## Language Reference

Release 5

## Language Reference

Release 5

## Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 425

## Fifth Edition (June 2004)

This edition applies to IBM High Level Assembler for MVS \& VM \& VSE, Release 5, 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
J87/D325
555 Bailey Avenue
SAN JOSE, CA 95141-1003
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 1982, 2004. All rights reserved.
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

## Contents

About this Manual ..... X
Who Should Use this Manual ..... X
Programming Interface Information ..... X
Organization of this Manual ..... xi
IBM High Level Assembler for MVS \& VM \& VSE Publications ..... xii
Publications ..... xii
Softcopy Publications ..... xiii
The High Level Assembler web site ..... xiii
Using LookAt to look up Message Explanations ..... xiii
Related Publications ..... xiv
Syntax Notation ..... xiv
Double-Byte Character Set Notation ..... xvi
Summary of Changes ..... xvii
Part 1. Assembler Language-Structure and Concepts ..... 1
Chapter 1. Introduction ..... 3
Language Compatibility ..... 4
Assembler Language ..... 4
Assembler Program ..... 6
Relationship of Assembler to Operating System ..... 8
Coding Made Easier ..... 9
Chapter 2. Coding and Structure ..... 11
Character Set ..... 11
Standard Character Set ..... 11
Double-Byte Character Set ..... 12
Translation Table ..... 14
Assembler Language Coding Conventions ..... 14
Field Boundaries ..... 14
Continuation Lines ..... 15
Blank Lines ..... 18
Comment Statement Format ..... 18
Instruction Statement Format ..... 19
Assembler Language Structure ..... 21
Overview of Assembler Language Structure ..... 23
Machine Instructions ..... 24
Assembler Instructions ..... 25
Conditional Assembly Instructions ..... 26
Macro Instructions ..... 27
Terms, Literals, and Expressions ..... 28
Terms ..... 28
Literals ..... 40
Expressions ..... 44
I Chapter 3. Program Structures and Addressing ..... 50
I Object Program Structures ..... 50
Source Program Structures ..... 51
Source Module ..... 52
Sections, Elements, and Parts ..... 52
Sections ..... 53
Reference Control Sections ..... 56
Classes (MVS and CMS) ..... 59
Parts (MVS and CMS) ..... 61
Location Counter Setting ..... 61
Addressing ..... 63
Addressing within Source Modules: Establishing Addressability ..... 64
Base Register Instructions ..... 66
Qualified Addressing ..... 66
Dependent Addressing ..... 67
Relative Addressing ..... 67
Literal Pools ..... 68
Establishing Residence and Addressing Mode ..... 68
Symbolic Linkages ..... 68
External Symbol Dictionary Entries ..... 72
Summary of Source and Object Program Structures ..... 73
Part 2. Machine and Assembler Instruction Statements ..... 75
Chapter 4. Machine Instruction Statements ..... 78
General Instructions ..... 78
Decimal Instructions ..... 79
Floating-Point Instructions ..... 79
Control Instructions ..... 79
Input/Output Operations ..... 80
Branching with Extended Mnemonic Codes ..... 80
Alternative Mnemonics for some Branch Relative Instructions ..... 83
Statement Formats ..... 83
Symbolic Operation Codes ..... 84
Operand Entries ..... 85
Registers ..... 86
Addresses ..... 87
Lengths ..... 90
Immediate Data ..... 91
Examples of Coded Machine Instructions ..... 91
RI Format ..... 91
RR Format ..... 93
RS Format ..... 94
RSI Format ..... 95
RX Format ..... 95
SI Format ..... 97
SS Format ..... 97
Chapter 5. Assembler Instruction Statements ..... 100
64-bit Addressing Mode ..... 101
*PROCESS Statement ..... 102
ACONTROL Instruction ..... 103
ADATA Instruction ..... 107
AINSERT Instruction ..... 108
ALIAS Instruction ..... 109
AMODE Instruction ..... 110
CATTR Instruction (MVS and CMS) ..... 112
CCW and CCWO Instructions ..... 115
CCW1 Instruction ..... 116
CEJECT Instruction ..... 118
CNOP Instruction ..... 119
COM Instruction ..... 121
COPY Instruction ..... 122
CSECT Instruction ..... 123
CXD Instruction ..... 125
DC Instruction ..... 126
Rules for DC Operand ..... 128
General Information About Constants ..... 129
Padding and Truncation of Values ..... 130
Subfield 1: Duplication Factor ..... 132
Subfield 2: Type ..... 133
Subfield 3: Type Extension ..... 134
Subfield 4: Program type ..... 135
Subfield 5: Modifier ..... 136
Subfield 6: Nominal Value ..... 140
DROP Instruction ..... 172
DS Instruction ..... 174
DSECT Instruction ..... 178
DXD Instruction ..... 180
EJECT Instruction ..... 181
END Instruction ..... 182
ENTRY Instruction ..... 183
EQU Instruction ..... 184
Using Conditional Assembly Values ..... 187
EXITCTL Instruction ..... 187
EXTRN Instruction ..... 189
ICTL Instruction ..... 189
ISEQ Instruction ..... 190
LOCTR Instruction ..... 191
LTORG Instruction ..... 193
Literal Pool ..... 194
Addressing Considerations ..... 195
Duplicate Literals ..... 195
MNOTE Instruction ..... 196
OPSYN Instruction ..... 198
ORG Instruction ..... 200
POP Instruction ..... 204
PRINT Instruction ..... 204
Process Statement ..... 208
PUNCH Instruction ..... 208
PUSH Instruction ..... 209
REPRO Instruction ..... 210
RMODE Instruction ..... 211
RSECT Instruction ..... 212
SPACE Instruction ..... 213
START Instruction ..... 214
TITLE Instruction ..... 215
USING Instruction ..... 218
How to Use the USING Instruction ..... 219
Base Registers for Absolute Addresses ..... 220

## Contents

Ordinary USING Instruction ..... 220
Labeled USING Instruction ..... 223
Dependent USING Instruction ..... 226
WXTRN Instruction ..... 229
XATTR Instruction (MVS and CMS) ..... 230
| Association of Code and Data Areas (MVS and CMS) ..... 232
Part 3. Macro Language ..... 235
Chapter 6. Introduction to Macro Language ..... 238
Using Macros ..... 238
Macro Definition ..... 238
Model Statements ..... 239
Processing Statements ..... 240
Comment Statements ..... 240
Macro Instruction ..... 241
Source and Library Macro Definitions ..... 241
Macro Library ..... 242
System Macro Instructions ..... 242
Conditional Assembly Language ..... 242
Chapter 7. How to Specify Macro Definitions ..... 243
Where to Define a Macro in a Source Module ..... 243
Format of a Macro Definition ..... 244
Macro Definition Header and Trailer ..... 244
MACRO Statement ..... 244
MEND Statement ..... 245
Macro Instruction Prototype ..... 245
Body of a Macro Definition ..... 247
Model Statements ..... 247
Variable Symbols as Points of Substitution ..... 248
Listing of Generated Fields ..... 248
Rules for Concatenation ..... 249
Rules for Model Statement Fields ..... 251
Symbolic Parameters ..... 253
Positional Parameters ..... 254
Keyword Parameters ..... 255
Combining Positional and Keyword Parameters ..... 255
Subscripted Symbolic Parameters ..... 255
Processing Statements ..... 256
Conditional Assembly Instructions ..... 256
Inner Macro Instructions ..... 256
Other Conditional Assembly Instructions ..... 256
AEJECT Instruction ..... 257
AINSERT Instruction ..... 257
AREAD Instruction ..... 257
ASPACE Instruction ..... 259
COPY Instruction ..... 260
MEXIT Instruction ..... 260
Comment Statements ..... 261
Ordinary Comment Statements ..... 261
Internal Macro Comment Statements ..... 261
System Variable Symbols ..... 262
Scope and Variability of System Variable Symbols ..... 262
\&SYSADATA_DSN System Variable Symbol ..... 263
\&SYSADATA_MEMBER System Variable Symbol ..... 264
\&SYSADATA_VOLUME System Variable Symbol ..... 265
\&SYSASM System Variable Symbol ..... 265
\&SYSCLOCK System Variable Symbol ..... 266
\&SYSDATC System Variable Symbol ..... 266
\&SYSDATE System Variable Symbol ..... 267
\&SYSECT System Variable Symbol ..... 267
\&SYSIN_DSN System Variable Symbol ..... 269
\&SYSIN_MEMBER System Variable Symbol ..... 270
\&SYSIN_VOLUME System Variable Symbol ..... 271
\&SYSJOB System Variable Symbol ..... 272
\&SYSLIB_DSN System Variable Symbol ..... 272
\&SYSLIB_MEMBER System Variable Symbol ..... 273
\&SYSLIB_VOLUME System Variable Symbol ..... 273
\&SYSLIN_DSN System Variable Symbol ..... 274
\&SYSLIN_MEMBER System Variable Symbol ..... 275
\&SYSLIN_VOLUME System Variable Symbol ..... 275
\&SYSLIST System Variable Symbol ..... 276
\&SYSLOC System Variable Symbol ..... 278
\&SYSMAC System Variable Symbol ..... 279
\&SYSM_HSEV System Variable Symbol ..... 279
\&SYSM_SEV System Variable Symbol ..... 280
\&SYSNDX System Variable Symbol ..... 281
\&SYSNEST System Variable Symbol ..... 283
\&SYSOPT_DBCS System Variable Symbol ..... 284
\&SYSOPT_OPTABLE System Variable Symbol ..... 285
\&SYSOPT_RENT System Variable Symbol ..... 285
\&SYSOPT_XOBJECT System Variable Symbol ..... 285
\&SYSPARM System Variable Symbol ..... 286
\&SYSPRINT_DSN System Variable Symbol ..... 286
\&SYSPRINT_MEMBER System Variable Symbol ..... 288
\&SYSPRINT_VOLUME System Variable Symbol ..... 288
\&SYSPUNCH_DSN System Variable Symbol ..... 289
\&SYSPUNCH_MEMBER System Variable Symbol ..... 290
\&SYSPUNCH_VOLUME System Variable Symbol ..... 290
\&SYSSEQF System Variable Symbol ..... 291
\&SYSSTEP System Variable Symbol ..... 292
\&SYSSTMT System Variable Symbol ..... 292
\&SYSSTYP System Variable Symbol ..... 292
\&SYSTEM_ID System Variable Symbol ..... 293
\&SYSTERM_DSN System Variable Symbol ..... 294
\&SYSTERM_MEMBER System Variable Symbol ..... 295
\&SYSTERM_VOLUME System Variable Symbol ..... 295
\&SYSTIME System Variable Symbol ..... 296
\&SYSVER System Variable Symbol ..... 296
Chapter 8. How to Write Macro Instructions ..... 297
Macro Instruction Format ..... 297
Alternative Formats for a Macro Instruction ..... 298
Name Entry ..... 299
Operation Entry ..... 299
Operand Entry ..... 300
Sublists in Operands ..... 304
Values in Operands ..... 307
Omitted Operands ..... 307
Unquoted Operands ..... 308
Special Characters ..... 308
Nesting Macro Instruction Definitions ..... 311
Inner and Outer Macro Instructions ..... 312
Levels of Macro Call Nesting ..... 312
General Rules and Restrictions ..... 313
Passing Values through Nesting Levels ..... 314
System Variable Symbols in Nested Macros ..... 316
Chapter 9. How to Write Conditional Assembly Instructions ..... 318
SET Symbols ..... 319
Subscripted SET Symbols ..... 319
Scope of SET Symbols ..... 319
Scope of Symbolic Parameters ..... 319
SET Symbol Specifications ..... 320
Subscripted SET Symbol Specification ..... 322
Created SET Symbols ..... 323
Data Attributes ..... 324
Attributes of Symbols and Expressions ..... 326
Type Attribute (T') ..... 328
Length Attribute (L') ..... 332
Scale Attribute ( $\mathrm{S}^{\prime}$ ) ..... 333
Integer Attribute (I') ..... 334
Count Attribute ( $\mathrm{K}^{\prime}$ ) ..... 335
Number Attribute ( $\mathrm{N}^{\prime}$ ) ..... 336
Defined Attribute (D') ..... 337
Operation Code Attribute ( $\mathrm{O}^{\prime}$ ) ..... 337
Sequence Symbols ..... 339
Lookahead ..... 340
Open Code ..... 342
Conditional Assembly Instructions ..... 343
Declaring SET Symbols ..... 343
GBLA, GBLB, and GBLC Instructions ..... 344
LCLA, LCLB, and LCLC Instructions ..... 345
Assigning Values to SET Symbols ..... 347
SETA Instruction ..... 347
SETB Instruction ..... 362
SETC Instruction ..... 369
Extended SET Statements ..... 387
SETAF Instruction ..... 388
SETCF Instruction ..... 389
Branching ..... 390
AIF Instruction ..... 390
AGO Instruction ..... 392
ACTR Instruction ..... 394
ANOP Instruction ..... 395
Chapter 10. MHELP Instruction ..... 397
Appendix A. Assembler Instructions ..... 402
Appendix B. Summary of Constants ..... 407
Appendix C. Macro and Conditional Assembly Language Summary ..... 409
Appendix D. Standard Character Set Code Table ..... 421
Notices ..... 425
Trademarks ..... 425
Bibliography ..... 427
High Level Assembler Publications ..... 427
Toolkit Feature Publications ..... 427
Related Publications (Architecture) ..... 427
Related Publications for MVS ..... 427
Related Publications for VM ..... 428
Related Publications for VSE ..... 428
Index ..... 429

## About this Manual

This manual describes the syntax of assembler language statements, and provides information about writing source programs that are to be assembled by IBM High Level Assembler for MVS \& VM \& VSE, Licensed Program 5696-234, hereafter referred to as High Level Assembler, or simply the assembler. It is meant to be used in conjunction with HLASM Programmer's Guide.

Detailed definitions of machine instructions are not included in this manual. See "Bibliography" on page 427 for a list of manuals that provide this information.

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 ${ }^{\text {™ }}$ | 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 ${ }^{\mathrm{TM}}$ ), to $\mathrm{OS} / 390 ®$, and to $\mathrm{z} / \mathrm{OS}$ ®.

CMS is used in this manual to refer to Conversational Monitor System on z/VM®.

VSE is used in this manual to refer to Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA ${ }^{\mathrm{TM}}$ ), and $\mathrm{z} / \mathrm{VSE}$.

## Who Should Use this Manual

HLASM Language Reference is for application programmers coding in the High Level Assembler language. It is not intended to be used for tutorial purposes, but is for reference only. If you are interested in learning more about assemblers, most libraries have tutorial books on the subject. It assumes you are familiar with the functional details of the Enterprise Systems Architecture, and the role of machine-language instructions in program execution.

## Programming Interface Information

This manual is intended to help the customer create application programs. This manual documents General-Use Programming Interface and Associated Guidance Information provided by IBM High Level Assembler for MVS \& VM \& VSE.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM High Level Assembler for MVS \& VM \& VSE.

