46   47   48   49   50   51
           52M  53   54   55M  56   57   58   59M  60   61   62   63   64   65   66   67
           68   69   70   71   72M  73   74   75   76   77   78   79   80   81   82   83
           84   85   86   87   88   89M  90   91   92   93M  94   95   96   97   98   99
          100  101  102  103  104  105  106  107  108  109  110  111  112
    3(3)  (no references identified)    3
    4(4)   16M 281
    5(5)  283
    6(6)   66N 167N 170  171  174  178  180N 190  192  193  194  197  199  200  201N
    7(7)  283
    8(8)  283
    9(9)  224  225  226  227
   10(A)  255U 342D
   11(B)  237  238  239N 240  241  242  243N 244  245N 271
   12(C)    8U
   13(D)  261  262  263  264  265  266
   14(E)  209  210  211  212  213  214  215  216
   15(F)   34  144
```

Figure 14. General Purpose Register Cross Reference

1 Lists the sixteen general registers (0–15).

2 The statements within the program that reference the register. Additional indicators are suffixed to the statement numbers as follows:

(blank) Referenced

M Modified

B Used as a branch address

U Used in USING statement

D Used in DROP statement

N Used as an index register

3 The assembler indicates when it has not detected any references to a register.

You can produce this section of the listing by specifying the RXREF

Note: The implicit use of a register to resolve a symbol to a base and displacement does not create a reference in the General Purpose Register Cross Reference.

Macro and Copy Code Source Summary

Figure 15 shows an example of the *Macro and Copy Code Source Summary* section of the listing. This section shows where the assembler read each macro or copy code member from. It helps you ensure you have included the correct version of a macro or copy code member. Either the MXREF(SOURCE), or MXREF(FULL) assembler option generates this section of the listing.

```
SAMP01                                    Macro and Copy Code Source Summary                                     Page   27
    1     2                                             3         4
 Con Source                                        Volume    Members                        HLASM R3.0  1998/09/25  11.38
     PRIMARY INPUT                                           A        AINSERT_TEST_MACRO         AL       L        MAC1
                                                             N        O        SL       ST       TYPCHKRX X
  L1 TEST     MACLIB   A1                          ADISK     SAMPLE   SAMPMAC  XIT1     XIT3
  L2 DSECT    MACLIB   A1                          ADISK     XIT2
  L3 OSMACRO  MACLIB   S2                          MNT190    DCBD     IHBERMAC SAVE
```

Figure 15. Macro and Copy Code Source Summary

1 Shows the concatenation value representing the source of the macros and copy code members. This number is not shown for PRIMARY INPUT. The number is prefixed with L which indicates Library. The concatenation value is cross referenced in the *Macro and Copy Code Cross Reference* section, and the *Diagnostic Cross Reference and Assembler Summary* section.

2 Shows the name of each library from which the assembler read a macro or a copy code member. The term PRIMARY INPUT is used for in-line macros.

3 Shows the volume serial number of the volume on which the library resides.

4 Shows the names of the macros or copy members.

You can suppress this section of the listing by specifying the NOMXREF assembler option, or by specifying the MXREF(XREF) assembler option.

Macro and Copy Code Cross Reference

Figure 16 shows an example of the *Macro and Copy Code Cross Reference* section of the listing. This section lists the names of macros and copy code members used in the program, and the statement numbers where each was called. Either the MXREF(XREF), or MXREF(FULL) assembler option generates this section of the listing.

```
SAMP01                                     Macro and Copy Code Cross Reference                          Page   28
1          2   3               4     5
Macro      Con  Called By        Defn  References                                    HLASM R3.0  1998/09/25  11.38
A               PRIMARY INPUT     826  971, 973, 998
AINSERT_TEST_MACRO
                PRIMARY INPUT       3  16
AL              PRIMARY INPUT     873  981, 983
DCBD        L3  PRIMARY INPUT       -  113
IHBERMAC    L3  DCBD                -  113
L               PRIMARY INPUT     816  966, 968
MAC1            PRIMARY INPUT      28  36
N               PRIMARY INPUT     933  991
O               PRIMARY INPUT     953  993
SAMPLE      L1  PRIMARY INPUT       -  85C   6
SAMPMAC     L1  PRIMARY INPUT       -  64
SAVE        L3  PRIMARY INPUT       -  42
SL              PRIMARY INPUT     883  986, 988
ST              PRIMARY INPUT     836  976, 978
TYPCHKRX        PRIMARY INPUT     745  775, 845, 892
X               PRIMARY INPUT     943  996
XIT1        L1  PRIMARY INPUT       -  30C
XIT2        L2  PRIMARY INPUT       -  32C
XIT3        L1  PRIMARY INPUT       -  34C
```

Figure 16. Macro and Copy Code Cross Reference

1 Shows the macro or copy code member name.

2 Shows the concatenation value representing the source of the macro or copy code member. This value is cross referenced in the *Macro and Copy Code Source Summary* section, and under *Datasets Allocated for this Assembly* in the *Diagnostic Cross Reference and Assembler Summary* section.

3 Shows the name of the macro that calls this macro or copy code member, or `PRIMARY INPUT`, meaning that the macro or copy code member was called directly from the primary input source.

4 Shows one of the following:

- The statement number for macros defined in the primary input file
- A dash (–) for macros or copy code members read from a library.

5 Shows the statement number that contains the macro call or COPY instruction.

6 Shows the statement reference number with a suffix of C, which indicates that the member is specified on a COPY instruction.

Figure 17 shows an example of the *Macro and Copy Code Cross Reference* section when you specify the LIBMAC assembler option.

```
SAMP01                                    Macro and Copy Code Cross Reference                              Page   81
Macro     Con  Called By       Defn  References                                          HLASM R3.0  1998/09/25  11.38
                                     1
A              PRIMARY INPUT   3667  3812, 3814, 3839
AINSERT_TEST_MACRO
               PRIMARY INPUT      3  16
AL             PRIMARY INPUT   3714  3822, 3824
DCBD      L3   PRIMARY INPUT    224X 2329
IHBERMAC  L3   DCBD            2331X 2954
L              PRIMARY INPUT   3657  3807, 3809
MAC1           PRIMARY INPUT     28  36
N              PRIMARY INPUT   3774  3832
O              PRIMARY INPUT   3794  3834
SAMPLE    L1   PRIMARY INPUT      -  195C
SAMPMAC   L1   PRIMARY INPUT    153X 174
SAVE      L3   PRIMARY INPUT     43X 130
SL             PRIMARY INPUT   3724  3827, 3829
ST             PRIMARY INPUT   3677  3817, 3819
TYPCHKRX       PRIMARY INPUT   3586  3616, 3686, 3733
X              PRIMARY INPUT   3784  3837
XIT1      L1   PRIMARY INPUT      -  30C
XIT2      L2   PRIMARY INPUT      -  32C
XIT3      L1   PRIMARY INPUT      -  34C
```

Figure 17. Macro and Copy Code Cross Reference - with LIBMAC option

1 The "X" flag indicates the macro was read from a macro library and imbedded in the input source program immediately preceding the invocation of that macro. For example, in Figure 17, you can see that SAMPMAC was called by the PRIMARY INPUT stream from LIBRARY L1, at statement number 174, after being imbedded in the input stream at statement number 153.

You can suppress this section of the listing by specifying the NOMXREF assembler option, or the MXREF(SOURCE) assembler option.

DSECT Cross Reference

Figure 18 shows an example of the *DSECT Cross Reference* section of the listing. This section shows the names of all internal and external dummy sections defined in the program, and the statement number where the definition of the dummy section begins.