## Organization of this Manual

This manual is organized as follows:

## Part 1, Assembler Language-Structure and Concepts

- Chapter 1, Introduction, describes the assembler language and how the assembler processes assembler language source statements. It also describes the relationship between the assembler and the operating system, and suggests ways to make the task of coding easier.
- Chapter 2, Coding and Structure, describes the coding rules for and the structure of the assembler language. It also describes the language elements in a program.
- Chapter 3, Program Structures and Addressing describes the concepts of addressability and symbolic addressing. It also describes control sections and the linkage between control sections.


## Part 2, Machine and Assembler Instruction Statements

- Chapter 4, Machine Instruction Statements describes the machine instruction types and their formats.
- Chapter 5, Assembler Instruction Statements describes the assembler instructions in alphabetical order.


## Part 3, Macro Language

- Chapter 6, Introduction to Macro Language, describes the macro facility concepts including macro definitions, macro instruction statements, source and library macro definitions, and conditional assembly language.
- Chapter 7, How to Specify Macro Definitions, describes the components of a macro definition.
- Chapter 8, How to Write Macro Instructions, describes how to call macro definitions using macro instructions.
- Chapter 9, How to Write Conditional Assembly Instructions, describes the conditional assembly language including SET symbols, sequence symbols, data attributes, branching, and the conditional assembly instructions.
- Chapter 10, MHELP Instruction, describes the MHELP instruction that you can use to control a set of macro trace and dump facilities.


## Appendixes

- Appendix A, Assembler Instructions summarizes the assembler instructions and assembler statements, and the related name and operand entries.
- Appendix B, Summary of Constants, summarizes the types of constants and related information.
- Appendix C, Macro and Conditional Assembly Language Summary, summarizes the macro language described in Part 3. This summary also includes a summary table of the system variable symbols.
- Appendix D, Standard Character Set Code Table, shows the code table for the assembler's standard character set.


## IBM High Level Assembler for MVS \& VM \& VSE Publications

High Level Assembler runs under MVS, VM and VSE. Its publications for the MVS, VM and VSE operating systems are described in this section.

## Publications

The books in the High Level Assembler library are shown in Figure 1. This figure shows which books can help you with specific tasks, such as application programming.

Figure 1. IBM High Level Assembler for MVS \& VM \& VSE Publications

| Task | Publication | Order Number |
| :---: | :---: | :---: |
| Evaluation and Planning | HLASM V1R5 General Information | GC26-4943 |
| Installation and Customization | HLASM V1R5 Installation and Customization Guide | SC26-3494 |
|  | HLASM V1R5 <br> Programmer's Guide | SC26-4941 |
|  | HLASM V1R5 Toolkit Feature Installation Guide | GC26-8711 |
| Application Programming | HLASM V1R5 <br> Programmer's Guide | SC26-4941 |
|  | HLASM V1R5 Language Reference | SC26-4940 |
|  | HLASM V1R5 General Information | GC26-4943 |
|  | HLASM V1R5 Toolkit Feature User's Guide | GC26-8710 |
|  | HLASM V1R5 Toolkit Feature Interactive Debug Facility User's Guide | GC26-8709 |
| Diagnosis | HLASM V1R5 Installation and Customization Guide | SC26-3494 |
| Warranty | HLASM V1R5 Licensed Program Specifications | GC26-4944 |

HLASM V1R5 General Information
Introduces you to the High Level Assembler product by describing what it does and which of your data processing needs it can fill. It is designed to help you evaluate High Level Assembler for your data processing operation and to plan for its use.
HLASM V1R5 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.

HLASM V1R5 Language Reference
Presents the rules for writing assembler language source programs to be assembled using High Level Assembler.
HLASM V1R5 Licensed Program Specifications
Contains a product description and product warranty information for High Level Assembler.
HLASM V1R5 Programmer's Guide
Describes how to assemble, debug, and run High Level Assembler programs.
HLASM V1R5 Toolkit Feature Installation Guide
Contains the information you need to install and customize, and diagnose failures in, the High Level Assembler Toolkit Feature.
HLASM V1R5 Toolkit Feature User's Guide
Describes how to use the High Level Assembler Toolkit Feature.
HLASM V1R5 Toolkit Feature Debug Reference Summary
Contains a reference summary of the High Level Assembler Interactive Debug Facility.
HLASM V1R5 Toolkit Feature Interactive Debug Facility User's Guide Describes how to use the High Level Assembler Interactive Debug Facility.

## Softcopy Publications

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

- z/OS V1Rx Collection, SK3T-4269
- z/OS V1Rx and Software Products DVD Collection, SK3T-4271
- z/VM Collection, SK2T-2067
- VSE Collection, SK2T-0060


## The High Level Assembler web site

The High Level Assembler web site, at
http://www.ibm.com/software/awdtools/hlasm
provides access to all HLASM publications, in downloadable or directly viewable PDF and BookMaster® formats.

The web site also provides access to other information relevant to High Level Assembler.

## Using LookAt to look up Message Explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from the following locations to find IBM message explanations for z/OS elements and features, z/VM, VSE/ESA, and Clusters for AIX® and Linux ${ }^{\mathrm{TM}}$ :

- The Internet. You can access IBM message explanations directly from the LookAt Web site at http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/
- Your z/OS TSO/E host system. You can install code on your z/OS or z/OS.e systems to access IBM message explanations, using LookAt from a TSO/E command line (for example, TSO/E prompt, ISPF, or z/OS UNIX® System Services running OMVS).
- Your Microsoft® Windows® workstation. You can install code to access IBM message explanations on the z/OS Collection (SK3T-4269), using LookAt from a Microsoft Windows command prompt (also known as the DOS command line).
- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example, Internet Explorer for Pocket PCs, Blazer, or Eudora for Palm OS, or Opera for Linux handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from a disk on your z/OS Collection (SK3T-4269), or from the LookAt Web site (click Download, and select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

## Related Publications

See "Bibliography" on page 427for a list of publications that supply information you might need while you are using High Level Assembler.

## Syntax Notation

Throughout this book, syntax descriptions use the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The $\rightsquigarrow$ - symbol indicates the beginning of a statement.
The $\longrightarrow$ symbol indicates that the statement syntax is continued on the next line.

The symbol indicates that a statement is continued from the previous line.
The $\longrightarrow \triangleleft$ indicates the end of a statement.
Diagrams of syntactical units other than complete statements start with the symbol and end with the $\longrightarrow$ symbol.

- Keywords appear in uppercase letters (for example, ASPACE) or upper and lower case (for example, PATHFile). They must be spelled exactly as shown. Lower case letters are optional (for example, you could enter the PATHFile keyword as PATHF, PATHFI, PATHFIL or PATHFILE).
Variables appear in all lowercase letters in a special typeface (for example, integer). They represent user-supplied names or values.
- If punctuation marks, parentheses, or such symbols are shown, they must be entered as part of the syntax.
- Required items appear on the horizontal line (the main path).
- $-I N S T R U C T I O N —$ required item-
- Optional items appear below the main path. If the item is optional and is the default, the item appears above the main path.
$\leadsto$ INSTRUCTION $-\frac{\text { default item— }}{\square} \square$
- When you can choose from two or more items, they appear vertically in a stack.

If you must choose one of the items, one item of the stack appears on the main path.


If choosing one of the items is optional, the whole stack appears below the main path.


- An arrow returning to the left above the main line indicates an item that can be repeated. When the repeat arrow contains a separator character, such as a comma, you must separate items with the separator character.
$\wedge$-INSTRUCTION— - repeatable item-

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

The following example shows how the syntax is used.


A The item is optional, and can be coded or not.
B The INSTRUCTION key word must be specified and coded as shown.

C
The item referred to by $\mathbf{1}$ is a required operand. Allowable choices for this operand are given in the fragment of the syntax diagram shown below 1 at the bottom of the diagram. The operand can also be repeated. That is, more than one choice can be specified, with each choice separated by a comma.

## Double-Byte Character Set Notation

Double-byte character set (DBCS) characters in terms, expressions, character strings, and comments are delimited by shift-out and shift-in characters. In this manual, the shift-out delimiter is represented pictorially by the < character, and the shift-in delimiter is represented pictorially by the > character. The EBCDIC codes for the shift-out and shift-in delimiters are $\mathrm{X}^{\prime} 0 \mathrm{E}^{\prime}$ and $\mathrm{X}^{\prime} \mathrm{OF}^{\prime}$, respectively.

The following figure summarizes the DBCS notation used throughout this manual.

| Character(s) | Represents |
| :--- | :--- |
| $<$ | Shift-out (SO) |
| $\boldsymbol{z}$ | Shift-in (SI) |
| D1D2D3... | Double-byte characters |
| DaDbDc... | Double-byte characters |
| $. A . B . C . ' . \& .$, | EBCDIC characters in double-byte form: A, B, C, single quotation mark, <br> ampersand, and comma. The dots separating the letters represent the <br> hexadecimal value X'42'. A double-byte character that contains the <br> value of an EBCDIC ampersand or single quotation mark in either byte <br> is not recognized as a delimiter when enclosed by SO and SI. |
| eeeeeee | Single-byte (EBCDIC) characters |
| abcd... | Single-byte (EBCDIC) characters |
| XXX | Extended continuation indicator for macro-generated statements |
| +++ | EBCDIC character containing all 0 bits. |
| $\boldsymbol{n}$ | EBCDIC character containing all 1 bits. |

## Summary of Changes

Date of Publication June 2004
Form of Publication Fifth Edition, SC26-4940-04
Here is a list of the changes to HLASM that are explained in this document.

## Extended support for machine instructions

- When the GOFF option is in force, the RI machine instructions are able to reference one or more external symbols "RI Format" on page 91.
- Support for 20-bit displacement instructions


## Changed Assembler instructions

- Quadword alignment
- CA data type
- CE data type
- 8-byte Q, R, J and V-cons
- CNOP
- DC/DS
- EQU
- ORG


## Changed assembler statements

- Support PART and PRIORITY attributes on the CATTR statement, for MVS and CMS "CATTR Instruction (MVS and CMS)" on page 112.


## AMODE

- ANY64 operand added "AMODE Instruction" on page 110


## Unified Opcode table

- A single opcode table is provided.
- OPTABLE option
- The OPTABLE option is permitted on the *PROCESS statement.


## ADATA enhancements

The following enhancements are made to ADATA:

- Revised layout of the ADATA records.
- The ADATA Exit processing is changed such that it mirrors the processing of other exits.
- GOFF ESD and RLD information are provided.
- The maximum record length is increased.


## Miscellany

- The ASCII translation table is upgraded.
- The Relocation Dictionary in the assembler listing is reformatted.
- A Numeric assembler version identifier is introduced.
- Additional diagnostic messages are provided.
- Enhancements to the External function parameter list.
- Enhancements to the Assembler summary listing.
- A new exit call - REINIT.
- Remove the internal dependency on the blocksize of SYSUT1.
- New limit of 1K for SETC and parameter string lengths.
- Enhancements to internal conditional assembly functions.


## Part 1. Assembler Language-Structure and Concepts

Chapter 1. Introduction ..... 3
Language Compatibility ..... 4
Assembler Language ..... 4
Machine Instructions ..... 5
Assembler Instructions ..... 5
Macro Instructions ..... 5
Assembler Program ..... 6
Basic Functions ..... 6
Associated Data ..... 6
Controlling the Assembly ..... 6
Processing Sequence ..... 7
Relationship of Assembler to Operating System ..... 8
Coding Made Easier ..... 9
Symbolic Representation of Program Elements ..... 9
Variety in Data Representation ..... 9
Controlling Address Assignment ..... 9
Relocatability ..... 9
Sectioning a Program ..... 9
Linkage between Source Modules ..... 10
Program Listings ..... 10
Multiple Source Modules ..... 10
Chapter 2. Coding and Structure ..... 11
Character Set ..... 11
Standard Character Set ..... 11
Double-Byte Character Set ..... 12
Translation Table ..... 14
Assembler Language Coding Conventions ..... 14
Field Boundaries ..... 14
Statement Field ..... 14
Continuation-Indicator Field ..... 15
Identification-Sequence Field ..... 15
Continuation Lines ..... 15
Alternative Statement Format ..... 16
Continuation of double-byte data ..... 16
Blank Lines ..... 18
Comment Statement Format ..... 18
Instruction Statement Format ..... 19
Statement Coding Rules ..... 19
Assembler Language Structure ..... 21
Overview of Assembler Language Structure ..... 23
Machine Instructions ..... 24
Assembler Instructions ..... 25
Conditional Assembly Instructions ..... 26
Macro Instructions ..... 27
Terms, Literals, and Expressions ..... 28
Terms ..... 28
Symbols ..... 29
Self-Defining Terms ..... 34
Location Counter ..... 36

## Part 1. Assembler Language-Structure and Concepts

Symbol Length Attribute Reference ..... 38
Other Attribute References ..... 40
Literals ..... 40
Literals, Constants, and Self-Defining Terms ..... 41
General Rules for Using Literals ..... 42
Literal Pool ..... 44
Expressions ..... 44
Rules for Coding Expressions ..... 45
Evaluation of Expressions ..... 46
Absolute and Relocatable Expressions ..... 47
Chapter 3. Program Structures and Addressing ..... 50
Object Program Structures ..... 50
Source Program Structures ..... 51
Source Module ..... 52
Beginning of a Source Module ..... 52
End of a Source Module ..... 52
Sections, Elements, and Parts ..... 52
Sections ..... 53
First Section ..... 54
Unnamed Section ..... 55
Reference Control Sections ..... 56
Dummy Control Sections ..... 56
Common Control Sections ..... 57
External Dummy Sections ..... 57
Classes (MVS and CMS) ..... 59
Class Binding and Loading Attributes ..... 60
Default Class Assignments ..... 60
Parts (MVS and CMS) ..... 61
Location Counter Setting ..... 61
Location Counter and Length Limits ..... 62
Use of Multiple Location Counters ..... 63
Addressing ..... 63
Addressing within Source Modules: Establishing Addressability ..... 64
How to Establish Addressability ..... 64
Base Register Instructions ..... 66
Qualified Addressing ..... 66
Dependent Addressing ..... 67
Relative Addressing ..... 67
Literal Pools ..... 68
Establishing Residence and Addressing Mode ..... 68
Symbolic Linkages ..... 68
Establishing symbolic linkage ..... 69
Referring to external data ..... 69
Branching to an external address ..... 69
Establishing an external symbol alias ..... 71
External Symbol Dictionary Entries ..... 72
Summary of Source and Object Program Structures ..... 73

## Chapter 1. Introduction

A computer can understand and interpret only machine language. Machine language is in binary form and, thus, very difficult to write. The assembler language is a symbolic programming language that you can use to code instructions instead of coding in machine language.