```
                                                 Dsect Cross Reference                                     Page   26
1          2          3          4
Dsect     Length      Id         Defn                                                    HLASM R3.0  1998/09/25  11.38
AXPRIL    0000003C  FFFFFFFD      655
AXPSIL    00000410  FFFFFFFC      771
AXPXITP   00000014  FFFFFFFE      641
IHADCB    00000060  FFFFFFFB      799
Statement
          00000050  FFFFFFFA     1370
WORKAREA  000001A8  FFFFFFFF      595
```

Figure 18. DSECT Cross Reference

1 Shows the name of each dummy section defined in your program.

2 Shows, in hexadecimal notation, the assembled byte length of the dummy section.

3 Shows the ESD ID assigned to the ESD entry for external dummy sections. For internal dummy sections it shows the control section ID assigned to the dummy control section. You can use this field in conjunction with the ID field

in the *Ordinary Symbol and Literal Cross Reference* section to relate symbols to a specific DSECT.

4 Shows the number of the statement where the definition of the dummy section begins.

You can suppress this section of the listing by specifying the NODXREF assembler option.

USING Map

Figure 19 shows an example of the *Using Map* section of the listing. It shows a summary of the USING, DROP, PUSH USING, and POP USING instructions used in your program.

```
                                                 Using Map                                     Page   27
                                                                             HLASM R3.0  1998/09/25  11.38
  1                       2    3              4        5        6        7  8
Stmt  -----Location----- Action ----------------Using----------------- Reg Max    Last  Label and Using Text
        Count     Id            Type          Value    Range     Id         Disp   Stmt
 170  00000000  00000001 USING  ORDINARY    00000000 00001000 00000001  15 02A     171  asmaxinv,r15
 175  00000030  00000001 DROP                                           15              r15
 185  00000034  00000001 USING  ORDINARY    00000000 00001000 00000001  12 000          asmaxinv,r12
 186  00000034  00000001 USING  ORDINARY    00000000 00001000 FFFFFFFD   7 034     508  axpril,r07
 187  00000034  00000001 USING  ORDINARY    00000000 00001000 FFFFFFFA   8 048     464  Statement,r08
 188  00000034  00000001 USING  ORDINARY    00000000 00001000 FFFFFFFC  10 404     474  axpsil,r10
 189  00000034  00000001 USING  ORDINARY    00000000 00001000 FFFFFFFB  11 052     465  ihadcb,r11
 190  00000034  00000001 USING  ORDINARY    00000000 00001000 00000001  12 589     519  asmaxinv,r12
 202  0000004E  00000001 USING  LABELED     00000000 00001000 FFFFFFFF   1 000          WA.WorkArea,r01
 203  0000004E  00000001 USING  LAB+DEPND  +0000014A 00000EB6 FFFFFFFB   1              local.ihadcb,WA.mydcb
 205  00000054  00000001 DROP                                            1              local
 212  0000006A  00000001 DROP                                            1              WA
 213  0000006A  00000001 USING  ORDINARY    00000000 00001000 FFFFFFFF  13 14A     527  WorkArea,r13
```

Figure 19. USING Map

1 Shows the number of the statement that contains the USING, DROP, PUSH USING, or POP USING instruction.

2 Indicates whether the instruction was a USING, DROP, PUSH, or POP instruction.

3 Shows the type of USING instruction. A USING instruction can be an ordinary USING, a labeled USING, a dependent USING, or a labeled dependent USING.

4 For ordinary and labeled USING instructions, this field indicates the base address specified in the USING. For dependent USING instructions, this field is prefixed with a plus sign (+) and indicates the hexadecimal offset of the address of the second operand from the base address specified in the corresponding ordinary USING.

5 Shows the range of the USING. For more information, see the description of the USING statement in the *High Level Assembler Language Reference*.

6 For USING instructions, this field indicates the ESDID of the section specified on the USING statement.

7 Indicates the registers specified in USING instructions, and DROP instructions. There is a separate line in the USING map for each register specified in the instruction.

8 Shows the maximum displacement from the base register that the assembler calculated when resolving symbolic addresses into base-displacement form.

You can suppress this section of the listing by specifying the USING(NOMAP) assembler option, or the NOUSING assembler option.

Diagnostic Cross Reference and Assembler Summary

Figure 20 shows an example of the *Diagnostic Cross Reference and Assembler Summary* section of the listing. This sample listing shows a combination of MVS and CMS data sets to highlight the differences in data set information.

This section includes a summary of the statements flagged with diagnostic messages, and provides statistics about the assembly. You cannot suppress this section unless you use a LISTING exit to discard the listing lines.

See the description of the sample LISTING exit on page 42, which lets you suppress this section.

```
                                  Diagnostic Cross Reference and Assembler Summary                             Page    9
                                                                                            HLASM R3.0  1998/09/25  11.38
Statements Flagged
         1
    1(P1,0), 3(P1,3), 4(P1,4), 5(P1,5), 6(P1,6), 7(P1,7), 8(P1,8), 170(L3:DCBD,2149)

 2  8 Statements Flagged in this Assembly       16 was Highest Severity Code
High Level Assembler, 5696-234, RELEASE 3.0
SYSTEM: CMS 11                   JOBNAME: (NOJOB)     STEPNAME: (NOSTEP)   PROCSTEP: (NOPROC)  3
Datasets Allocated for this Assembly  4
 Con DDname   Dataset Name                              Volume  Member
  P1 SYSIN    XITDIS   ASSEMBLE A1                      ADISK
  L1 SYSLIB   TEST     MACLIB   A1                      ADISK
  L2          DSECT    MACLIB   A1                      ADISK
  L3          OSMACRO  MACLIB   S2                      MNT190
  L4          OSMACRO1 MACLIB   S2                      MNT190
 5   SYSLIN   XITDIS   TEXT     A1                      ADISK
     SYSPRINT XITDIS   LISTING  A1                      ADISK

External Function Statistics  6
 ----Calls----  Message   Highest  Function
 SETAF   SETCF    Count  Severity  Name
     3       1        5      22    MSG
     1       0        2       8    MSG1
     1       0        1       0    MSG2
 7
Input/Output Exit Statistics
Exit Type  Name      Calls  ---Records---  Diagnostic
                            Added Deleted  Messages
LIBRARY    CTLXIT     258      0       0        2
LISTING    ASMAXPRT   195      0      52        0