Because the assembler language lets you use meaningful symbols made up of alphabetic and numeric characters, instead of just the binary digits 0 and 1 used in machine language, you can make your coding easier to read, understand, and change. The assembler must translate the symbolic assembler language into machine language before the computer can run your program. The specific procedures followed to do this may vary according to the system you are using. However, the method is basically the same for all systems:


Figure 2. Assembling and Link-Editing Your Assembler Language Program
Your program, written in the assembler language, becomes the source module that is input to the assembler. The assembler processes your source module and produces an object module in machine language (called object code). The object module can be used as input to be processed by the linker or the binder. The linker or binder produces a load module (MVS and CMS), or a phase (VSE), that can be loaded later into the main storage of the computer. When your program is loaded, it can then be run. Your source module and the object code produced are printed, along with other information, on a program listing.

## 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 Assembler H Version 2. Similarly, 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 High Level Assembler symbolic operation codes
- SET symbols with names that conflict with the names of High Level Assembler system variable symbols
- Dependencies on the type attribute values of certain variable symbols or macro instruction operands

With the exception of these possible conflicts, and with the appropriate High Level Assembler option values, source 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.

VSE 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, and High Level Assembler requires that library macros be stored in source statement format. A library input exit can be used to analyze a macro definition, and, in the case of an E-Deck, call the VSE/ESA ESERV program to change, the E-Deck definition, line by line, back into source format required by the assembler, without modifying the original library file.

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 the exit and how to use it.

## Assembler Language

The assembler language is the symbolic programming language that lies closest to the machine language in form and content. You will, therefore, find the assembler language useful when:

- You need to control your program closely, down to the byte and even the bit level.
- You must write subroutines for functions that are not provided by other symbolic programming languages, such as COBOL, FORTRAN, or PL/I.

The assembler language is made up of statements that represent either instructions or comments. The instruction statements are the working part of the language and are divided into the following three groups:

- Machine instructions
- Assembler instructions
- Macro instructions


## Machine Instructions

A machine instruction is the symbolic representation of a machine language instruction of the following instruction sets:

- IBM System/370 ${ }^{\text {TM }}$
- IBM System/370 Extended Architecture (370-XA)
- Enterprise Systems Architecture/370 ${ }^{\text {TM }}$ (ESA/370)
- Enterprise Systems Architecture/390® (ESA/390)
- z/Architecture ${ }^{\text {TM }}$

It is called a machine instruction because the assembler translates it into the machine language code that the computer can run. Machine instructions are described in Chapter 4, "Machine Instruction Statements."

## Assembler Instructions

An assembler instruction is a request to the assembler to do certain operations during the assembly of a source module; for example, defining data constants, reserving storage areas, and defining the end of the source module. Except for the instructions that define constants, and the instruction used to generate no-operation instructions for alignment, the assembler does not translate assembler instructions into object code. The assembler instructions are described in Chapter 3, "Program Structures and Addressing," Chapter 5, "Assembler Instruction Statements," and Chapter 9, "How to Write Conditional Assembly Instructions."

## Macro Instructions

A macro instruction is a request to the assembler program to process a predefined sequence of instructions called a macro definition. From this definition, the assembler generates machine and assembler instructions, which it then processes as if they were part of the original input in the source module.

IBM supplies macro definitions for input/output, data management, and supervisor operations that you can call for processing by coding the required macro instruction. (These IBM-supplied macro instructions are described in the applicable Macro Instructions manual.)

You can also prepare your own macro definitions, and call them by coding the corresponding macro instructions. Rather than code all of this sequence each time it is needed, you can create a macro instruction to represent the sequence and then, each time the sequence is needed, simply code the macro instruction statement. During assembly, the sequence of instructions represented by the macro instruction is inserted into the source program.

A complete description of the macro facility, including the macro definition, the macro instruction, and the conditional assembly language, is given in Part 3, "Macro Language."

## Assembler Program

The assembler program, also referred to as the assembler, processes the machine, assembler, and macro instructions you have coded (source statements) in the assembler language, and produces an object module in machine language.

## Basic Functions

Processing involves the translation of source statements into machine language, assignment of storage locations to instructions and other elements of the program, and performance of auxiliary assembler functions you have designated. The output of the assembler program is the object program, a machine language translation of the source program. The assembler produces a printed listing of the source statements and object program statements and additional information, such as error messages, that are useful in analyzing the program. The object program is in the format required by the linker.

## Associated Data

The assembler can produce an associated data file that contains information about the source program and the assembly environment. The ADATA information includes information such as:

- Data sets used by the assembler
- Program source statements
- Macros used by the assembler
- Program symbols
- Program object code
- Assembly error messages

Different subsets of this information are needed by various consumers, such as configuration managers, debuggers, librarians, metrics collectors, and many more.

## Controlling the Assembly

You can control the way the assembler produces the output from an assembly, using assembler options and assembler language instructions.

Assembler options are described in the HLASM Programmer's Guide. A subset of assembler options can be specified in your source program using the *PROCESS statement described on page 102

Assembler language instructions are assembler language source statements that cause the assembler to perform a specific operation. Some assembler language instructions, such as the DC instruction, generate object code. Assembler language instructions are categorized as follows:

## Assembler Instructions

These include instructions for:

- Producing associated data
- Assigning base registers
- Defining data constants
- Controlling listing output
- Redefining operation codes
- Sectioning and linking programs
- Defining symbols

These instructions are described in Chapter 5, Assembler Instruction Statements

## Macro Instructions

These instructions let you define macros for generating a sequence of assembler language statements from a single instruction. These instructions are described in Part 3, Macro Language

## Conditional Assembly Instructions

These instructions let you perform general arithmetic and logical computations, and condition tests that can vary the output generated by the assembler. These instructions are described under "Conditional Assembly Instructions" on page 343 .

## Processing Sequence

The assembler processes the machine and assembler language instructions at different times during its processing sequence. You should be aware of the assembler's processing sequence in order to code your program correctly.

The assembler processes most instructions twice, first during conditional assembly and, later, at assembly time. However, as shown below, it does some processing only during conditional assembly.

Conditional Assembly and Macro Instructions: The assembler processes conditional assembly instructions and macro processing instructions during conditional assembly. During this processing the assembler evaluates arithmetic, logical, and character conditional assembly expressions. Conditional assembly takes place before assembly time.

The assembler processes the machine and ordinary assembler instructions generated from a macro definition called by a macro instruction at assembly time.

Machine Instructions: The assembler processes all machine instructions, and translates them into object code at assembly time.

Assembler Instructions: The assembler processes ordinary assembler instructions at assembly time. During this processing:

- The assembler evaluates absolute and relocatable expressions (sometimes called assembly-time expressions)
- Some instructions, such as ADATA, ALIAS, CATTR and XATTR (MVS and CMS), DC, DS, ENTRY, EXTRN, PUNCH, and REPRO, produce output for later processing by programs such as the linker.

The assembler prints in a program listing all the information it produces at the various processing times discussed above. The assembler also produces information for other processors. The linker uses such information at link-edit time to combine object modules into load modules. At program fetch time, the load module produced by the linker is loaded into virtual storage. Finally, at execution time, the computer runs the load module.

## Relationship of Assembler to Operating System

High Level Assembler operates under the OS/390 operating system, the MVS/ESA operating system, the CMS component of the VM/ESA® operating system, and the VSE/ESA ${ }^{\text {TM }}$ operating system. These operating systems provide the assembler with services for:

- Assembling a source module
- Running the assembled object module as a program

In writing a source module, you must include instructions that request any required service functions from the operating system.

MVS: MVS provides the following services:

- For assembling the source module:
- A control program
- Sequential data sets to contain source code
- Libraries to contain source code and macro definitions
- Utilities
- For preparing for the execution of the assembler program as represented by the object module:
- A control program
- Storage allocation
- Input and output facilities
- Linker or binder
- Loader

CMS: CMS provides the following services:

- For assembling the source module:
- An interactive control program
- Files to contain source code
- Libraries to contain source code and macro definitions
- Utilities
- For preparing for the execution of the assembler program as represented by the object modules:
- An interactive control program
- Storage allocation
- Input and output facilities
- Linker
- A loader

VSE: VSE provides the following services:

- For assembling the source module:
- A control program
- Sequential data sets to contain source code
- Libraries to contain source code and macro definitions
- Utilities
- For preparing for the execution of the assembler program as represented by the object module:
- A control program
- Storage allocation
- Input and output facilities
- Linker


## Coding Made Easier

It can be very difficult to write an assembler language program using only machine instructions. The assembler provides additional functions that make this task easier. They are summarized below.

## Symbolic Representation of Program Elements

Symbols greatly reduce programming effort and errors. You can define symbols to represent storage addresses, displacements, constants, registers, and almost any element that makes up the assembler language. These elements include operands, operand subfields, terms, and expressions. Symbols are easier to remember and code than numbers; moreover, they are listed in a symbol cross reference table, which is printed in the program listings. Thus, you can easily find a symbol when searching for an error in your code. See page 29 for details about symbols, and how you can use them in your program.

## Variety in Data Representation

You can use decimal, binary, hexadecimal, or character representation of machine language binary values in writing source statements. You select the representation best suited to the purpose. The assembler converts your representations into the binary values required by the machine language.

## Controlling Address Assignment

you code the correct assembler instruction, the assembler computes the relative offset, or displacement from a base address, of any symbolic addresses you specify in a machine instruction. It inserts this displacement, along with the base register assigned by the assembler instruction, into the object code of the machine instruction.

At execution time, the object code of address references must be in relative-immediate or base-displacement form. The computer obtains the required address by adding the displacement to the base address contained in the base register, or from the relative-immediate offset of the instruction.

## Relocatability

The assembler produces an object module that is independent of the location it is initially assigned in virtual storage. That is, it can be loaded into any suitable virtual storage area without affecting program execution. This is made easier because most addresses are assembled in their base-displacement form.

## Sectioning a Program

You can divide a source module into one or more control sections. After assembly, you can include or delete individual control sections from the resulting object module before you load it for execution. Control sections can be loaded separately into storage areas that are not contiguous. A discussion of sectioning is contained in "Source Program Structures" on page 51.

## Linkage between Source Modules

You can create symbolic linkages between separately assembled source modules. This lets you refer symbolically from one source module to data and instructions defined in another source module. You can also use symbolic addresses to branch between modules.

A discussion of sectioning and linking is contained in "Source Program Structures" on page 51

## Program Listings

The assembler produces a listing of your source module, including any generated statements, and the object code assembled from the source module. You can control the form and content of the listing using assembler listing control instructions, assembler options, and user I/O exits. The listing control instructions are described in Chapter 5, "Assembler Instruction Statements" on page 100 and in "Processing Statements" on page 256 Assembler options and user I/O exits are discussed in the HLASM Programmer's Guide.

The assembler also prints messages about actual errors and warnings about potential errors in your source module.

## Multiple Source Modules

The assembler can assemble more than one source module in a single input stream, if the BATCH option is specified. For more information about the BATCH option, see HLASM Programmer's Guide.

An "input stream" may contain one or more "source modules", and may also consist of one or more data sets if the host operating system supports data set or file concatenation. A "source module" is a single assembly.

## Chapter 2. Coding and Structure

This chapter provides information about assembler language coding conventions and assembler language structure.

## Character Set

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

## Standard Character Set

The standard (default) character set used by High Level Assembler is a subset of the EBCDIC character set. This subset consists of letters of the alphabet, national characters, the underscore character, digits, and special characters. The complete set of characters that make up the standard assembler language character set is shown in Figure 3.

Figure 3. Standard Character Set

| Alphabetic characters | a through z |
| :--- | :--- |
|  | A through Z |
|  | national characters @, \$, and \# |
|  | underscore character_ |
| Digits | 0 through 9 |
| Special characters | ,$+-=^{*}()^{\prime} / \&$ |
|  | space |

For a description of the binary and hexadecimal representations of the characters that make up the standard character set, see Appendix D, "Standard Character Set Code Table" on page 421 .

When you code terms and expressions (see "Terms, Literals, and Expressions" on page 28) in assembler language statements, you can only use the set of characters described above. However, when you code remarks, comments or character strings between paired single quotation marks, you can use any character in the EBCDIC character set.

The term alphanumeric characters includes both alphabetic characters and digits, but not special characters. Normally, you would use strings of alphanumeric characters to represent terms, and special characters as:

- Arithmetic operators in expressions
- Data or field delimiters
- Indicators to the assembler for specific handling

Whenever a lowercase letter (a through $z$ ) is used, the assembler considers it to be identical to the corresponding uppercase character (A through Z), except when it is used within a character string enclosed in single quotation marks, or within the positional and keyword operands of macro instructions.

Compatibility with Earlier Assemblers: You can specify the COMPAT(MACROCASE) assembler option to instruct the assembler to maintain uppercase alphabetic character set compatibility with earlier assemblers for unquoted macro operands. The assembler converts lowercase alphabetic characters (a through $z$ ) in unquoted macro operands to uppercase alphabetic characters (A through Z).

## Double-Byte Character Set

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

Figure 4. Double-Byte Character Set (DBCS)

| Double-byte space | X'4040' |
| :---: | :---: |
| Double-byte characters | Each double-byte character contains two bytes, each of which must be in the range $\mathrm{X}^{\prime} 41^{\prime}$ to $\mathrm{X}^{\prime}$ FE'. The first byte of a double-byte character is known as the ward byte. For example, the ward byte for the double-byte representation of EBCDIC characters is $\mathrm{X}^{\prime} 42^{\prime}$. |
| Shift codes | Shift-out (SO) - X'0E' <br> Shift-in (SI) - X'OF' |

## Note:

1. SO and SI delimit DBCS data only when the DBCS assembler option is specified. The DBCS assembler option is described in the HLASM Programmer's Guide.
2. When the DBCS assembler option is specified, double-byte characters may be used anywhere that EBCDIC characters enclosed by single quotation marks can be used.
3. Regardless of the invocation option, double-byte characters may be used in remarks, comments, and the statements processed by AREAD and REPRO statements.

Examples showing the use of EBCDIC characters and double-byte characters are given in Figure 5. For a description of the DBCS notation used in the examples, see "Double-Byte Character Set Notation" on page xvi]

| Figure 5 (Page | 1 | of 2 ). Examples Using | Character Set |
| :--- | :--- | :--- | :--- |
| Characters | Usage | Example | Constituting |
| Alphanumeric | In ordinary symbols | Label | Terms |
|  |  | FIELD\#01 |  |
|  | In variable symbols | Save_Total | \&EASY_TO_READ |

Figure 5 (Page 2 of 2). Examples Using Character Set

| Characters | Usage | Example | Constituting |
| :---: | :---: | :---: | :---: |
| Special |  |  |  |
| Characters | As operators |  |  |
| + | Addition | NINE+FIVE | Expressions |
| - | Subtraction | NINE-5 | Expressions |
| * | Multiplication | 9*FIVE | Expressions |
| / | Division | TEN/3 | Expressions |
| + or - | (Unary) | +NINE -FIVE | Terms ${ }^{1}$ |
|  | As delimiters |  |  |
| Spaces | Between fields | LABEL AR 3,4 | Statement |
| Comma | Between operands | OPND1,OPND2 | Operand field |
| Single Quotation Marks | Enclosing character strings | 'STRING' | String |
|  | Attribute operator | L'OPND1 | Term |
| Parentheses | Enclosing subfields or subexpressions | MOVE MVC TO(80), $\mathrm{FROM}(\mathrm{A}+\mathrm{B} *(\mathrm{C}-\mathrm{D})$ ) | Statement Expression |
| SO and SI | Enclosing double-byte data | $\begin{aligned} & C^{\prime}<. A . B . C>a b C^{\prime} \\ & \mathrm{G}^{\prime}<D 11 D 2 D 3 D 4>{ }^{\prime} \end{aligned}$ | Mixed string Pure DBCS |
|  | As indicators for |  |  |
| Ampersand | Variable symbol | \&VAR | Term |
| Period | Symbol qualifier | QUAL. SYMBOL | Term |
|  | Sequence symbol | . SEQ | (label) |
|  | Comment statement in macro definition | .*THIS IS A COMMENT | Statement |
|  | Concatenation | \&VAR.A | Term |
|  | Bit-length specification | DC CL. ${ }^{\prime}{ }^{\prime} \mathrm{AB}^{\prime}$ | Operand |
|  | Decimal point | DC F'1.7E4' | Operand |
| Asterisk | Location counter reference | *+72 | Expression |
|  | Comment statement | *THIS IS A COMMENT | Operand |
| Equal sign | Literal reference | L 6,=F'2' | Operand |
|  | Keyword | \& $K E Y=D$ | Keyword parameter |

## Note:

1. If these are passed as macro arguments, they are treated as expressions, not terms. Expressions cannot be substituted into SETA expressions.

## 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. You can supply a translation table using the TRANSLATE assembler option described in the HLASM Programmer's Guide

Self-defining Terms: Self-defining terms are not translated when a translation table is used. See "How to Generate a Translation Table" in the HLASM Programmer's Guide.

## Assembler Language Coding Conventions

Figure 6 shows the standard format used to code an assembler language statement.


Figure 6. Standard Assembler Coding Format

## Field Boundaries

Assembler language statements usually occupy one 80-character record, or line. For information about statements that occupy more than 80 characters, see "Continuation Lines" on page 15. Each line is divided into three main fields:

- Statement field
- Continuation-indicator field
- Identification-sequence field

If it can be printed, any character coded into any column of a line, or otherwise entered as a position in a source statement, is reproduced in the listing printed by the assembler. Whether it can be printed or not depends on the printer.

Uppercase Printing: Use the FOLD assembler option to instruct the assembler to convert lowercase alphabetic characters to uppercase alphabetic characters before they are printed.

## Statement Field

The instructions and comment statements must be written in the statement field. The statement field starts in the begin column and ends in the end column. The continuation-indicator field always lies in the column after the end column, unless the end column is column 80 , in which case no continuation is possible. The identification-sequence field usually lies in the field after the continuation-indicator field. Any continuation lines needed must start in the continue column and end in the end column.

Blank lines are acceptable. For more information, see "Blank Lines" on page 18

The assembler assumes the following standard values for these columns:

- The begin column is column 1
- The end column is column 71
- The continue column is column 16

These standard values can be changed by using the Input Format Control (ICTL) assembler instruction. The ICTL instruction can, for example, be used to reverse the order of the statement field and the identification-sequence field by changing the standard begin, end, and continue columns. However, all references to the begin, end, and continue columns in this manual refer to the standard values described above.

## Continuation-Indicator Field

The continuation-indicator field occupies the column after the end column. Therefore, the standard position for this field is column 72. A non-space character in this column indicates that the current statement is continued on the next line. This column must be a space character on the last (or only) line of a statement. If this column is not a space, the assembler treats the statement that follows on the next line as a continuation line of the current statement.

If the DBCS assembler option is specified, then:

- When an SI is placed in the end column of a continued line, and an SO is placed in the continue column of the next line, the SI and SO are considered redundant and are removed from the statement before statement analysis is done.
- 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.
- The double-byte delimiters SO and SI cannot be used as continuation-indicators.


## Identification-Sequence Field

The identification-sequence field can contain identification characters or sequence numbers or both. If the ISEQ instruction has been specified to check this field, the assembler verifies whether or not the source statements are in the correct sequence.

The columns checked by the ISEQ function are not restricted to columns 73 through 80, or by the boundaries determined by any ICTL instruction. The columns specified in the ISEQ instruction can be anywhere on the input statement, including columns that are occupied by the statement field.

## Continuation Lines

To continue a statement on another line, follow these rules:

1. Enter a non-space character in the continuation-indicator field (column 72). This non-space character must not be part of the statement coding. When more than one continuation line is needed, enter a non-space character in column 72 of each line that is to be continued.
2. Continue the statement on the next line, starting in the continue column (column 16). Columns to the left of the continue column must be spaces. Comment statements may be continued after column 16.

If an operand is continued after column 16, it is taken to be a comment. Also, if the continuation-indicator field is filled in on one line and you try to start a new statement after column 16 on the next line, this statement is taken as a comment belonging to the previous statement.

Specify the FLAG(CONT) assembler option to instruct the assembler to issue warning messages when it suspects a continuation error in a macro call instruction. Refer to the FLAG option description in the HLASM Programmer's Guide for details about the situations that might be flagged as continuation errors.

Unless it is one of the statement types listed below, nine continuation lines are allowed for a single assembler language statement.

## Alternative Statement Format

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

- AGO conditional assembly statement, see "Alternative Format for AGO Instruction" on page 393
- AIF conditional assembly statement, see "Alternative Format for AIF Instruction" on page 392
- GBLA, GBLB, and GBLC conditional assembly statements, see "Alternative Format for GBLx Statements" on page 345
- LCLA, LCLB, and LCLC conditional assembly statements, see "Alternative Format for LCLx Statements" on page 347
- Macro instruction statement, see "Alternative Formats for a Macro Instruction" On page 298
- Prototype statement of a macro definition, see "Alternative Formats for the Prototype Statement" on page 246
- SETA, SETB, SETAF, SETCF and SETC conditional assembly statements, see "Alternative Statement Format" on page 388

Examples of the alternative statement format for each of these instructions are given with the description of the individual instruction.

## Continuation of double-byte data

No special considerations apply to continuation:

- Where double-byte data is created by a code-generation program, and
- There is no requirement for double-byte data to be readable on a device capable of presenting DBCS characters

A double-byte character string may be continued at any point, and SO and SI must be balanced within a field, but not within a statement line.

Where double-byte data is created by a workstation that has the capability of presenting DBCS characters, such as the IBM 5550 multistation, or where readability of double-byte data in High Level Assembler source input or listings is required, special features of the High Level Assembler language may be used. When the DBCS assembler option is specified, High Level Assembler provides the flexibility to cater for any combination of double-byte data and single-byte data. The special features provided are:

- Removal of redundant SI/SO at continuation points. When an SI is placed in the end column of a continued line, and an SO is placed in the continue
column of the next line, the SI and SO are considered redundant and are removed from the statement before statement analysis.
- An extended continuation-indicator provides a flexible end column on a line-by-line basis to support any alignment of double-byte data in a source statement. The end column of continued lines may be shifted to the left by extending the continuation-indicator.
- To guard against accidental continuation caused by double-byte data ending in the continuation-indicator column, neither SO nor SI is regarded as a continuation-indicator. If either is used, the following warning message is issued:

```
ASMA201W S0 or SI in continuation column - no continuation
assumed
```

The examples below show the use of these features. Refer to "Double-Byte Character Set Notation" on page xvil for the notation used in the examples.

## Source Input Considerations

- Extended continuation-indicators may be used in any source statement, including macro statements and statements included by the COPY instruction. This feature is intended for source lines containing double-byte data, however it becomes available to all lines when the DBCS option is set.
- On a line with a non-space continuation-indicator, the end column is the first column to the left of the continuation-indicator which has a value different from the continuation-indicator.
- When converting existing programs for assembly with the DBCS option, ensure that continuation-indicators are different from the adjacent data in the end column.
- The extended continuation-indicators must not be extended into the continue column, otherwise the extended continuation-indicators are treated as data, and the assembler issues the following error message:

```
ASMA205E Extended continuation column must not extend into continue
column
```

- For SI and SO to be removed at continuation points, the SI must be in the end column, and the SO must be in the continue column of the next line.


## Examples:

| Name | Operation | Operand |
| :--- | :--- | :--- |
| DBCS1 | DC | $C^{\prime}<D 1 D 2 D 3 D 4 D 5 D 6 D 7 D 8 D 9>X X X X X X X X X X X X X X X X X X X X$ <br> $<D a D b>1$ |
| DBCS2 | DC | $C^{\prime} a b c d e f g h i j k 1 m n o p q r s t u v w x y z 0123456789 X X X X ~$ <br> $<D a D b>1$ |
| DBCS3 | DC | $C^{\prime} a b c d e f g h i j k 1 m n o p q r s t u v<D 1 D 2 D 3 D 4 D 5 D 6 D 7>X X ~$ <br> $<D a D b>1$ |

DBCS1: The DBCS1 constant contains 11 double-byte characters bracketed by SO and SI. The SI and SO at the continuation point are not assembled into the operand. The assembled value of DBCS1 is:
<D1D2D3D4D5D6D7D8D9DaDb>

DBCS2: The DBCS2 constant contains an EBCDIC string which is followed by a double-byte string. Because there is no space for any double-byte data on the first line, the end column is extended three columns to the left and the double-byte data started on the next line. The assembled value of DBCS2 is:
abcdefghijk1mnopqrstuvwxyz0123456789<DaDb>
DBCS3: The DBCS3 constant contains 22 EBCDIC characters followed by 9 double-byte characters. Alignment of the double-byte data requires that the end column be extended one column to the left. The SI and SO at the continuation point are not assembled into the operand. The assembled value of DBCS3 is:
abcdefghijk1mnopqrstuv<D1D2D3D4D5D6D7DaDb>

## Source Listing Considerations

- For source that does not contain substituted variable symbols, the listing exactly reflects the source input.
- Double-byte data input from code-generation programs, that contain no substituted variables, are not readable in the listing if the source input was not displayable on a device capable of presenting DBCS characters.
- Refer to "Listing of Generated Fields Containing Double-Byte Data" on page 249 for details of extended continuation and macro-generated statements.


## Blank Lines

Blank lines are accepted in source programs. 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 Statement Format

Comment statements are not assembled as part of the object module, but are only printed in the assembly listing. You can write as many comment statements as you need, provided you follow these rules:

- Comment statements require an asterisk in the begin column. Internal macro definition comment statements require a period in the begin column, followed by an asterisk. Internal macro comments are accepted as comment statements in open code.
- Any characters of the EBCDIC character set, or double-byte character set can be used (see "Character Set" on page 11).
- Comment statements must lie within the statement field. If the comment extends into the continuation-indicator field, the statement following the comment statement is considered a continuation line of that comment statement.
- Comment statements must not appear between an instruction statement and its continuation lines.


## Instruction Statement Format

Instruction statements must consist of one to four entries in the statement field. They are:

- A name entry
- An operation entry
- An operand entry
- A remarks entry

These entries must be separated by one or more spaces, and must be written in the order stated.

## Statement Coding Rules

The following general rules apply to the coding of an instruction statement:

- The entries must be written in the following order: name, operation, operand, and remarks.
- The entries must be contained in the begin column (1) through the end column (71) of the first line and, if needed, in the continue column (16) through the end column (71) of any continuation lines.
- The entries must be separated from each other by one or more spaces.
- If used, a name entry must start in the begin column.
- The name and operation entries, each followed by at least one space, must be contained in the first line of an instruction statement.
- The operation entry must begin at least one column to the right of the begin column.

Statement Example: The following example shows the use of name, operation, operand, and remarks entries. The symbol COMP names a compare instruction, the operation entry (CR) is the mnemonic operation code for a register-to-register compare operation, and the two operands $(5,6)$ designate the two general registers whose contents are to be compared. The remarks entry reminds readers that this instruction compares NEW SUM to OLD.
COMP CR 5,6 NEW SUM TO OLD
Descriptions of the name, operation, operand, and remarks entries follow:
Name Entry: The name entry is a symbol created by you to identify an instruction statement. A name entry is usually optional. Except for two instances, the name entry, when provided, must be a valid symbol at assembly time (after substituting variable symbols, if specified). For a discussion of the exceptions to this rule, see "TITLE Instruction" on page 215 and "Macro Instruction Format" on page 297

The symbol must consist of 63 or fewer alphanumeric characters, the first of which must be alphabetic. It must be entered with the first character appearing in the begin column. If the begin column is a space, the assembler program assumes no name has been entered. No spaces or double-byte data may appear in the symbol.

Operation Entry: The operation entry is the symbolic operation code specifying the machine, assembler, or macro instruction operation. The following rules apply to the operation entry:

## Assembler Language Coding Conventions

- An operation entry is mandatory, and it must appear on the same line as any name entry.
- For machine and assembler instructions, it must be a valid symbol at assembly time (after substitution for variable symbols, if specified), consisting of 63 or fewer alphanumeric characters, the first which must be alphabetic. Most standard symbolic operation codes are five characters or fewer. For a description of machine instructions, see the applicable Principles of Operation manual. For a summary of assembler instructions, see Appendix A, "Assembler Instructions."

The standard set of codes can be changed by OPSYN instructions (see "OPSYN Instruction" on page 198.

- For macro instructions, the operation entry can be any valid symbol.
- An operation entry cannot be continued on the next statement.

Operand Entries: Operand entries contain zero or more operands that identify and describe data to be acted upon by the instruction, by indicating such information as storage locations, masks, storage area lengths, or types of data. The following rules apply to operands:

- One or more operands are usually required, depending on the instruction.
- Operands must be separated by commas. No spaces are allowed between the operands and the commas that separate them.
- A space normally indicates the end of the operand entry, unless the operand is in single quotes. This applies to machine, assembler, and macro instructions.
- A space does not end the operand in some types of SET statement. Spaces that do not end operands are discussed further at:
- "Arithmetic (SETA) Expressions" on page 352
- "Logical (SETB) Expressions" on page 365
- "Character (SETC) Expressions" on page 371

There are two examples of operands containing spaces in Figure 10 on page 26; the last box in Row 3, and the middle box in Row 4.

- The alternative statement format uses slightly different rules. For more information, see "Alternative Formats for a Macro Instruction" on page 298

The following instruction is correctly coded:
LA
R1,4+5
No space

The following instruction may appear to be the same, but is not:

$$
\text { LA } \quad \text { R1,4 + } 5 \quad \text { Spaces included }
$$

In this example, the embedded space means that the operand finishes after "4." There is no assembler error, but the result is a LA R1,4, which may not be what you intended.