   4622K allocated to Buffer Pool,      489K would be required for this to be an In-Storage Assembly
 8  16 Primary Input Records Read      3072 Library Records Read     0 Work File Reads
    141 Primary Print Records Written      2 Punch Records Written    0 Work File Writes
      0 ADATA Records Written
Assembly Start Time: 12.06.06 Stop Time: 12.06.07 Processor Time: 00.00.00.1771  9
Return Code 016
```

Figure 20. Diagnostic Cross Reference and Assembler Summary

1 The statement number of a statement that causes an error message, or contains an MNOTE instruction, appears in this list. Flagged statements are shown in either of two formats. When assembler option FLAG(NORECORD) is specified, only the statement number is shown. When assembler option FLAG(RECORD) is specified, the format is: *statement(dsnum:member,record)*, where:

statement	is the sequential, absolute statement number as shown in the source and object section of the listing.
dsnum	is the value applied to the source or library dataset, showing the type of input file and the concatenation number. "P" indicates the statement was read from the primary input source, and "L" indicates the statement was read from a library. This value is cross-referenced to the input datasets listed under the sub-heading "Datasets Allocated for this Assembly" **4** .

member	is the name of the macro from which the statement was read. On MVS, this may also be the name of a partitioned data set member that is included in the primary input (SYSIN) concatenation.
record	is the relative record number from the start of the dataset or member which contains the flagged statement.

2 The number of statements flagged, and the highest non-zero severity code of all messages issued.

3 Provides information about the system on which the assembly was run.

4 On MVS and CMS, all data sets used in the assembly are listed by their standard ddname. The data set information includes the data set name, and the serial number of the volume containing the data set. On MVS, the data set information may also include the name of a member of a partitioned data set (PDS).

If a user exit provides the data set information, then the data set name is the value extracted from the Exit-Specific Information block described in the *High Level Assembler Programmer's Guide*.

The "Con" column shows the concatenation value assigned for each input data set. You use this value to cross-reference flagged statements, and macros and copy code members listed in the Macro and Copy Code Cross Reference section.

5 Output data sets do not have a concatenation value.

| **6** The usage statistics of external functions for the assembly. The following statistics are reported:

SETAF function calls	The number of times the function was called from a SETAF assembler instruction.
SETCF function calls	The number of times the function was called from a SETCF assembler instruction.
Messages issued	The number of times the function requested that a message be issued.
Messages severity	The maximum severity for the messages issued by this function.
Function Name	The name of the external function module.

7 The usage statistics of the I/O exits you specified for the assembly. If you do not specify an exit, the assembler does not produce any statistics. The following statistics are reported:

Exit Type	The type of exit.
Name	The name of the exit module as specified in the EXIT assembler option.
Calls	The number of times the exit was called.
Records	The number of records added and deleted by the exit.
Diagnostic Messages	The number of diagnostic messages printed, as a result of exit processing.

All counts are shown right justified and leading zeroes are suppressed, unless the count is zero.

8 Statistics about the assembly.

9 On VSE, the assembly start and stop times in hours, minutes and seconds.

On MVS and CMS, the assembly start and stop times in hours, minutes and seconds and the approximate amount of processor time used for the assembly, in hours, minutes, and seconds to four decimal places.

Improved Page-Break Handling

In order to prevent unnecessary page ejects that leave blank pages in the listing, the assembler takes into account the effect EJECT, SPACE and TITLE instructions have when the assembler listing page is full. The EJECT and TITLE instruction explicitly starts a new page, while the assembler implicitly starts a new page when the current page is full.

When an explicit new page is pending the following processing occurs:

- Successive EJECT statements are ignored
- Successive TITLE statements allow the title to change but the EJECT is ignored
- A SPACE statement forces a new page heading to be written, followed by the given number of blank lines. The number of blank lines specified can cause an implicit page eject if the number exceeds the the page depth.

When an implicit new page is pending the following processing occurs:

- An EJECT statement converts the implicit new page to an explicit pending new page.
- A TITLE statement converts the implicit new page to an explicit pending new page and redefines the title.
- Any other statement forces a new page heading to be printed.

Diagnostic Messages in Open Code

The *Source and Object* section of the assembler listing shows in-line diagnostic messages. The *Diagnostic Cross Reference and Assembler Summary* shows the total number of diagnostic messages and the statement numbers of flagged statements. Many in-line messages include a copy of the segment of the statement that is in error.

When you specify the FLAG assembler option, the assembler may print additional diagnostic messages. The FLAG(ALIGN) option directs the assembler to issue diagnostic messages when there is an alignment error between an operation code and the operand data address. The FLAG(CONT) option directs the assembler to issue diagnostic messages when the assembler detects a possible continuation error. The FLAG(RECORD) option directs the assembler to print an additional informational message after the last error diagnostic message for each statement in error. Figure 21 shows the effect of the FLAG(RECORD) option:

```
000000                            1         CSECT
                                            ⋮
                                 22         COMM
** ASMA057E Undefined operation code - COMM
** ASMA435I Record 22 in 'HLASM3.SAMPLE.SOURCE(SAMP01)' on volume: HLASM3
                                            ⋮
000000                           35         DS    (*+5)F
** ASMA032E Relocatable value found when absolute value required - (*+5)F
** ASMA435I Record 35 in 'HLASM3.SAMPLE.SOURCE(SAMP01)' on volume: HLASM3
000000 00000000                  36 2NAME    DC    F'0'
** ASMA147E Symbol too long, or first character not a letter - 2NAME
** ASMA435I Record 36 in 'HLASM3.SAMPLE.SOURCE(SAMP01)' on volume: HLASM3
                                            ⋮
                                118 &C       SETC  'AGO'
                                119          &C    .X
   ASMA001E Operation code not allowed to be generated - AGO
   ASMA435I Record 119 in 'HLASM3.SAMPLE.SOURCE(SAMP01)' on volume: HLASM3
                                            ⋮
                                151          END
```

Figure 21. In-line Error Messages in Open Code

You can locate messages in your assembly listing by searching for "** ASMA" in the listing. The preferred alternative is to specify the TERM option.

Macro-Generated Statements

A macro-generated statement is a statement generated by the assembler after a macro call. During macro generation, the assembler copies any model statements processed in the macro definition into the input stream for further processing. Model statements are statements from which assembler language statements are generated during conditional assembly. You can use variable symbols as points of substitution in a model statement to vary the contents or format of a generated statement.

Open Code: Model statements can also be included in open code by using variable symbols as points of substitution.

Sequence Field in Macro-Generated Statements

The *Source and Object* section of the listing includes an identification-sequence field for macro-generated statements. This field is printed to the extreme right of each generated statement in the listing.

When a statement is generated from a library macro, the identification-sequence field of the generated statement contains the nesting level of the macro call in the first two columns, a hyphen in the third column, and the macro definition name in the remaining columns.

On MVS and CMS, when you specify the LIST(121) assembler option, the first 5 characters of the macro name are printed after the hyphen. When you specify the LIST(133) assembler option, the first 8 characters of the macro name are printed after the hyphen.

On VSE, only the first 5 characters of the macro name are printed after the hyphen.

This information can be an important diagnostic aid when analyzing output dealing with macro calls within macro calls.

When a statement is generated from an in-line macro or a copied library macro, the identification-sequence field of the generated statement contains the nesting level of the macro call in the first two columns, a hyphen in the third column, and the model statement number from the definition in the remaining columns.

Format of Macro-Generated Statements

Whenever possible, the assembler prints a generated statement in the same format as the corresponding macro-definition (model) statement. The assembler preserves the starting columns of the operation, operand, and comments fields unless they are displaced by field substitution, as shown in the following example:

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                   HLASM R3.0  1998/09/25  11.38
                                      1          macro
                                      2          macgen
                                      3 &A       SETC 'abcdefghijklmnopq'
                                      4 &A       LA   1,4                   Comment
                                      5 &B       SETC 'abc'
                                      6 &B       LA   1,4                   Comment
                                      7          mend
                                      8          macgen
000000 4110 0004            00004     9+abcdefghijklmnopq LA 1,4            Comment                              01-00004
000004 4110 0004            00004    10+abc      LA   1,4                   Comment                              01-00006
                                     11          end
```

Figure 22. Format of macro-generated statements

Macro-Generated Statements with PRINT NOGEN

The PRINT NOGEN instruction suppresses the printing of all statements generated by the processing of a macro. PRINT NOGEN also suppress the generated statement for model statements in open code. When the PRINT NOGEN instruction is in effect, the assembler prints one of the following on the same line as the macro call or model statement:

- The object code for the first instruction generated. The object code includes the data that is shown under the ADDR1 and ADDR2 columns of the assembler listing.
- The first 8 bytes of generated data from a DC instruction

When the assembler forces alignment of an instruction or data constant, it generates zeros in the object code and prints the generated object code in the listing. When you use the PRINT NOGEN instruction the generated zeros are not printed.

Note: If the next line to print after macro call or model statement is a diagnostic message, the object code or generated data is not shown in the assembler listing.

Figure 23 shows the object code of the first statement generated for the `wto` macro instruction when PRINT NOGEN is effective. The data constant (DC) for `jump` causes 7 bytes of binary zeroes to be generated before the DC to align the constant on a double word. With PRINT NOGEN effective, these are not shown, but the location counter accounts for them.