A space inside unquoted parentheses is an error, and leads to a diagnostic. The following instruction is correctly coded:

> DC CL(L'STRLEN)' ' Space within quotes

The following instruction, with an extra space, is not correct:

DC CL(L'STRLEN )' ' Space not within quotes
The following example shows a space enclosed in quotes, as part of a string. This space is properly accounted for:

MVC
AREA1,=C'This Area' Space inside quotes
In quotes, spaces and parentheses can occur in any quantity and in any order:

> LA
R1,=C'This is OK (isn''t it)'

Remarks Entries: Remarks are used to describe the current instruction. The following rules apply to remarks:

- Remarks are optional.
- They can contain any character from the EBCDIC character set, or the double-byte characters set.
- They can follow any operand entry.
- In statements in which an optional operand entry is omitted, but you want to code a comment, indicate the absence of the operand by a comma preceded and followed by one or more spaces, as shown below:

```
End of Program
```


## Assembler Language Structure

This section describes the structure of the assembler language, including the statements that are allowed in the language, and the elements that make up those statements.
"Statement Coding Rules" on page 19 describes the composition of an assembler language source statement.

The figures in this section show the overall structure of the statements that represent the assembler language instructions, and are not specifications for these instructions. The individual instructions, their purposes, and their specifications are described in other sections of this manual.

Model statements, used to generate assembler language statements, are described in Chapter 7, "How to Specify Macro Definitions."

The remarks entry in a source statement is not processed by the assembler, but it is printed in the assembler listing. For this reason, it is only shown in the overview of the assembler language structure in Figure 7 on page 23 , and not in the other figures.

The machine instruction statements are described in Figure 8 on page 24 , discussed in Chapter 4, "Machine Instruction Statements,"] and summarized in the applicable Principles of Operation manual.

Assembler instruction statements are described in Figure 9 on page 25, discussed in Chapter 3, "Program Structures and Addressing" and Chapter 5, "Assembler Instruction Statements," and are summarized in Appendix A, "Assembler Instructions."

## Assembler Language Structure

Conditional assembly instruction statements and the macro processing statements (MACRO, MEND, MEXIT, MNOTE, AREAD, ASPACE, and AEJECT) are described in Figure 10 on page 26 The conditional assembly instructions are discussed in Chapter 9, "How to Write Conditional Assembly Instructions," and macro processing instructions in Chapter 7, "How to Specify Macro Definitions." Both types are summarized in|Appendix A, "Assembler Instructions."

Macro instruction statements are described in Figure 11 on page 27, and discussed in Chapter 8, "How to Write Macro Instructions" on page 297.

## Overview of Assembler Language Structure



Figure 7. Overview of Assembler Language Structure

## Assembler Language Structure

## Machine Instructions


${ }^{1}$ Can be an ordinary symbol, a variable symbol, or a sequence symbol
2 With DBCS option only
Figure 8. Machine Instructions

## Assembler Instructions



Figure 9. Ordinary Assembler Instruction Statements

## Assembler Language Structure

## Conditional Assembly Instructions



Figure 10. Conditional Assembly Instructions
Macro instruction statements are described in Figure 11 on page 27.

## Macro Instructions



Figure 11. Macro Instructions

## Terms, Literals, and Expressions

The most basic element of the assembler language is the term. Terms may be used alone, or in combination with other terms in expressions. This section describes the different types of terms used in the assembler language, and how they can be used.

## Terms

A term is the smallest element of the assembler language that represents a distinct and separate value. It can, therefore, be used alone or in combination with other terms to form expressions. Terms are classified as absolute or relocatable, depending on the effect of program relocation upon them. Program relocation is the loading of the object program into storage locations other than those originally assigned by the assembler. Terms have absolute or relocatable values that are assigned by the assembler or that are inherent in the terms themselves.

A term is absolute if its value does not change upon program relocation. A term is relocatable if its value changes by $n$ if the origin of the control section in which it appears is relocated by $n$ bytes.

Terms in Parentheses: Terms in parentheses are reduced to a single value; thus the terms in parentheses, in effect, become a single term.

You can use arithmetically combined terms, enclosed in parentheses, in combination with terms outside the parentheses, as follows:
14+BETA- (GAMMA-LAMBDA)
When the assembler encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value which may be absolute or relocatable, depending on the combination of terms. This value is then used in reducing the rest of the combination to another single value.

You can include terms in parentheses within a set of terms in parentheses:
$A+B-(C+D-(E+F)+10)$
The innermost set of terms in parentheses is evaluated first. Any number of levels of parentheses are allowed. A level of parentheses is a left parenthesis and its corresponding right parenthesis. An arithmetic combination of terms is evaluated as described in "Expressions" on page 44. Figure 12 summarizes the various types of terms, and gives a reference to the page number that discusses the term and the rules for using it.

Figure 12 (Page 1 of 2). Summary of Terms

|  | Term <br> can be <br> absolute | Term can <br> be <br> relocatable | Value is <br> assigned <br> by <br> assembler | Value is <br> inherent <br> in term | Page <br> reference |
| :--- | :--- | :--- | :--- | :--- | :--- |
| Symbols | $X$ | $X$ | $X$ | 29 |  |
| Literals | $X$ | $X$ | $X$ |  | 41 |
| Self-defining terms | $X$ |  |  | $X$ | 34 |

Figure 12 (Page 2 of 2). Summary of Terms

|  | Term <br> can be <br> absolute | Term can <br> be <br> relocatable | Value is <br> assigned <br> by <br> assembler | Value is <br> inherent <br> in term | Page <br> reference |
| :--- | :--- | :--- | :--- | :--- | :--- |
| Terms | X | X | 36 |  |  |
| Location counter <br> reference | X | X | 38 |  |  |
| Symbol length <br> attribute | X | X | 40 |  |  |
| Other data <br> attributes |  |  |  |  |  |

## Notes:

1. Other valid data attributes are $S$ and $I$.

For more information about absolute and relocatable expressions, see "Absolute and Relocatable Expressions" on page 47

## Symbols

You can use a symbol to represent storage locations or arbitrary values. If you write a symbol in the name field of an instruction, you can then specify this symbol in the operands of other instructions and thus refer to the former instruction symbolically. This symbol represents a relocatable address.

You can also assign an absolute value to a symbol by coding it in the name field of an EQU instruction with an operand whose value is absolute. This lets you use this symbol in instruction operands to represent:

- Registers
- Displacements in explicit addresses
- Immediate data
- Lengths
- Implicit addresses with absolute values

For details of these program elements, see "Operand Entries" on page 85
The advantages of symbolic over numeric representation are:

- Symbols are easier to remember and use than numeric values, thus reducing programming errors and increasing programming efficiency.
- You can use meaningful symbols to describe the program elements they represent. For example, INPUT can name a field that is to contain input data, or INDEX can name a register to be used for indexing.
- You can change the value of one symbol that is used in many instructions (through an EQU instruction) more easily than you can change several numeric values in many instructions.
- If the symbols are relocatable, the assembler can calculate displacements and assign base registers for you.
- Symbols are entered into a cross reference table that is printed in the Ordinary Symbol and Literal Cross Reference section of the assembler listing. The cross reference helps you find a symbol in the source and object section of the listing because it shows:
- The number of the statement that defines the symbol. A symbol is defined when it appears in the name entry of a statement.
- The number of all the statements in which the symbol is used as an operand.

Symbol Table: When the assembler processes your source statements for the first time, it assigns an absolute or relocatable value to every symbol that appears in the name field of an instruction. The assembler enters this value, which normally reflects the setting of the location counter, into the symbol table. It also enters the attributes associated with the data represented by the symbol. The values of the symbol and its attributes are available later when the assembler finds this symbol or attribute reference used as a term in an operand or expression. See Symbol Length Attribute Reference' and 'Self-Defining Terms' in this chapter for more details. The three types of symbols recognized by the assembler are:

- Ordinary symbols
- Variable symbols
- Sequence symbols

Ordinary Symbols: Ordinary symbols can be used in the name and operand fields of machine and assembler instruction statements. Code them to conform to these rules:

- The symbol must not consist of more than 63 alphanumeric characters. The first character must be an alphabetic character. An alphabetic character is a letter from A through $Z$, or from a through $z$, or \$, _, \#, or @. The other characters in the symbol may be alphabetic characters, digits, or a combination of the two.
- The assembler makes no distinction between upper-case and lower-case letters used in symbols.
- If the GOFF option is not specified, external symbols may not consist of more than 8 characters.
- No other special characters may be included in an ordinary symbol.
- No spaces are allowed in an ordinary symbol.
- No double-byte data is allowed in an ordinary symbol.

In the following sections, the term symbol refers to the ordinary symbol.
The following examples are valid ordinary symbols:

| ORDSYM\#435A | HERE | \$OPEN |
| :--- | :--- | :--- |
| K4 | \#0123 | X |
| B49467LITTLENAIL | 033 | _TOTAL_SAVED |

Variable Symbols: Variable symbols must begin with an \& followed by an alphabetic character and, optionally, up to 61 alphanumeric characters. Variable symbols can be used in macro processing and conditional assembly instructions, and to provide substitution in machine and assembler instructions. They allow different values to be assigned to one symbol. A complete discussion of variable symbols appears in Chapter 7, "How to Specify Macro Definitions" on page 243.

The following examples are valid variable symbols:

| \&VARYINGSYMABC | \&@ME |
| :--- | :--- |
| \&F346944 | $\& A$ |
| \&EASY_TO_READ |  |

System Variable Symbol Prefix: A variable symbol should not begin with the characters \&SYS as they are used to prefix System Variable Symbols. See "System Variable Symbols" on page 262 for a list of the System Variable Symbols provided with High Level Assembler.

Sequence Symbols: Sequence symbols consist of a period (.) followed by an alphabetic character, and up to 61 additional alphanumeric characters. Sequence symbols can be used in macro processing and conditional assembly instructions. They indicate the position of statements within the source program or macro definition. They are used in AIF and AGO statements to vary the sequence in which statements are processed by the assembler program. (See the complete discussion in Chapter 9, "How to Write Conditional Assembly Instructions.")

The following examples are valid sequence symbols:

```
.BLABEL04 .#359
.BRANCHTOMEFIRST .A
```

Symbol Definition: An ordinary symbol is defined in:

- The name entry in a machine or assembler instruction of the assembler language
- One of the operands of an EXTRN or WXTRN instruction

Ordinary symbols can also be defined in instructions generated from model statements during conditional assembly.

In Figure 13 on page 32, the assembler assigns a value to the ordinary symbol in the name entry according to the following rules:

1. The symbol is assigned a relocatable address value if the first byte of the storage field contains one of the following:

- Any machine or assembler instruction, except the EQU or OPSYN instruction (see 1 in Figure 13)
- A storage area defined by the DS instruction (see 2 in Figure 13)
- Any constant defined by the DC instruction (see $\mathbf{3}$ in Figure 13)
- A channel command word defined by the CCW, CCW0, or CCW1 instruction
The address value assigned is relocatable, because the object code assembled from these items is relocatable. The relocatability of addresses is described in "Addresses" on page 87

2. The symbol is assigned the value of the first or only expression specified in the operand of an EQU instruction. This expression can have a relocatable (see 4 in Figure 13) or absolute (see 5 in Figure 13) value, which is then assigned to the ordinary symbol.

The value of an ordinary absolute symbol must lie in the range $-2^{31}$ through $+2^{31}-1$. Relocatable symbols have unsigned address values in the range from 0 to $2^{24}-1$, or 0 to $2^{31}-1$ if the GOFF option is specified.


Figure 13. Transition from Assembler Language Statement to Object Code
Restrictions on Symbols: A symbol must be defined only once in a source module with one or more control sections, with the following exceptions:

- The symbol in the name field of a CSECT, RSECT, DSECT, or COM instruction can be the same as the name of previous CSECT, RSECT, DSECT, or COM instruction, respectively. It identifies the resumption of the control section specified by the name field.
- CMS, MVS The symbol in the name field of a CATTR instruction can be the same as the name of a previous CATTR instruction. It identifies the resumption of the class specified by the name field. $\qquad$
- The symbol in the name field of a LOCTR instruction can be the same as the name of a previous START, CSECT, RSECT, DSECT, COM, or LOCTR instruction. It identifies the resumption of the location counter specified by the name field.
- The symbol in the name field of a labeled USING instruction can be the same as the name of a previous labeled USING instruction. It identifies the termination of the domain of the previous labeled USING instruction with the specified name.
- A symbol can be used as an operand of a V-type constant and as an ordinary label, without duplication, because the operand of a V-type constant does not define the symbol in the symbol table.

An ordinary symbol is not defined when:

- It is used in the name field of an OPSYN or TITLE instruction. It can, therefore, be used in the name field of any other statement in a source module.
- It is used as the operand of a V-type address constant.
- It is only used in the name field of a macro instruction and does not appear in the name field of a macro-generated assembler statement. It can, therefore, be used in the name field of any other statement in a source module.
- It is only used in the name field of an ALIAS instruction and does not appear in one of the following:
- The name field of a START, CSECT, RSECT, COM, or DXD instruction.
- The name field of a DSECT instruction and the nominal value of a Q-type address constant.
- The operand of an ENTRY, EXTRN or WXTRN instruction.

Previously Defined Symbols: An ordinary symbol is previously defined if the statement that defines it is processed before the statement in which the symbol appears in an operand.

An ordinary symbol must be defined by the time the END statement is reached, however, it need not be previously defined when it is used as follows:

- In operand expressions of certain instructions such as CNOP instructions and some ORG instructions
- In modifier expressions of DC, DS, and DXD instructions
- In the first operand of an EQU instruction
- In Q-type constants

When using the forward-reference capability of the assembler, avoid the following types of errors:

- Circular definition of symbols, such as:

| $X$ | EQU | $Y$ |
| :--- | :--- | :--- |
| $Y$ | EQU | $X$ |

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

| A | DS | $(B-A) C$ |
| :--- | :--- | :--- |
| $B$ | $L R$ | 1,2 |

The first statement in this example cannot be resolved because the value of the duplication factor is dependent on the location of $B$, which is, in turn, dependent upon the length and duplication factor of A .

Literals may contain symbolic expressions in modifiers, but any ordinary symbols used must have been previously defined.

## Self-Defining Terms

A self-defining term lets you specify a value explicitly. With self-defining terms, you can also specify decimal, binary, hexadecimal, or character data. If the DBCS assembler option is specified, you can specify a graphic self-defining term that contains pure double-byte data, or include double-byte data in character self-defining terms. These terms have absolute values and can be used as absolute terms in expressions to represent bit configurations, absolute addresses, displacements, length or other modifiers, or duplication factors.

Using Self-Defining Terms: Self-defining terms represent machine language binary values and are absolute terms. Their values do not change upon program relocation. Some examples of self-defining terms and the binary values they represent are given below:

| Self-Defining Term | Decimal Value | Binary Value |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 15 | 15 |  |  |  | 1111 |
| 241 | 241 |  |  | 1111 | 0001 |
| B'1111' | 15 |  |  |  | 1111 |
| B'11110001' | 241 |  |  | 1111 | 0001 |
| B'100000001' | 257 |  | 0001 |  | 0001 |
| $X^{\prime} \mathrm{F}^{\prime}$ | 15 |  |  |  | 1111 |
| X'F1' | 241 |  |  | 1111 | 0001 |
| X'101' | 257 |  | 0001 |  | 0001 |
| $C^{\prime} 1^{\prime}$ | 241 |  |  | 1111 | 0001 |
| $C^{\prime} A^{\prime}$ | 193 |  |  | 1100 | 0001 |
| $C^{\prime}{ }^{\prime} \mathrm{BB}^{\prime}$ | 49,602 | 1100 | 0001 | 1100 | 0010 |
| $\mathrm{G}^{\prime}<.{ }^{\prime}{ }^{\prime}$ | 17,089 | 0100 | 0010 | 1100 | 0001 |

The assembler carries the values represented by self-defining terms to 4 bytes or 32 bits, the high-order bit of which is the sign bit. ( $\mathrm{A}^{\prime} 1{ }^{\prime}$ in the sign bit indicates a negative value; a ' 0 ' indicates a positive value.)

The use of a self-defining term is distinct from the use of data constants or literals. When you use a self-defining term in a machine instruction statement, its value is used to determine the binary value that is assembled into the instruction. When a data constant is referred to or a literal is specified in the operand of an instruction, its address is assembled into the instruction. Self-defining terms are always right-justified. Truncation or padding with zeros, if necessary, occurs on the left.

Decimal Self-Defining Term: A decimal self-defining term is simply an unsigned decimal number written as a sequence of decimal digits. High-order zeros may be used (for example, 007). Limitations on the value of the term depend on its use. For example, a decimal term that designates a general register should have a value between 0 and 15. A decimal term that represents an address should not exceed the size of storage. In any case, a decimal term may not consist of more than 10 digits, nor exceed $2,147,483,647\left(2^{31}-1\right)$. A decimal self-defining term is assembled as its binary equivalent. Some examples of decimal self-defining terms are: $8,147,4092$, and 00021.

Hexadecimal Self-Defining Term: A hexadecimal self-defining term consists of 1 -to-8 hexadecimal digits enclosed in single quotation marks and preceded by the letter X; for example, X'C49 '.

Each hexadecimal digit is assembled as its 4-bit binary equivalent. Thus, a hexadecimal term used to represent an 8-bit mask would consist of 2 hexadecimal digits. The maximum value of a hexadecimal term is X'FFFFFFFF' $^{\prime}$; this allows a range of values from $-2,147,483,648$ through $2,147,483,647$.

The hexadecimal digits and their bit patterns are as follows:

| $0-0000$ | $4-0100$ | $8-1000$ | C -1100 |
| :--- | :--- | :--- | :--- |
| $1-0001$ | $5-0101$ | $9-1001$ | D -1101 |
| $2-0010$ | $6-0110$ | A -1010 | E -1110 |
| $3-0011$ | $7-0111$ | B -1011 | F -1111 |

When used as an absolute term in an expression, a hexadecimal self-defining term has a negative value if the high-order bit is 1 .

Binary Self-Defining Term: A binary self-defining term is written as an unsigned sequence of 1 s and 0 s enclosed in single quotation marks and preceded by the letter B; for example, $\mathrm{B}^{\prime} 10001101^{\prime}$. A binary term may have up to 32 bits. This allows a range of values from $-2,147,483,648$ through $2,147,483,647$.

When used as an absolute term in an expression, a binary self-defining term has a negative value if the term is 32 bits long and the high-order bit is 1 .

Binary representation is used primarily in designating bit patterns of masks or in logical operations.

The following shows a binary term used as a mask in a Test Under Mask (TM) instruction. The contents of GAMMA are to be tested, bit by bit, against the pattern of bits represented by the binary term.
ALPHA TM GAMMA, B'10101101'
Character Self-Defining Term: A character self-defining term consists of 1-to-4 characters enclosed in single quotation marks, and must be preceded by the letter C. All letters, decimal digits, and special characters may be used in a character self-defining term. In addition, any of the remaining EBCDIC characters may be designated in a character self-defining term. Examples of character self-defining terms are:

```
C'/'
C' ' (space)
C'ABC'
C'13'
```

Because of the use of single quotation marks in the assembler language and ampersands in the macro language as syntactic characters, the following rule must be observed when using these characters in a character self-defining term:

For each single quotation mark or ampersand you want in a character self-defining term, two single quotation marks or ampersands must be written.
For example, the character value A' \# would be written as 'A' '\#', while a single quotation mark followed by a space and another single quotation mark would be written as ''r ''

For C-type character self-defining terms, each character in the character sequence is assembled as its 8 -bit code equivalent (see Appendix D, "Standard Character Set Code Table" on page 421. The two single quotation marks or ampersands that must be used to represent a single quotation mark or ampersand within the character sequence are assembled as a single quotation mark or ampersand. Double-byte data may appear in a character self-defining term, if the DBCS assembler option is specified. The assembled value includes the SO and SI delimiters. Hence a character self-defining term containing double-byte data is limited to one double-byte character delimited by SO and SI. For example, $C^{\prime}<. A>1$.

Since the SO and SI are stored, the null double-byte character string, C' ${ }^{\prime}$ ' , is also a valid character self-defining term.
Note: The assembler will not support character self-defining terms of the form CU'x' because self-defining terms are required by definition of the Assembler Language to have fixed values.

Graphic Self-Defining Term: If the DBCS assembler option is specified, a graphic self-defining term can be specified. A graphic self-defining term consists of 1 or 2 double-byte characters delimited by SO and SI, enclosed in single quotation marks and preceded by the letter G. Any valid double-byte characters may be used. Examples of graphic self-defining terms are:

```
G'<.A>'
G'<.A.B>'
G'<Da>'
G'<.A><.B>'
```

The SO and SI are not represented in the assembled value of the self-defining term, hence the assembled value is pure double-byte data. A redundant SI/SO pair can be present between two double-byte characters, as shown in the last of the above examples. However, if SO and SI are used without an intervening double-byte character, this error is issued:

ASMA148E Self-defining term lacks ending quote or has bad character

## Location Counter

The assembler maintains a location counter to assign storage addresses to your program statements. It is the assembler's equivalent of the execution-time instruction counter in the computer. You can refer to the current value of the location counter at any place in a source module by specifying an asterisk (*) as a term in an operand.

As the instructions and constants of a source module are being assembled, the location counter has a value that indicates a location in the program. The assembler increments the location counter according to the following:

1. After an instruction or constant has been assembled, the location counter indicates the next available location.
2. Before assembling the current instruction or constant, the assembler checks the boundary alignment required for it and adjusts the location counter, if necessary, to the correct boundary.
3. While the instruction or constant is being assembled, the location counter value does not change. It indicates the location of the current data after boundary
alignment and is the value assigned to the symbol, if present, in the name field of the statement.
4. After assembling the instruction or constant, the assembler increments the location counter by the length of the assembled data to indicate the next available location.

These rules are shown below:

| Location in <br> Hexadecimal |  | Source <br> Statements |
| :--- | :--- | :--- |
| 000004 | DONE | DC CL3'ABC' |
| 000007 | BEFORE | EQU * |
| 000008 | DURING | DC F'200' |
| $00000 C$ | AFTER | EQU * |
| 000010 | NEXT | DS D |

You can specify multiple location counters for each control section in a source module; for more details about the location counter setting in control sections, see "Location Counter Setting" on page 61

Maximum Location Counter Value: The assembler carries internal location counter values as 4-byte (31-bit unsigned) values. When you specify the NOGOFF assembler option, the assembler uses only the low-order 3 bytes for the location counter, and prints only the low-order 3 bytes in the assembly source and object code listing if the LIST(121) option is active. All four bytes are displayed if the LIST(133) option is active. In this case the maximum valid location counter value is 224-1.

CMS, MVS When you specify the GOFF assembler option, the assembler requires the LIST(133) option, and uses the entire 4-byte value for the location counter and prints the 4-byte value in the assembly listings. In this case the maximum valid location counter value is $2^{31}-1$. CMS, MVS

If the location counter exceeds its valid maximum value the assembler issues error message

ASMA039S Location counter error
Controlling the Location Counter Value: You can control the setting of the location counter in a particular control section by using the START or ORG instruction, described in Chapter 3, "Program Structures and Addressing" and Chapter 5, "Assembler Instruction Statements,"] respectively. The counter affected by either of these assembler instructions is the counter for the control section in which they appear.

Location Counter Reference: You can refer to the current value of the location counter at any place in a program by using an asterisk as a term in an operand. The asterisk is a relocatable term, specified according to the following rules:

- The asterisk can be specified only in the operands of:
- Machine instructions
- DC and DS instructions
- EQU, ORG, and USING instructions
- It can also be specified in literal constants. See "Literals" on page 40 For example:

THERE L =3A(*)
generates three identical address constants, each with value A(THERE).
The value of the location counter reference (*) is the same as the value of the symbol THERE, the current value of the location counter of the control section in which the asterisk $\left(^{*}\right)$ is specified as a term. The asterisk has the same value as the address of the first byte of the instruction in which it appears. For example:
HERE B *+8
where the value of * is the value of HERE.

For the value of the asterisk in address constants with duplication factors, see
"Subfield 1: Duplication Factor" on page 132 of "DC Instruction" on page 126 and "Address Constants-A and Y" on page 153. For a discussion of location counter references in literals, see "Subfield 1: Duplication Factor" on page 132

## Symbol Length Attribute Reference

The length attribute of a symbol may be used as a term. Reference to the attribute is made by coding $L^{\prime}$ followed by the symbol, as in:

## L'BETA

The length attribute of BETA is substituted for the term. When you specify a symbol length attribute reference, you obtain the length of the instruction or data named by a symbol. You can use this reference as a term in instruction operands to:

- Specify assembler-determined storage area lengths
- Cause the assembler to compute length specifications for you
- Build expressions to be evaluated by the assembler

The symbol length attribute reference must be specified according to the following rules:

- The format must be L' immediately followed by a valid symbol (L'SYMBOL), an expression (L'SYMBOL+SYMBOL2-SYMBOL7), or the location counter reference ( $L^{\prime} *$ ). If the operand is an expression, the length attribute of its leftmost term is used.
- Symbols must be defined in the same source module in which the symbol length attribute reference is specified.
- The symbol length attribute reference can be used in the operand of any instruction that requires an absolute term. However, it cannot be used in the form L'* in any instruction or expression that requires a previously defined symbol.

The value of the length attribute is normally the length in bytes of the storage area required by an instruction, constant, or field represented by a symbol. The assembler stores the value of the length attribute in the symbol table along with the address value assigned to the symbol.

When the assembler encounters a symbol length attribute reference, it substitutes the value of the attribute from the symbol table entry for the symbol specified.

The assembler assigns the length attribute values to symbols in the name field of instructions as follows:

- For machine instructions (see 1 in Figure 14), it assigns either 2, 4, or 6, depending on the format of the instruction.
- For the DC and DS instructions (see $\mathbf{2}$ in Figure 14), it assigns either the implicitly or explicitly specified length of the first or only operand. The length attribute is not affected by a duplication factor.
- For the EQU instruction, it assigns the length attribute value of the first or only term (see $\mathbf{3}$ in Figure 14) of the first expression in the first operand, unless a specific length attribute is supplied in a second operand.
Note the length attribute values of the following terms in an EQU instruction:
- Self-defining terms (see 4 in Figure 14)
- Location counter reference (see 5 in Figure 14)
- L'* (see 6 in Figure 14)

For assembler instructions such as DC, DS, and EQU, the length attribute of the location counter reference ( $\mathrm{L}^{\prime *}$ - see $\mathbf{6}$ in Figure 14) is equal to 1. For machine instructions, the length attribute of the location counter reference ( $L^{\prime *}$ see 7 in Figure 14) is equal to the length attribute of the instruction in which the L'* appears.

Figure 14. Assignment of Length Attribute Values to Symbols in Name Fields

| Source Module |  |  | Length Attribute Reference <br> L'MACHA | Value of Symbol Length Attribute At Assembly Time |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
| MACHA | MVC | T0, FROM |  | 6 | 1 |
| MACHB | L | 3,ADCON | L'MACHB | 4 | 1 |
| MACHC | LR | 3,4 | L'MACHC | 2 | 1 |
| T0 | DS | CL80 | L'T0 | 80 | 2 |
| FROM | DS | CL240 | L'FROM | 240 | 2 |
| ADCON | DC | A (OTHER) | L'ADCON | 4 | 2 |
| CHAR | DC | C'YUKON' | L'CHAR | 5 | 2 |
| DUPL | DC | $3 F^{\prime} 200{ }^{\prime}$ | L'DUPL | 4 | 2 |
| RELOC1 | EQU | T0 3 | L'RELOC1 | 80 |  |
| RELOC2 | EQU | T0+80 $\quad 3$ | L'RELOC2 | 80 |  |
| RELOC3 | EQU | T0,44 ${ }^{3}$ | L'RELOC3 | 44 |  |
| ABS0L1 | EQU | FROM-T0 3 | L'ABS0L1 | 240 |  |
| ABSOL2 | EQU | ABSOL1 3 | L'ABSOL2 | 240 |  |
| SDT1 | EQU | 102 3 | L'SDT1 | 1 | 4 |
| SDT2 | EQU | $X^{\prime} F F^{\prime}+A-B$ | L'SDT2 | 1 | 4 |
| SDT3 | EQU | C'YUK' | L'SDT3 | 1 | 4 |
| ASTERISK | EQU | *+10 3 | L'ASTERISK | 1 | 5 |
| LOCTREF | EQU | L'* 3 | L'LOCTREF | 1 | 6 |
| LENGTH1 | DC | A(L'*) | L'* | 1 | 6 |
|  |  |  | L'LENGTH1 | 4 |  |
| LENGTH2 | MVC | TO(L'*), FROM | L'* | 6 | 7 |
| LENGTH3 | MVC | TO(L'T0-20), FROM | L'T0 | 80 |  |

Note: Instructions that contain length attribute references L'SDT1, L'SDT2, L'SDT3, L'ASTERISK, and L'LOCTREF as shown in this figure may generate ASMA019W.

The following example shows how to use the length attribute to move a character constant into either the high-order or low-order end of a storage field.

| A1 | DS | CL8 |
| :--- | :--- | :--- |
| B2 | DC | CL2'AB' |
| HIORD | MVC | A1(L'B2),B2 |
| LOORD | MVC | A1+ L'A1-L'B2(L'B2) ,B2 |

A1 names a storage field 8 bytes in length and is assigned a length attribute of 8 . B2 names a character constant 2 bytes in length and is assigned a length attribute of 2. The statement named HIORD moves the contents of B2 into the first 2 bytes of A1. The term L'B2 in parentheses provides the length specification required by the instruction.