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R3.0  1998/09/25  11.38
                                                    ⋮
000016 1851                         13         lr    5,1
                                    14         print nogen
000018 4510 F026           00002    15         wto 'Hello'
000028 C1                           23         dc    cl1'A'
000030 4238000000000000             24 jump     dc    d'56'
                                                    ⋮
```

Figure 23. The effect of the PRINT NOGEN instruction

Diagnostic Messages in Macro Assembly

The diagnostic facilities for High Level Assembler include diagnostic messages for format errors within macro definitions, and assembly errors caused by statements generated by the macro.

Error Messages for a Library Macro Definition

Format errors within a particular library macro definition are listed directly following the first call to that macro. Subsequent calls to the library macro do not result in this type of diagnostic. You can bring the macro definition into the source program with a COPY statement or by using the LIBMAC assembler option. The format errors then follow immediately after the statements in error. The macro definition in Figure 24 shows a format error in the LCLC instruction:

```
Name          Operation      Operand             Comment

              MACRO
              MAC1
              ⋮
              LCLC           &.A                 Invalid variable symbol
              ⋮
&N            SETA           &A
              ⋮
              MEND
```

Figure 24. Macro Definition with Format Error

Figure 25 shows the placement of error messages when the macro is called:

```
                                    1          MAC1
** ASMA024E Invalid variable symbol - MACRO - MAC1
                                   ⋮
** ASMA003E Undeclared variable symbol; default=0, null, or type=U - LIBMA/A
                                   ⋮
                                   36          MAC1
                                   ⋮
** ASMA003E Undeclared variable symbol; default=0, null, or type=U - LIBMA/A
                                   ⋮
                                   66          END
```

Figure 25. Error Messages for a Library Macro Definition

Error Messages for Source Program Macro Definitions

The assembler prints diagnostic messages for macro-generated statements even if the PRINT NOGEN instruction is in effect. In-line macro editing error diagnostic messages are inserted in the listing directly following the macro definition statement in error. Errors analyzed during macro generation produce in-line messages in the generated statements.

Terminal Output

On MVS and CMS, the TERM option lets you receive a summary of the assembly at your terminal. You may direct the terminal output to a disk data set.

On VSE, the TERM option lets you send a summary of the assembly to SYSLOG.

The output from the assembly includes all error diagnostic messages and the source statement in error. It also shows the number of flagged statements and the highest severity code.

The terminal output can be shown in two formats. Figure 26, the wide format, shows the source statements in the same columns as they were in the input data set. Figure 27, the narrow format, shows the source statements which have been compressed by replacing multiple consecutive blanks with a single blank. Use the TERM assembler option to control the format.

```
                                        1 &abc setc l'f
                                  00001000
 ** ASMA154E Operand must be absolute, predefined symbols; set to zero - OPENC/
l'f
 000000                                 +     dc c''
                                   00002000
 ** ASMA074E Illegal syntax in expression - '

 Assembler Done      2 Statements Flagged /   8 was Highest Severity Code
```

Figure 26. Sample terminal output in the WIDE format

```
  1 &abc setc l'f 00001000
 ** ASMA154E Operand must be absolute, predefined symbols; set to zero - OPENC/
l'f
 000000 + dc c'' 00002000
 ** ASMA074E Illegal syntax in expression - '

 Assembler Done      2 Statements Flagged /   8 was Highest Severity Code
```

Figure 27. Sample terminal output in the NARROW format

You can replace or modify the terminal output using a TERM user exit. See Chapter 5, “Using Exits to Complement File Processing” on page 39.

Input/Output Enhancements

High Level Assembler includes the following enhancements:

- QSAM Input/Output

  The assembler uses QSAM input/output for all sequential data sets.

- System-Determined Blocksize

  Under MVS/ESA, High Level Assembler supports MVS/DFP System-Determined Blocksize (SDB) for all output datasets, except SYSPUNCH and SYSLIN.

  SDB is applicable when all of the following conditions are true:

  – You run High Level Assembler under an MVS/ESA operating system that includes a MVS/DFP level of 3.1 or higher.

  – You DO NOT allocate the data set to SYSOUT.

  – Your JCL omits the blocksize, or specifies a blocksize of zero.

  – You specify a record length (LRECL).

  – You specify a record format (RECFM).

  – You specify a data set organization (DSORG).

  If these conditions are met, MVS/DFP selects the appropriate blocksize for a new data set depending on the device type you select for output.

  If the System-Determined Blocksize feature is not available, and your JCL omits the blocksize, or specifies a blocksize of zero, the assembler uses the logical record length as the blocksize.

CMS Interface Command

The name of the CMS interface command is ASMAHL. Your installation can create a synonym for ASMAHL when High Level Assembler is installed.

You can specify assembler options as parameters when you issue the High Level Assembler command. You may delimit each parameter using either a space or comma. There must be no intervening spaces when you specify suboptions and their delimiters.

The following invocation of High Level Assembler is not correct:

```
ASMAHL XREF( SHORT )
```

The assembly continues but issues message `ASMA400W ERROR IN INVOCATION PARAMETER` in the *High Level Assembler Options Summary* section of the assembly listing.

The correct way to specify the option is as follows:

```
ASMAHL XREF(SHORT)
```

The Assembler H Version 2 CMS-specific options NUM, STMT, and TERM have been removed. SYSTERM support is provided by the standard assembler TERM option.

The new SEG and NOSEG options let you specify from where CMS should load the High Level Assembler modules. By default the assembler loads its modules from the Logical Saved Segment (LSEG), but if the LSEG is not available, it loads the modules from disk. You can specify the NOSEG option to force the assembler to load its modules from disk, or you can specify the SEG option to force the assembler to load its modules from the Logical Saved Segment (LSEG). If the assembler cannot load its modules it terminates with an error message.

Macro Trace Facility (MHELP)

The assembler provides you with a set of trace and dump facilities to assist you in debugging errors in your macros and conditional assembly language. You use the MHELP instruction to invoke these trace and dump facilities. You can code a MHELP instruction anywhere in open code or in macro definitions. The operands on the MHELP instruction let you control which facilities to invoke. Each trace or dump remains in effect until you supersede it with another MHELP instruction.

The MHELP instruction lets you select one or more of the following facilities:

Macro Call Trace
A one-line trace for each macro call

Macro Branch Trace
A one-line trace for each AGO and true AIF conditional assembly statement within a macro

Macro Entry Dump
A dump of parameter values from the macro dictionary immediately after a macro call is processed

Macro Exit Dump
A dump of SET symbol values from the macro dictionary on encountering a MEND or MEXIT statement

Macro AIF Dump
A dump of SET symbol values from the macro dictionary immediately before each AIF statement that is encountered

Global Suppression
Suppresses the dumping of global SET symbols in the two preceding types of dump

Macro Hex Dump
An EBCDIC and hexadecimal dump of the parameters and SETC symbol values when you select the Macro AIF dump, the Macro Exit dump or the Macro Entry dump

MHELP Suppression
Stops all active MHELP options.

MHELP Control on &SYSNDX
Controls the maximum value of the &SYSNDX system variable symbol. The limit is set by specifying the number in the operand of the MHELP instruction. When the &SYSNDX value is exceeded, the assembler produces a diagnostic message, terminates all current macro generation, and ignores all subsequent macro calls.

Abnormal Termination of Assembly

Whenever the assembler detects an error condition that prevents the assembly from completing, it issues an assembly termination message and, in most cases, produces a specially formatted dump. This feature helps you determine the nature of the error. The dump is also useful if the abnormal termination is caused by an error in the assembler itself.

Diagnosis Facility

If there is an error in the assembler, the IBM service representative may ask for the output produced by the assembler, and the source program to help debug the error. A new internal trace facility in the assembler can provide the IBM service representative with additional debugging information. The IBM service representative determines the need for this information and the circumstances under which it can be produced. Until this facility is invoked, its inclusion in the assembler does not impact the performance.

Chapter 7. Associated Data Architecture

This chapter describes High Level Assembler support for the associated data architecture. Associated data was previously known as assembler language program data. This support includes a general-use programming interface which lets you write programs to use the associated data records the High Level Assembler produces.

The associated data (ADATA) file contains language-dependent and language-independent records. Language-dependent records contain information that is relevant only to programs assembled by the High Level Assembler. Language-independent records contain information that is common to all programming languages that produce ADATA records, and includes information about the environment the program is assembled in. You use the ADATA assembler option to produce this file.

The ADATA file contains variable-length blocked records. The maximum record length is 8188 bytes.

The file contains records classified into different record types. Each type of record provides information about the assembler language program being assembled. Each record consists of two parts:

- A 12-byte header section (increased from 8 bytes), which has the same structure for all record types
- A variable-length data section, which varies by record type

The header section contains:

- The language code
- The record code, which identifies the type of record
- The associated data file architecture level
- A continuation flag indicator
- The length of data following

The records written to the ADATA file are:

Job Identification
This record provides information about the assembly job, and its environment, including the names of primary input files.

ADATA Identification
This record contains the Universal Time, and the Coded Character Set used by the assembler.

ADATA Compilation-Unit (Start)
This record contains the assembly start time.

ADATA Compilation-Unit (End)
This record contains the assembly stop time, and the number of ADATA records written.

Output File Information
This record provides information about the data sets the assembler produces.

Options This record contains the assembler options specified for the assembly.

External Symbol Dictionary (ESD)
This record describes all the control sections, including DSECTs, defined in the program.

Source Analysis
This record contains the assembled source statements, with additional data describing the type and specific contents of the statement.

Source Error
This record contains error message information the assembler produces after a source statement in error.

DC/DS This record describes the constant or storage defined by a source program statement that contains a DC or DS instruction. If a source program statement contains a DC or DS instruction, then a DC/DS record is written following the Source record.

DC Extension
This record describes the object code generated by a DC statement when the DC statement has repeating fields. This record is only created if the DC statement has a duplication factor greater than 1 and at least one of the operand values has a reference to the current location counter (*****).

Machine Instruction
This record describes the object code generated for a source program statement. If a source program statement causes machine instructions to be generated, then a Machine Instruction record is written following the Source record.

Relocation Dictionary (RLD)
This record describes the relocation dictionary information that is included in the object module.

Symbol This record describes a single symbol or literal defined in the program.

Ordinary Symbol and Literal Cross Reference
This record describes references to a single symbol.

Macro and Copy Code Source Summary
This record describes the source of each macro and copy code member retrieved by the program.

Macro and Copy Code Cross Reference
This record describes references to a single macro or copy code member.

USING Map
This record describes all USING, DROP, PUSH USING, and POP USING statements in the program.

Statistics This record describes the statistics about the assembly.

User-supplied Information
This record contains data from the ADATA instruction.

| **Register Cross Reference**
| This record describes references to a single General Purpose register.

Chapter 8. Factors Improving Performance

This chapter describes some of the methods used by High Level Assembler that improve assembler execution performance relative to earlier assemblers. These improvements are gauged on the performance of typical assemblies, and there might be cases where the particular circumstances of your application or system configuration do not achieve them. The main factors that improve the performance of High Level Assembler are:

- Logical text stream and tables that are a result of the internal assembly process remain resident in virtual storage, whenever possible, throughout the assembly.
- High Level Assembler can be installed in shared virtual storage.
- High Level Assembler exploits 31-bit addressing.
- Two or more assemblies can be done with one invocation of the assembler.
- High Level Assembler edits only the macro definitions that it encounters during a given macro generation or during conditional assembly of open code, as controlled by AIF and AGO statements.
- Source text assembly passes are consolidated. The edit and generation of macro statements are done on a demand basis in one pass of the source text.

Resident Tables and Source Text: Keeping intermediate text, macro definition text, dictionaries, and symbol tables in main storage whenever possible improves performance. High Level Assembler only writes working storage blocks to the assembler work data set when its working storage is exhausted. Less input and output reduces system overhead and frees channels and input/output devices for other uses.

The amount of working storage allocated to High Level Assembler is determined by the SIZE assembler option, and is limited only by the amount available in the address space.

Shared Virtual Storage: High Level Assembler is a reentrant program that can be installed in shared virtual storage, such as the MVS Link Pack Area (LPA), a CMS logical saved segment or in a VSE Shared Virtual Area (SVA). When High Level Assembler is installed in shared virtual storage, the demand for system resources associated with loading the assembler load modules is reduced. In a multi-user environment, multiple users are able to share one copy of the assembler load modules.

31-bit Addressing: High Level Assembler takes advantage of the extended address space, available in extended architecture operating systems, by allowing most of its data areas to reside above the 16-megabyte line. I/O areas and exit parameter lists remain in storage below the 16-megabyte line to satisfy access method requirements and user exits using 24-bit addressing mode. The High Level Assembler's modules can be loaded above the 16-megabyte line, except for some initialization routines. The SIZE assembler option is used to control where the assembler work areas reside. 31-bit addressing increases the assembler's available work area, which allows larger programs than previously possible to be assembled in-storage. In-storage assemblies reduce the input and output system overhead and free channels and input/output devices for other uses.

© Copyright IBM Corp. 1981, 1998

Multiple Assembly: You can run multiple assemblies, known as batching, with one invocation of the assembler. Source records are placed together, with no intervening '/*' JCL statement.

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

Macro-Editing Process: High Level Assembler edits only those macro definitions encountered during a given macro generation or during conditional assembly or 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, High Level Assembler edits only the set of library macro definitions that are expanded; as a result, High Level Assembler may edit fewer library macro definitions than previous assemblers.

Unlike DOS/VSE Assembler, High Level Assembler requires that library macros be stored in source format. This removes the necessity to edit library macros before they can be stored in the library.

Consolidating Source Text Passes: Consolidating assembly source text passes and other new organization procedures reduce the number of internal processor instructions used to handle source text in High Level Assembler, which causes 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.

Appendix A. Assembler Options

High Level Assembler provides you with many assembler options for controlling the operation and output of the assembler. You can set default values at assembler installation time for most of these assembler options. You can also fix a default option so the option cannot be overridden at assembly time. See "IBM-Supplied Default Assembler Options" on page 18 for a list of the changes to the IBM-supplied default assembler options from High Level Assembler Release 2.

You specify the options at assembly time on:

- The JCL PARM parameter of the EXEC statement on MVS and VSE, or the ASMAHL command on CMS.
- The JCL OPTION statement on VSE.
- The *PROCESS assembler statement.

The assembler options are:

ADATA | NOADATA
Produce the associated data file.

ALIGN | NOALIGN
Check alignment of addresses in machine instructions and whether DC, DS, DXD, and CXD are aligned on correct boundaries.

ASA | NOASA
(MVS and CMS) Produce the assembly listing using American National Standard printer-control characters. If NOASA is specified the assembler uses machine printer-control characters.

BATCH | NOBATCH
Specify multiple assembler source programs are in the input data set.

COMPAT(*suboption*) | NOCOMPAT
Direct the assembler to remain compatible with earlier assemblers in its handling of lowercase characters in the source program, and its handling of sublists in SETC symbols, and its handling of unquoted macro operands. The LITTYPE suboption instructs the assembler to return 'U' as the type attribute for all literals.

DBCS | NODBCS
Specify that the source program contains double-byte characters.

DECK | NODECK
Produce an object module.

DXREF | NODXREF
Produce the *DSECT Cross Reference* section of the assembler listing.

ESD | NOESD
Produce the *External Symbol Dictionary* section of the assembler listing.

EXIT(*suboption1,suboption2,...*) | NOEXIT
Provide user exits to the assembler for input/output processing.

ADEXIT(*name(string)*) | NOADEXIT
Identify the name of a user-supplied ADATA exit module.

INEXIT(*name(string)***) | NOINEXIT**
Identify the name of a user-supplied SOURCE exit module.

LIBEXIT(*name(string)***) | NOLIBEXIT**
Identify the name of a user-supplied LIBRARY exit module.

OBJEXIT(*name(string)***) | NOOBJEXIT**
Identify the name of a user-supplied OBJECT exit module.

PRTEXIT(*name(string)***) | NOPRTEXIT**
Identify the name of a user-supplied LISTING exit module.

TRMEXIT(*name(string)***) | NOTRMEXIT**
Identify the name of a user-supplied TERM exit module.

FLAG(*suboption1,suboption2,...***)**
Specify one or more of the following:

- The level of error diagnostic messages to be written.
- Whether warning messages for alignment errors should be written.
- Whether warning messages for possible statement continuation errors should be written.
- Whether informational messages about an instruction relying on an implied length should be written.
- Whether warning messages about baseless resolution should be written.
- Whether message ASMA435I should be produced with each diagnostic message. Message ASMA435I provides the record number and dataset name of the statement in error.
- Whether warning message ASMA094 should be produced when the second subscript value of the substring notation indexes past the end of the character expression.

FOLD | NOFOLD
Convert lowercase characters to uppercase characters in the assembly listing.

INFO
Display service information selected by date.

LANGUAGE(EN | ES | DE | JP | UE)
Specify the language in which assembler diagnostic messages are presented. High Level Assembler lets you select any of the following:

- English mixed case (EN)
- English uppercase (UE)
- German (DE)
- Japanese (JP)
- Spanish (ES)

When you select either of the English languages, the assembler listing headings are produced in the same case as the diagnostic messages.

When you select either the German language or the Spanish language, the assembler listing headings are produced in mixed case English.

When you select the Japanese language, the assembler listing headings are produced in uppercase English.

The assembler uses the default language for messages produced on CMS by the High Level Assembler command.

LIBMAC | NOLIBMAC
Instruct the assembler to imbed library macro definitions in the input source program.

LINECOUNT(*integer*)
Specify the number of lines to print in each page of the assembly listing.

LIST | LIST(121 | 133 | MAX) | NOLIST
(MVS and CMS) Specify whether the assembler produces an assembly listing. The listing may be produced in 121-character format or 133-character format.

LIST | NOLIST
(VSE only) Specify whether the assembler produces an assembly listing.

MXREF | MXREF(FULL | SOURCE | XREF) | NOMXREF
Produce the *Macro and Copy Code Source Summary*, or the *Macro and Copy Code Cross Reference*, or both, in the assembly listing.

OBJECT | NOOBJECT
Produce an object module.

OPTABLE(DOS | ESA | UNI | XA | 370)
Specify the operation code table to use to process machine instructions in the source program.

PCONTROL(*suboption1,suboption2,...*) | NOPCONTROL
Specify whether the assembler should override certain PRINT statements in the source program.

PESTOP
Specify that the assembler should stop immediately if errors are detected in the invocation parameters.

PROFILE | PROFILE(*name*) | NOPROFILE
Specify the name of a library member, containing assembler source statements, that is copied immediately following an ICTL statement or *PROCESS statements, or both. The library member can be specified as a default in the installation options macro ASMAOPT.

RA2 | NORA2
Specify whether the assembler is to suppress error diagnostic message ASMA066 when 2-byte relocatable address constants are defined in the source program.

RENT | NORENT
Check for possible coding violations of program reenterability.

RLD | NORLD
Produce the *Relocation Dictionary* section of the assembler listing.

| **RXREF**
| Produce the *Register Cross Reference* section of the assembler listing.

SIZE(*value*)
Specify the amount of virtual storage that the assembler can use for working storage.

SYSPARM(*value*)
Specify the character string that is to be used as the value of the &SYSPARM system variable.

TERM(WIDE | NARROW) | NOTERM
Specify whether error diagnostic messages are to be written to the terminal data set on MVS and CMS, or SYSLOG on VSE.

TEST | NOTEST
Specify whether special symbol table data is to be generated as part of the object module.

TRANSLATE(AS | *suffix***) | NOTRANSLATE**
Specify whether characters contained in character (C-type) data constants (DCs) and literals should be translated using a user-supplied translation table. The suboption AS directs the assembler to use the ASCII translation table provided with High Level Assembler.

USING(*suboption1,suboption2,...***) | NOUSING**
Specify the level of monitoring of USING statements required, and whether the assembler is to generate a USING map as part of the assembly listing.

| **XOBJECT | GOFF | NOXOBJECT**
| (MVS and CMS) Set extended object format.

XREF(SHORT | UNREFS | FULL) | NOXREF
Produce the *Ordinary Symbol and Literal Cross Reference*, or the *Unreferenced Symbols Defined in CSECTs*, or both, in the assembly listing.

Appendix B. System Variable Symbols

System variable symbols are a special class of variable symbols, starting with the characters &SYS. There values are set by the assembler according to specific rules. You cannot declare system variable symbols in local SET symbols or global SET symbols, nor can you use them as symbolic parameters.