The statement named LOORD moves the contents of B 2 into the rightmost 2 bytes of A1. The combination of terms $A 1+L^{\prime} A 1-L^{\prime} B 2$ adds the length of $A 1$ to the beginning address of $A 1$, and subtracts the length of $B 2$ from this value. The result is the address of the seventh byte in field A 1 . The constant represented by B 2 is moved into A1 starting at this address. L'B2 in parentheses provides the length specification in both instructions.

For ease in following the preceding example, the length attributes of A1 and B2 are specified explicitly in the DS and DC statements that define them. However, keep in mind that the L'symbol term makes coding such as this possible in situations where lengths are unknown. For example:

| C3 | DC | C'This is too long a string to be worth counting' |
| :--- | :--- | :--- |
| STRING | MVC | BUF (L'C3),C3 |

## Other Attribute References

Other attributes describe the characteristics and structure of the data you define in a program; for example, the kind of constant you specify or the number of characters you need to represent a value. These other attributes are:

- Count (K')
- Defined (D')
- Integer ( $I^{\prime}$ )
- Number ( ${ }^{\prime}$ ')
- Operation code ( $\mathrm{O}^{\prime}$ )
- Scale (S')
- Type ( $\mathrm{T}^{\prime}$ ); allowed only in conditional assembly.

You can refer to the count ( $\mathrm{K}^{\prime}$ ), defined ( $\mathrm{D}^{\prime}$ ), number ( $\mathrm{N}^{\prime}$ ), and operation code $\left(\mathrm{O}^{\prime}\right)$ attributes only in conditional assembly instructions and expressions. For full details, see""Data Attributes" on page 324

## Literals

You can use literals as operands in order to introduce data into your program. The literal is a special type of relocatable term. It behaves like a symbol in that it represents data. However, it is a special kind of term because it also is used to define the constant specified by the literal. This is convenient because:

- The data you enter as numbers for computation, addresses, or messages to be printed is visible in the instruction in which the literal appears.
- You avoid the added effort of defining constants elsewhere in your source module and then using their symbolic names in machine instruction operands.

The assembler assembles the data item specified in a literal into a literal pool (See "Literal Pool" on page 44). It then assembles the address of this literal data item in the pool into the object code of the instruction that contains the literal specification. Thus, the assembler saves you a programming step by storing your literal data for you. The assembler also organizes literal pools efficiently, so that the literal data is aligned on the correct boundary alignment and occupies a minimum amount of space.

## Literals, Constants, and Self-Defining Terms

Literals, constants, and self-defining terms differ in three important ways:

- Where you can specify them in machine instructions, that is, whether they represent data or an address of data
- Whether they have relocatable or absolute values
- What is assembled into the object code of the machine instruction in which they appear

Figure 15 on page 42 shows examples of the differences between literals, constants, and self-defining terms.

1. A literal with a relocatable address:

|  | $L$ | $3,=F^{\prime} 33^{\prime}$ | Register 3 set to 33. |
| :--- | :--- | :--- | :--- |

2. A literal with a self-defining term and a symbol with an absolute value

3. A symbol having an absolute address value specified by a self-defining term

| LA | 4, LOCORE | Register 4 set to 1000. See note 4 |
| :--- | :--- | :--- | :--- |
| LA | 4,1000 | Register 4 set to 1000 . See note 3 |

## Notes:

1. A literal both defines data and represents data. The address of the literal is assembled into the object code of the instruction in which it is used. The constant specified by the literal is assembled into the object code, in the literal pool.
2. A constant is represented by a symbol with a relocatable value. The address of a constant is assembled into the object code.
3. A self-defining term has an absolute value. In this example, the absolute value of the self-defining term is assembled into the object code.
4. A symbol with an absolute value does not represent the address of a constant, but represents either immediate data or an absolute address. When a symbol with an absolute value represents immediate data, it is the absolute value that is assembled into the object code.

Figure 15. Differences between Literals, Constants, and Self-Defining Terms

## General Rules for Using Literals

You can specify a literal as either a complete operand in a machine instruction, or as part of an expression in the operand of a machine instruction. A literal can also be specified as the name field on a macro call instruction.

Because literals define read-only data, they must not be used in operands that represent the receiving field of an instruction that modifies storage.

The assembler requires a description of the type of literal being specified as well as the literal itself. This descriptive information assists the assembler in assembling the literal correctly. The descriptive portion of the literal must indicate the format of the constant. It can also specify the length of the constant.

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in a single operand of a DC assembler instruction. The only difference is that the literal must start with an equal sign (=), which indicates to the assembler that a literal follows. The length of the literal, including the equal sign, constant type and modifiers, delimiters, and nominal values is limited to a maximum of 256 characters.

A literal may be coded as indicated here:
$=10 \times L^{\prime}$ F3'
where the subfields are:

| Duplication factor | 10 |
| :--- | ---: |
| Type | X |
| Modifiers | L5 |
| Nominal value | 'F3' |

The following instruction shows one use of a literal:

```
GAMMA L 10,=F'274'
```

The statement GAMMA is a load instruction using a literal as the second operand. When assembled, the second operand of the instruction refers to the relative address at which the value $\mathrm{F}^{\prime} 274$ ' is stored.

You cannot rely on the ordering of literals in the literal pool remaining the same. For this reason, referring to a point that extends beyond the bounds of a literal is flagged with warning message ASMA015W. Here is an example of such a reference:

$$
\text { BETA L } 10,=F^{\prime} 274^{\prime}+4
$$

In general, literals can be used wherever a storage address is permitted as an operand, including in conjunction with an index register in instructions with the RX format. For example:
DELTA LH
5,=H'11,23,39,48,64'(6)
is equivalent to:

| DELTA | LH | 5,LENGTHS (6) |
| :--- | :--- | :--- |
|  | $\cdot$ |  |
| LENGTHS | DC |  |
|  | H' $11,23,39,48,64^{\prime}$ |  |

See "DC Instruction" on page 126 for a description of how to specify the subfields in a literal.

Literals cannot be used in any assembler instruction where a previously defined symbol is required, but length attribute references to previously defined literals are allowed. Literals are relocatable terms because the address of the literal, rather than the literal-generated constant itself, is assembled in the statement that references a literal. The assembler generates the literals, collects them, and places them in a specific area of storage, as explained under "Literal Pool" on page 44 . Because the assembler determines the order in which literals are placed in the literal pool, the effect of using two literals as paired relocatable terms (see "Paired Relocatable Terms" on page 47) is unpredictable.
"Location Counter Reference" on page 37 describes how you can use the current location counter in a literal.

## Expressions

This section discusses the expressions used in coding operand entries for source statements. You can use an expression to specify:

- An address
- An explicit length
- A modifier
- A duplication factor
- A complete operand

Expressions have absolute and relocatable values. Whether an expression is absolute or relocatable depends on the value of the terms it contains. The assembler evaluates relocatable and absolute expressions at assembly time. Figure 16 on page 45 shows examples of valid expressions.

In addition to expressions used in coding operand entries, there are three types of expression that you can use only in conditional assembly instructions: arithmetic, logical, and character expressions. They are evaluated during conditional assembly. For more information, see "Assigning Values to SET Symbols" on page 347

An expression is composed of a single term or an arithmetic combination of terms. The assembler reduces multiterm expressions to single values. Thus, you do not have to compute these values yourself. The following are examples of valid expressions:

| $*$ | BETA*10 |
| :--- | :--- |
| AREA1+X'2D' | B' $^{\prime} 101^{\prime}$ |
| $*+32$ | C'ABC' $^{\prime}$ |
| N-25 | 29 |
| FIELD+332 | L'FIELD $^{\text {FIELD }}$ |
| (EXIT-ENTRY+1)+GO | TAMBDA+GAMMA |
| ALPHA-BETA/(10+AREA*L'FIELD)-100 | $=F^{\prime} 1234^{\prime}$ |
| $=$ A(100,133,175,221)+8 |  |

Figure 16. Examples of Valid Expressions

## Rules for Coding Expressions

The rules for coding an absolute or relocatable expression are:

- Unary (operating on one value) operators and binary (operating on two values) operators are allowed in expressions.
- An expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression.
- An expression must not begin with a binary operator, nor can it contain two binary operators in succession. When + and - are used as prefix operators, then they are unary, and not binary, operators.
- An expression starting with * is interpreted as a location counter reference, and not a multiplication operator.
- An expression must not contain two terms in succession.
- No spaces are allowed between an operator and a term, nor between two successive operators.
- An expression can contain any number of unary and binary operators, and any number of levels of parentheses.
- A single relocatable term is not allowed in a multiply or divide operation. Note that paired relocatable terms have absolute values and can be multiplied and divided if they are enclosed in parentheses. See "Paired Relocatable Terms" on page 47
Figure 17 on page 46 shows the definitions of absolute and relocatable expressions.


Figure 17. Definitions of Absolute and Relocatable Expressions

## Evaluation of Expressions

A single-term expression, like 29 or BETA, has the value of the term involved. The assembler reduces a multiterm expression, like $25 * 10+A / B$ or $B E T A+10$, to a single value, as follows:

1. It evaluates each term.
2. It does arithmetic operations from left to right. However:
a. It does unary operations before binary operations.
b. It does binary operations of multiplication and division before the binary operations of addition and subtraction.
3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives 0 .
4. In parenthesized expressions, the assembler evaluates the innermost expressions first and then considers them as terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated.
5. A term or expression's intermediate value and computed result must lie in the range of $-2^{31}$ through $+2^{31}-1$.

The assembler evaluates paired relocatable terms at each level of expression nesting.

## Absolute and Relocatable Expressions

An expression is absolute if its value is unaffected by program relocation. An expression is relocatable if its value depends upon program relocation. The two types of expressions, absolute and relocatable, take on these characteristics from the term or terms composing them. A description of the factors that determine whether an expression is absolute or relocatable follows.

Absolute Expression: An absolute expression is one whose value remains the same after program relocation. The value of an absolute expression is called an absolute value.

An expression is absolute, and is reduced to a single absolute value if the expression:

1. Comprises a symbol with an absolute value, a self-defining term, or a symbol length attribute reference, or any arithmetic combination of absolute terms.
2. Contains relocatable terms alone or in combination with absolute terms, and if all these relocatable terms are paired.

Relocatability Attribute: The relocatability attribute describes the attribute of a relocatable term. If a pair of terms are defined in the same control section, they are characterized as having the same relocatability attribute.

If the terms are defined in different control sections, or have different relocatability attributes, the expression is said to be "complex relocatable."

The relocatability attribute is the same as the ESDID for external symbols, and the "Relocation ID" in the listing.

Paired Relocatable Terms: An expression can be absolute even though it contains relocatable terms, provided that all the relocatable terms are paired. The pairing of relocatable terms cancels the effect of relocation.

The assembler reduces paired terms to single absolute terms in the intermediate stages of evaluation. The assembler considers relocatable terms as paired under the following conditions:

- The paired terms must have the same relocatability attribute.
- The paired terms must have opposite signs after all unary operators are resolved. In an expression, the paired terms do not have to be contiguous (that is, other terms can come between the paired terms).
- The value represented by the paired terms is absolute.

The following examples show absolute expressions. A is an absolute term; $X$ and $Y$ are relocatable terms with the same relocatability:

```
A-Y+X
A
A*A
X-Y+A
(*+*)-(*+*)
*-*
```

A reference to the location counter must be paired with another relocatable term from the same control section; that is, with the same relocatability. For example:
*-Y
Relocatable Expression: A relocatable expression is one whose value changes by $n$ if the origin of the control section in which it appears is relocated $n$ bytes.

A relocatable expression can be a single relocatable term. The assembler reduces a relocatable expression to a single relocatable value if the expression:

1. Is composed of a single relocatable term, or
2. Contains relocatable terms, alone or in combination with absolute terms, and
a. All the relocatable terms but one are paired. Note that the unpaired term gives the expression a relocatable value; the paired relocatable terms and other absolute terms constitute increments or decrements to the value of the unpaired term.
b. The relocatability attribute of the whole expression is that of the unpaired term.
c. The sign preceding the unpaired relocatable term must be positive, after all unary operators have resolved.

The following examples show relocatable expressions. A is an absolute term, W and $X$ are relocatable terms with the same relocatability attribute, and $Y$ is a relocatable term with a different relocatability attribute.

```
Y-32*A W-X+* =F'1234' (1iteral)
* (reference to
    location counter)
W-X+W Y
W-X+Y A*A+W-W+Y
```

Complex Relocatable Expressions: Complex relocatable expressions, unlike relocatable expressions, can contain:

- Two or more unpaired relocatable terms
- An unpaired relocatable term preceded by a negative sign

Using the same symbols, the following are examples of complex relocatable expressions:

```
W+X *+*
X-Y A-W+Y
```

Complex relocatable expressions are used in A-type and Y-type address constants to generate address constant values. For more details, refer to Complex Relocatable Expressions", and "Address Constants-A and Y" on page 153.
V-type and S-type constants may not contain complex relocatable expressions.

You can assign a complex relocatable value to a symbol using the EQU instruction, as described on page 184.

## Chapter 3. Program Structures and Addressing

This chapter describes how you use symbolic addresses to refer to data in your assembler language program, and how you divide a large program into smaller parts and use symbolic addresses in one part to refer to data in another part.

## Object Program Structures

High Level Assembler supports two object-program models. The older "load module" model generally involves one or more independently relocatable control sections combined into a single block of machine language text, which is loaded into a single contiguous portion of memory. Addresses within this block of text are resolved to locations within the block, or are left unresolved. Such programs may be considered one-dimensional structures. Examples include MVS load modules, CMS modules, and VSE phases.

CMS, MVS - The second object-program model supports a two-dimensional structure called a program object. The loaded program may consist of one or more contiguous blocks of machine language text grouped in classes and placed in different portions of memory. Each contribution of machine language text to a class is provided by an owning section, and the independently relocatable text from a section that contributes to a class is an element. For certain types of class, an element may contain parts. Unlike a control section, a program object section may specify more than one independently relocatable block of text. Addresses within each class may be resolved to addresses in the same or different classes. A class in a program object has behavior properties similar to those of a load module.

Section names are specified with the CSECT, RSECT, and START statements, and class and part names are specified with the CATTR statement. Additional attributes can be assigned to external symbols with the XATTR statement.

## CMS, MVS

The program object model can be created only when the GOFF option is specified. The "load module" model can be created when either the NOGOFF or GOFF option is specified, but there are limitations on source program statements if GOFF is specified.

Note: The term "section" is used in different senses for each object-program model. In the load module model, a section is a control section. In the program object model, a section is a one-dimensional cross-section of program object data containing contributions to one or more classes.

CMS, MVS Note: Features supported by High Level Assembler when you specify the GOFF option may not be supported by the system linker/binder or run-time environment where the assembled program will be processed. You should check the relevant product documentation before utilizing the assembler's features.

The following figure illustrates the differences between the object-program models.