You can use these symbols as points of substitution in model statements and conditional assembly instructions. You can use some system variable symbols both inside macro definitions and in open code, and some system variable symbols only in macro definitions.

In High Level Assembler enhancements have been made to some system variable symbols and many new system variable symbols have been introduced.

New system variable symbols provided by High Level Assembler Release 3 are:

Variable	Description
&SYSCLOCK	A local-scope variable that holds the date and time at which a macro is generated.
&SYSMAC	A local-scope variable that can be subscripted, thus referring to the name of any of the macros opened between opencode and the current nesting level.
&SYSOPT_XOBJECT	A global-scope variable that indicates if the XOBJECT assembly option was specified.
&SYSM_HSEV	A global-scope variable that indicates the latest MNOTE severity so far for the assembly.
&SYSM_SEV	A global-scope variable that indicates the latest MNOTE severity for the macro most recently called from this level.

The system variable symbols provided by High Level Assembler Release 2 are:

Variable	Description
&SYSADATA_DSN	A local-scope variable containing the name of the data set where associated data (ADATA) records are written.
&SYSADATA_MEMBER	A local-scope variable containing the name of the partitioned data set member where associated data (ADATA) records are written.
&SYSADATA_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the ADATA data set.
&SYSLIN_DSN	A local-scope variable containing the name of the data set where object module records are written.
&SYSLIN_MEMBER	A local-scope variable containing the name of the partitioned data set member where object module records are written.

© Copyright IBM Corp. 1981, 1998

&SYSLIN_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the object module data set.
&SYSPRINT_DSN	A local-scope variable containing the name of the data set where listing records are written.
&SYSPRINT_MEMBER	A local-scope variable containing the name of the partitioned data set member where listing records are written.
&SYSPRINT_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the listing data set.
&SYSPUNCH_DSN	A local-scope variable containing the name of the data set where object module records are written.
&SYSPUNCH_MEMBER	A local-scope variable containing the name of the partitioned data set member where object module records are written.
&SYSPUNCH_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the object module data set.
&SYSTERM_DSN	A local-scope variable containing the name of the data set where terminal messages are written.
&SYSTERM_MEMBER	A local-scope variable containing the name of the partitioned data set member where terminal messages are written.
&SYSTERM_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the terminal messages data set.

System variable symbols provided by High Level Assembler Release 1 are:

Variable	**Description**
&SYSASM	A global-scope variable containing the name of the assembler product being used.
&SYSDATC	A global-scope variable containing the date, with the century designation included, in the form *YYYYMMDD*.
&SYSIN_DSN	A local-scope variable containing the name of the input data set.
&SYSIN_MEMBER	A local-scope variable containing the name of the current member in the input data set.
&SYSIN_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the input data set.
&SYSJOB	A global-scope variable containing the job name of the assembly job, if available, or '(NOJOB)'.

&SYSLIB_DSN	A local-scope variable containing the name of the library data set from which the current macro was retrieved.
&SYSLIB_MEMBER	A local-scope variable containing the name of the current macro retrieved from the library data set.
&SYSLIB_VOLUME	A local-scope variable containing the volume identifier of the first volume containing the library data set from which the current macro was retrieved.
&SYSNEST	A local-scope variable containing the current macro nesting level. &SYSNEST is set to 1 for a macro called from open code.
&SYSOPT_DBCS	A global-scope Boolean variable containing the value 1 if the DBCS assembler option was specified, or 0 if NODBCS was specified.
&SYSOPT_OPTABLE	A global-scope variable containing the name of the operation code table specified in the OPTABLE assembler option.
&SYSOPT_RENT	A global-scope Boolean variable containing the value 1 if the RENT assembler option was specified, or 0 if NORENT was specified.
&SYSSEQF	A local-scope variable containing the identification-sequence field information of the macro instruction in open code that caused, directly or indirectly, the macro to be called.
&SYSSTEP	A global-scope variable containing the step-name, if available, or '(NOSTEP)'.
&SYSSTMT	A global-scope variable that contains the statement number of the next statement to be generated.
&SYSSTYP	A local-scope variable containing the current control section type (CSECT, DSECT, RSECT or COM) at the time the macro is called.
&SYSTEM_ID	A global-scope variable containing the name and release level of the operating system under which the assembly is run.
&SYSVER	A global-scope variable containing the maintenance version, release, and modification level of the assembler.

In addition, High Level Assembler provides the following system variable symbols not provided by DOS/VSE Assembler but provided by Assembler H Version 2:

Variable	**Description**
&SYSDATE	A global-scope variable containing the date in the form *MM/DD/YY*.
&SYSLOC	A local-scope variable containing the name of the location counter now in effect. &SYSLOC can only be used in macro definitions.

&SYSNDX	A local-scope variable containing a number from 1 to 9999999. Each time a macro definition is called, the number in &SYSNDX increases by 1.
&SYSTIME	A global-scope variable containing the time the assembly started, in the form *HH.MM*.

Appendix C. Hardware and Software Requirements

This appendix describes the environments in which High Level Assembler runs.

Hardware Requirements

High Level Assembler, and its generated object programs, can run in any IBM ES/9000, 3090, 308X, 43XX, or 937X processor supported by the operating systems listed below under Software Requirements. However, you can only run a generated object program that uses 370-XA machine instructions on a 370-XA mode processor under an operating system that provides the necessary architecture support for the 370-XA instructions used. Similarly, you can only run a generated object program that uses ESA/370 or ESA/390 machine instructions on an associated processor under an operating system that provides the necessary architecture support for the ESA/370 and ESA/390 instructions used.

Software Requirements

High Level Assembler runs under the operating systems listed below. Unless otherwise stated, the assembler also operates under subsequent versions, releases, and modification levels of these systems:

MVS/ESA System Product Version 4 Release 2
MVS/ESA System Product Version 4 Release 3
MVS/ESA System Product Version 5 Release 1
| MVS/ESA System Product Version 5 Release 2
| OS/390 System Product Version 1 Release 1
| OS/390 System Product Version 1 Release 2
| OS/390 System Product Version 1 Release 3
| OS/390 System Product Version 2 Release 4
| OS/390 System Product Version 2 Release 5
| OS/390 System Product Version 2 Release 6
VM/ESA Release 1
VM/ESA Release 2
VSE/ESA Version 1 Release 3
| VSE/ESA Version 1 Release 4
| VSE/ESA Version 2 Release 2
| VSE/ESA Version 2 Release 3

In addition, installation of High Level Assembler requires one of the following:

MVS/ESA IBM System Modification Program/Extended (SMP/E)

All load modules are reentrant, and you can place them in the link pack area (LPA).

VM/ESA IBM VM Serviceability Enhancements Staged/Extended (VMSES/E) and VMFPLC2

Most load modules are reentrant, and you can place them in a logical saved segment.

VSE Maintain System History Program (MSHP) to install High Level Assembler. Most phases are reentrant, and you can place them in the shared virtual area (SVA).

Assembling under MVS

The minimum amount of virtual storage required by High Level Assembler is 560K bytes. 360K bytes of storage are required for High Level Assembler load modules. The rest of the storage allocated to the assembler is used for assembler working storage.

At assembly time, and during subsequent link-editing, High Level Assembler requires associated devices for the following types of input and output:

- Source program input
- Macro library input
- Printed listing
- Object module in relocatable card-image format, or the new object-file format
- Terminal output
- ADATA output
- Work file

Figure 28 shows the DDNAME and allowed device types associated with a particular class of assembler input or output:

Figure 28. Assembler Input/Output Devices (MVS)

Function	**DDNAME**	**Device Type**	**When Required**
Input	SYSIN	DASD Magnetic tape Card reader	Always[1]
Macro Library	SYSLIB	DASD	When a library macro is called or a COPY statement used[1]
Print	SYSPRINT	DASD Magnetic tape Printer	When the LIST assembler option is specified[1]
Output to Linkage Editor	SYSLIN	DASD Magnetic tape	When the OBJECT assembler option or the XOBJECT assembler option is specified[1]
Output to Linkage Editor (card deck)	SYSPUNCH	DASD Magnetic tape Card punch	When the DECK assembler option is specified[1]
Display	SYSTERM	DASD Magnetic tape Terminal Printer	When the TERM assembler option is specified[1]
Assembler Language Program Data	SYSADATA	DASD Magnetic tape	When the ADATA assembler option is specified
Working Storage	SYSUT1	DASD	When adequate storage is not available

Note:

1. You can specify a user-supplied exit in place of this device For more information about the EXIT option, see Appendix A, “Assembler Options” on page 71.

Assembling under VM/CMS

High Level Assembler runs under the Conversational Monitor System (CMS) component of VM/ESA, and, depending upon system requirements, requires a virtual machine size of at least 1800K bytes.

A minimum of 560K bytes of storage is required by High Level Assembler. 360K bytes of storage are required for High Level Assembler load modules. The rest of the storage allocated to the assembler is used for assembler working storage.

At assembly time, and during subsequent object module processing, High Level Assembler requires associated devices for the following types of input and output:

- Source program input
- Macro library input
- Printed listing
- Object module in relocatable card-image format
- Terminal output
- ADATA output
- Work file

Figure 29 shows the characteristics of each device required at assembly time:

Figure 29. Assembler Input/Output Devices (CMS)

Function	DDNAME	Default File Type	Device Type	When Required
Input	SYSIN	ASSEMBLE	DASD Magnetic tape Card reader	Always[1]
Macro Library	SYSLIB	MACLIB	DASD	When a library macro is called or a COPY statement used[1]
Print	SYSPRINT	LISTING	DASD Magnetic tape Printer	When the LIST assembler option is specified
Object module	SYSLIN	TEXT	DASD Magnetic tape Card punch	When the OBJECT assembler option is specified[1]
Text deck	SYSPUNCH	N/A	DASD Magnetic tape Card punch	When the DECK assembler option is specified[1]
Display	SYSTERM	N/A	DASD Magnetic tape Terminal Printer	When the TERM assembler option is specified[1]
Assembler Language Program Data	SYSADATA	SYSADATA	DASD Magnetic tape	When the ADATA assembler option is specified
Working Storage	SYSUT1	SYSUT1	DASD	When adequate storage is not available

Note:

1. You can specify a user-supplied exit in place of this device For more information about the EXIT option, see Appendix A, “Assembler Options” on page 71.

Assembling under VSE

The minimum amount of virtual storage required by High Level Assembler is 560K bytes. 360K bytes of storage are required for High Level Assembler load modules. The rest of the storage allocated to the assembler is used for assembler working storage.

At assembly time, and during subsequent link-editing, High Level Assembler requires appropriate devices for the following types of input and output:

- Source program input
- Macro library input
- Printed listing
- Object module in relocatable card-image format
- Terminal output
- ADATA output
- Work file

Figure 30 shows the file name and allowed device types associated with a particular class of assembler input or output:

Figure 30. Assembler Input/Output Devices (VSE)

Function	File Name	Device Type	When Required
Input	IJSYSIN (SYSIPT)	DASD Magnetic tape Card reader	Always[1]
Macro Library	LIBRARIAN sublibraries	DASD	When a library macro is called or a COPY statement used[1]
Print	IJSYSLS (SYSLST)	DASD Magnetic tape Printer	When the LIST assembler option is specified[1]
Output to Linkage Editor	IJSYSLN (SYSLNK)	DASD Magnetic tape	When the OBJECT assembler option is specified
Output to LIBR utility (card deck)	IJSYSPH (SYSPCH)	DASD Magnetic tape Card punch	When the DECK assembler option is specified[1]
Display	SYSLOG	Terminal	When the TERM assembler option is specified[1]
Assembler Language Program Data	SYSADAT (SYSnnn)	DASD Magnetic tape	When the ADATA assembler option is specified
Working Storage	IJSYS03 (SYS003)	DASD	When adequate storage is not available

Note:

1. You can specify a user-supplied exit in place of this device For more information about the EXIT option, see Appendix A, “Assembler Options” on page 71.

Bibliography

High Level Assembler Publications

High Level Assembler General Information, GC26-4943

High Level Assembler Installation and Customization Guide, SC26-3494

High Level Assembler Language Reference, SC26-4940

High Level Assembler Licensed Program Specifications, GC26-4944

High Level Assembler Programmer's Guide, SC26-4941

Toolkit Feature Publications

High Level Assembler Toolkit Feature User's Guide, GC26-8710

High Level Assembler Toolkit Feature Debug Reference Summary, GC26-8712

High Level Assembler Toolkit Feature Interactive Debug Facility User's Guide, GC26-8709

High Level Assembler Toolkit Feature Installation and Customization Guide, GC26-8711

Related Publications (Architecture)

Enterprise Systems Architecture/390 Principles of Operation,SA22-7201

Vector Operations, SA22-7207

System/370 Enterprise Systems Architecture Principles of Operation, SA22-7200

System/370 Principles of Operation, GA22-7000

System/370 Extended Architecture Principles of Operation, SA22-7085

Related Publications for MVS

OS/390 MVS:

OS/390 MVS JCL Reference, GC28-1757

OS/390 MVS JCL User's Guide, GC28-1758

OS/390 MVS Assembler Services Guide, GC28-1757

OS/390 MVS Assembler Services Reference, GC28-1910

OS/390 MVS Auth Assembler Services Guide, GC28-1763

OS/390 MVS Auth Assembler Services Reference ALE-DYN, GC28-1764

OS/390 MVS Auth Assembler Services Reference ENF-ITT, GC28-1765

OS/390 MVS Auth Assembler Services Reference LLA-SDU, GC28-1766

OS/390 MVS Auth Assembler Services Reference SET-WTO, GC28-1767

OS/390 MVS System Codes, GC28-1780

OS/390 MVS System Commands, GC28-1781

OS/390 MVS System Messages, Vol 1 (ABA-ASA), GC28-1784

OS/390 MVS System Messages, Vol 2 (ASB-EWX), GC28-1785

OS/390 MVS System Messages, Vol 3 (GDE-IEB), GC28-1786

OS/390 MVS System Messages, Vol 4 (IEC-IFD), GC28-1787

OS/390 MVS System Messages, Vol 5 (IGD-IZP), GC28-1788

MVS/ESA Version 5:

MVS/ESA JCL Reference, GC28-1479

MVS/ESA JCL User's Guide, GC28-1473

MVS/ESA Programming: Assembler Services Guide, GC28-1466

MVS/ESA Programming: Assembler Services Guide, GC28-1474

MVS/ESA Programming: Authorized Assembler Services Guide, GC28-1467

MVS/ESA Programming: Authorized Assembler Services Reference Volumes 1 - 4, GC28-1475, GC28-1476, GC28-1477, GC28-1478

MVS/ESA System Codes, GC28-1486

MVS/ESA System Commands, GC28-1442

MVS/ESA System Messages Volumes 1 - 5 , GC28-1480, GC28-1481, GC28-1482, GC28-1483, GC28-1484

MVS/ESA Version 4:

MVS/ESA JCL User's Guide, GC28-1653

© Copyright IBM Corp. 1981, 1998

MVS/ESA Application Development Reference: Services for Assembler Language Programs, GC28-1642

MVS/ESA JCL Reference, GC28-1654

MVS/ESA System Codes, GC28-1664

MVS/ESA System Messages Volumes 1 - 5, GC28-1656, GC28-1657, GC28-1658, GC28-1659, GC28-1660

MVS/ESA OpenEdition®:

MVS/ESA OpenEdition MVS User's Guide, SC23-3013

OS/390 OpenEdition:

OS/390 OpenEdition User's Guide, SC28-1891

MVS/DFP™:

MVS/DFP Version 3.3: Utilities, SC26-4559

MVS/DFP Version 3.3: Linkage Editor and Loader, SC26-4564

DFSMS/MVS®:

DFSMS/MVS Program Management, SC26-4916

TSO/E (MVS):

TSO/E Command Reference, SC28-1881

TSO/E (OS/390):

OS/390 TSO/E Command Reference, SC28-1969

MVS SMP/E:

SMP/E Messages and Codes, SC28-1108

SMP/E Reference, SC28-1107

SMP/E Reference Summary, SX22-0006

SMP/E User's Guide, SC28-1302

OS/390 SMP/E:

OS/390 SMP/E Messages and Codes, SC28-1738

OS/390 SMP/E Reference, SC28-1806

OS/390 SMP/E Reference Summary, SX22-0037

OS/390 SMP/E User's Guide, SC28-1740

Related Publications for VM

VM/ESA CMS Application Development Guide, SC24-5450

VM/ESA CMS Application Development Guide for Assembler, SC24-5452

VM/ESA CMS Application Development Reference, SC24-5451

VM/ESA CMS Application Development Reference for Assembler, SC24-5453

VM/ESA CMS User's Guide, SC24-5460

VM/ESA XEDIT Command and Macro Reference, SC24-5464

VM/ESA XEDIT User's Guide, SC24-5463

VM/ESA CMS Planning and Administration Guide, SC24-5445

VM/ESA CP Command and Utility Reference, SC24-5519

VM/ESA CP Planning and Administration, SC24-5521

VMSES/E Introduction and Reference, SC24-5444

VM/ESA Service Guide, SC24-5527

VM/ESA CMS Command Reference, SC24-5461

VM/ESA SFS and CRR Planning, Administration, and Operation, SC24-5649

VM/ESA System Messages and Codes Reference, SC24-5529

VMSES/E 1.5 370 Feature Introduction and Reference, SC24-5680

VM/ESA 1.5 370 Feature Service Guide for 370, SC24-5429

Related Publications for VSE

VSE/ESA Administration, SC33-6505

VSE/ESA Guide to System Functions, SC33-6511

VSE/ESA Installation, SC33-6504

VSE/ESA Planning, SC33-6503

VSE/ESA System Control Statements, SC33-6513

General Publications

BRIEF OS/390 Software Management Cookbook, SG24-4775

Index

© Copyright IBM Corp. 1981, 1998

We'd Like to Hear from You

High Level Assembler for MVS® & VM & VSE
General Information
Release 3

Publication No. GC26-4943-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB2VVG at IBMMAIL
  - IBMLink: HLASMPUB at STLVM27
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

High Level Assembler for MVS® & VM & VSE
General Information
Release 3

Publication No. GC26-4943-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name	Address
Company or Organization	
Phone No.	

Readers' Comments
GC26-4943-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

GC26-4943-02

Cut or Fold
Along Line

IBM®

Program Number: 5696-234

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

High Level Assembler Publications

SC26-4941	High Level Assembler Programmer's Guide
GC26-4943	High Level Assembler General Information
GC26-4944	High Level Assembler Licensed Program Specifications
SC26-4940	High Level Assembler Language Reference
SC26-3494	High Level Assembler Installation and Customization Guide

High Level Assembler Toolkit Feature Publications

GC26-8709	High Level Assembler Toolkit Feature Interactive Debug Facility User's Guide
GC26-8710	High Level Assembler Toolkit Feature User's Guide
GC26-8711	High Level Assembler Toolkit Feature Installation and Customization Guide
GC26-8712	High Level Assembler Toolkit Feature Debug Reference Summary

GC26-4943-02

Spine information:

IBM HLASM General Information *Release 3*