Figure 18. Load Module and Program Object Structures

## Source Program Structures

This part of the chapter explains how to subdivide a large program into smaller parts that are easier to understand and maintain. It also explains how to divide
these smaller parts such as one section or element to contain executable instructions, and another to contain data constants and work areas.

You should consider two different subdivisions when writing an assembler language program:

- The source module
- The control section (load module model), or sections, elements, and parts (program object model)

You can divide a program into two or more source modules. Each source module is assembled into a separate object module. The object modules can then be combined to form an executable program.

You can also divide a source module into two or more sections, or (in the program object model) into sections containing multiple classes. Each section is assembled as part of the same object module. By writing the correct linker control statements, you can select a complete object module or any individual section of the object module to be linked and later loaded as an executable program.

Size of Program Components: If a source module becomes so large that its logic is not easily understood, divide it into smaller modules. For some instructions, at most 4096 bytes can be addressed by one base register. Long-displacement instructions allow you to address 1048576 bytes with one base register.

Communication between Program Components: You must be able to communicate between the components of your program; that is, be able to refer to data in a different component or branch to an instruction in another component.

To communicate between two or more source modules, you must link them together with applicable symbolic references.

To communicate between two or more sections or elements within a source module, you must correctly establish the addressability of each to the others.

## Source Module

A source module is composed of source statements in the assembler language. You can include these statements in the source module in two ways:

- You can enter them directly into the file that contains your source program.
- You specify one or more COPY instructions among the source statements being entered. When High Level Assembler encounters a COPY instruction, it replaces the COPY instruction with a predetermined set of source statements from a library. These statements then become a part of the source module. See "COPY Instruction" on page 122 for more details.


## Beginning of a Source Module

The first statement of a source module can be any assembler language statement, except MEXIT and MEND. You can initiate the first control section of a source module by using the START instruction. However, you can write some source statements before the beginning of the first control statement. See "First Section" on page 54 for more details.

## End of a Source Module

The END instruction marks the end of a source module. However, you can code several END instructions; conditional assembly processing can determine which of several coded or substituted END instructions is to be processed. Also, specifying the BATCH option allows you to supply more than one source module in a single input stream; when BATCH is specified, the assembler completes assembling a source module when an END statement is encountered, and if further statements are found in the input stream, assembly of a new source module is begun. See "END Instruction" on page 182 for more details, and HLASM Programmer's Guide for information about the BATCH option.

Conditional Assembly: Conditional assembly processing can determine which of several coded or substituted END instructions is to be processed.

## | Sections, Elements, and Parts

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

In the program object model, elements and parts are the smallest subdivisions of a program that can be relocated as a unit. Sections allow grouping all element and part contributions under a single name. The assembled sections, elements, and parts contain the object code for the machine instructions, data, and areas.

Consider the concept of a control section at different processing times:
At coding time: You create a control section or an element or part when you write the instructions it contains. In addition, you establish the addressability of each component within the source module, and provide any symbolic linkages between components that lie in different source modules. You also write the linker control statements to combine sections into a load module or program object, and to provide an entry point address for the beginning of program execution.

At assembly time: High Level Assembler translates the source statements into object code. Each source module is assembled into one object module. The contents of the object module are relocatable.

At linking time: As specified by linker or binder control statements, the linker or binder combines the object code of one or more sections into one load module or program object. It also calculates the addresses needed to accommodate common sections and external dummy sections from different object modules. In addition, it calculates the space needed to accommodate external dummy sections.

You can specify the relocatable address of the starting point for program execution in a linker control statement or request a starting address in the operand field of an assembler END instruction.

At program fetch time: The control program loads the load module or program object into virtual storage. All the relocatable addresses are converted to fixed locations in storage.

At execution time: The control program passes control to the loaded program now in virtual storage, and your program is run.

In the load module model, control sections may generate machine language text containing instructions and data, or define mappings of storage areas to be referenced at execution time. Control sections that generate machine language text are called executable control sections, even though they may contain only data. Control sections that create only mappings are called reference control sections.

CMS, MVS In the program object model, sections may define classes containing elements. (Classes are described in "Classes (MVS and CMS)" on page 59.) Elements may contain machine language text or define mappings, or both. Elements may in turn contain one or more parts, which are described at "Parts (MVS and CMS)" on page 61.

Elements containing machine language text are usually linked in a class comprising other elements containing machine language text, and elements defining mappings are usually linked in a class with other elements defining mappings.

The section name is used in binder operations to refer to its entire collection of elements and parts, but a program object section is not the same as a load module control section. A section name may be referenced as an external name only if defined as an entry point in an element belonging to that section. (By default, the assembler will generate an entry point in class B_TEXT with the section's name. See "Classes (MVS and CMS)" on page 59 for more information.) CMS, MVS

The term "executable" is used to describe both executable control sections in the load module model, or sections in the program object model.

You initiate a section by using the START, CSECT, or RSECT instruction, as described below:

- The START instruction can be used to initiate the first or only section of a source module. For more information about the START instruction, see "START Instruction" on page 214
- The CSECT instruction can be used anywhere in a source module to initiate or continue a section. For more information about the CSECT instruction, see "CSECT Instruction" on page 123
- Like the CSECT instruction, the RSECT instruction can be used anywhere in a source module to initiate or continue a section. Unlike the CSECT instruction, however, the RSECT instruction causes the assembler to check the coding in the section for possible violations of reenterability. For more information about the RSECT instruction, see "RSECT Instruction" on page 212.

A section can also be initiated as an unnamed section, or private code, without using the START, CSECT, or RSECT instruction. For more information, see "Unnamed Section" on page 55

## First Section

Before you initiate the first section in your source module, you may code only certain instructions. The following information lists those instructions that initiate the first section, and those instructions that may precede the first section.

What must appear before the first section: The ICTL instruction, if specified, must be the first statement in a source module.
*PROCESS statements must precede all other statements in a source module, except the ICTL instruction. There is a limit of 10 *PROCESS statements allowed in a source module. Additional *PROCESS statements are treated as assembler comment statements. See page 102 for a description of the *PROCESS statement.

What can optionally appear before the first executable control section: The instructions or groups of instructions that can optionally be specified before the first executable control section are:

- The following assembler instructions:

| ACONTROL | ADATA | AINSERT | ALIAS | CEJECT | COPY |
| :--- | :--- | :--- | :--- | :--- | :--- |
| DXD | EJECT | ENTRY | EXITCTL | EXTRN | ISEQ |
| MACRO | MEND | MEXIT | POP | PRINT | PUNCH |
| PUSH | REPRO | SPACE | TITLE | WXTRN | XATTR |

- Comment statements, including macro format comment statements
- Any statement which is part of an inline macro definition
- Common control sections
- Dummy control sections
- Any conditional assembly instruction
- Macro instructions that do not generate statements that establish the first section

The above instructions or groups of instructions belong to a source module, but are not considered part of an executable section.

Instructions that establish the first section: Any instruction that affects the location counter, or uses its current value, establishes the beginning of the first executable section. The instructions that establish the first section include any machine instruction and the following assembler instructions:

| CCW | CCW0 | CCW1 | CNOP | CSECT | CXD |
| :--- | :--- | :--- | :--- | :--- | :--- |
| DC | DS | EQU | LOCTR | LTORG | ORG |
| RSECT | START | USING |  |  |  |

CSECT, RSECT, and START start a possibly named control section. The other statements start an unnamed control section.

These instructions are always considered a part of the control section in which they appear.

The statements copied into a source module by a COPY instruction determine whether it initiates the first control section. The PROFILE option causes the assembler to generate a COPY statement as the first statement after any ICTL or *PROCESS statements.

The DSECT, COM, and DXD instructions initiate reference control sections and do not establish the first executable section.

Any instructions copied by a COPY instruction, or generated by the processing of a macro instruction before the first section, must belong exclusively to one of the groups of instructions shown above. Any other instructions cause the assembler to establish the first section.

All the instructions or groups of instructions listed above can also appear as part of a section.

If you specify the PROFILE assembler option the assembler generates a COPY statement as the first statement in the assembly after any ICTL or *PROCESS statements. The copy member should not contain any ICTL or *PROCESS statements.

## Unnamed Section

The unnamed section is an executable section that can be initiated in one of the following two ways:

- By coding a START, CSECT, or RSECT instruction without a name entry
- By coding any instruction, other than the START, CSECT, or RSECT instruction, that initiates the first executable section

An unnamed control section is sometimes referred to as private code. Private code sections are sometimes difficult to manage with other system components such as linkers and configuration management tools. Avoiding their use is recommended. (Zero-length private code sections are sometimes ignored or discarded by system linkers.)

All sections should be given names so they can be referred to symbolically:

- Within a source module
- In EXTRN and WXTRN instructions
- in linker control statements for section ordering and replacement, and for linkage between source modules

Unnamed common control sections or dummy control sections can be defined if the name entry is omitted from a COM or DSECT instruction.

If you include an AMODE or RMODE instruction in the assembly and leave the name field blank, you must provide an unnamed control section.

## Reference Control Sections

A reference control section is one you initiate by using the DSECT, COM, or DXD instruction, as follows:

- You can use the DSECT instruction to initiate or continue a dummy control section. For more information about dummy sections, see "Dummy Control Sections."
- You can use the COM instruction to initiate or continue a common control section. For more information about common sections, see "Common Control Sections" on page 57
- You can use the DXD instructions to define an external dummy section. For more information about external dummy sections, see "External Dummy Sections" on page 57

At assembly time, reference control sections are not assembled into object code. You can use a reference control section either to reserve storage areas or to describe data to which you can refer from executable control sections. These reference control sections are considered empty at assembly time, and the actual binary data to which they refer is not available until execution time.

## Dummy Control Sections

A dummy control section is a reference control section that describes the layout of data in a storage area without actually reserving any virtual storage.

You may want to describe the format of an area whose storage location is not determined until the program is run. You can do so by describing the format of the area in a dummy section, and using symbols defined in the dummy section in the operands of machine instructions.

The DSECT instruction initiates a dummy control section or indicates its continuation. For more information about the DSECT instruction, see "DSECT Instruction" on page 178

How to use a dummy control section: A dummy control section (dummy section) lets you write a sequence of assembler language statements to describe the layout of data located elsewhere in your source module. The assembler produces no object code for statements in a dummy control section, and it reserves no storage in the object module for it. Rather, the dummy section provides a symbolic format that is empty of data. However, the assembler assigns location values to the symbols you define in a dummy section, relative to its beginning.

Therefore, to use a dummy section, you must:

- Reserve a storage area for the data
- Ensure that the locations of the symbols in the dummy section actually correspond to the locations of the data being described
- Establish the addressability of the dummy section in combination with the storage area

You can then refer to the data symbolically by using the symbols defined in the dummy section.

## Common Control Sections

A common control section is a reference control section that lets you reserve a storage area that can be used by one or more source modules. One or more common sections can be defined in a source module.

The COM instruction initiates a common control section, or indicates its continuation. For more information about the COM instruction, see "COM Instruction" on page 121

How to use a common control section: A common control section (common section) lets you describe a common storage area in one or more source modules.

When the separately assembled object modules are linked as one program, the required storage space is reserved for the common control section. Thus, two or more modules may share the common area.

Only the storage area is provided; the assembler does not assemble the source statements that make up a common control section into object code. You must provide the data for the common area at execution time.

The assembler assigns locations to the symbols you define in a common section relative to the beginning of that common section. This lets you refer symbolically to the data that is placed in the common section at execution time. If you want to refer to data in a common control section, you must establish the addressability of the common control section in each source module that contains references to it. If you code identical common sections in two or more source modules, you can communicate data symbolically between these modules through this common section.

Communicating with Modules in Other Languages: Some high-level languages such as COBOL, PL/I, C, and FORTRAN use common control sections. This lets you communicate between assembler language modules and modules written in those languages.

## External Dummy Sections

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

- Work areas for each source module
- Communication areas between two or more source modules

Note: External dummy sections are also called "pseudo-registers" in other contexts.
When the assembled object modules are linked and loaded, you can dynamically allocate the storage required for all your external dummy sections at one time from one source module (for example, by using the MVS GETMAIN macro instruction).

This is not only convenient, but it saves space and reduces fragmentation of virtual storage.

Typical bind-time processing of external dummy sections involves "merging" the attributes of identically-named external dummy sections, retaining only the longest length and strictest alignment. In particular, the lengths of identically-named external dummy sections are not additive.

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

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

For more information about the DXD and CXD instructions, see "DXD Instruction" on page 180 and "CXD Instruction" on page 125

Note: The names of dummy external control sections may match the names of other external symbols that are not names of dummy control sections, without conflict.

Generating an external dummy section: An external dummy section is generated when you specify a DXD instruction, or when you specify a DSECT instruction whose name appears in a Q-type address constant.

When a DSECT name is used as an operand of a Q-type address constant, that name becomes an external symbol with type XD. The name must satisfy the name-length requirements of the object file format specified in the assembler options.

DXD names may match the names of other types of external symbols without conflict.

Use the Q-type address constant to reserve storage for the offset to the external dummy section whose name is specified in the operand. This offset is the distance in bytes from the beginning of the area allocated for all the external dummy sections to the beginning of the external dummy section specified. You can use this offset value to address the external dummy section.

Using external dummy sections: To use an external dummy section, you must do the following:

1. Identify and define the external dummy section. The assembler computes the length and alignment required. The linker will merge this definition with other definitions of the same name, assigning the longest length and strictest alignment.
2. Provide a Q-type constant for each external dummy section defined.
3. Use the CXD instruction to reserve a fullword area into which the linker or loader inserts the total length of all the external dummy sections that are specified in the source modules of your program. The linker computes this length from the accumulated lengths of the individual external dummy sections supplied by the assembler.
4. Allocate a storage area using this computed total length.
5. Load the address of the allocated area into a register.
6. Add to the address in the register the offset into the allocated area of the applicable external dummy section. The linker inserts this offset into the area reserved by the associated Q-type address constant.
7. Establish the addressability of the external dummy section in combination with the portion of the allocated area reserved for the external dummy section.

You can now refer symbolically to the locations in the external dummy section. Note that the source statements in an external dummy section are not assembled into object code. Thus, you must create the data described by external dummy sections at execution time.

Note: During linking, external dummy sections may be arranged in any order. Do not assume any ordering relationship among external dummy sections.

## Classes (MVS and CMS)

Each section's contributions to a program object are assigned to one or more classes, according to their desired binding and loading properties. Class names are assigned either by default (see "Default Class Assignments" on page 60 or explicitly. You define a class with the CATTR instruction, which must follow the initiation of an executable section. The class name is provided in the name entry of the CATTR instruction, and attributes of the class are provided by the operands of the first CATTR instruction declaring the class. (See "CATTR Instruction (MVS and CMS)" on page 112 for further information.) The element containing subsequent machine language text or storage definitions is defined by the combination of the section and class names, as illustrated in Figure 18 on page 51.

For example, suppose you define two classes, CLASS_X and CLASS_Y:

```
SECT_A CSECT , Define section SECT_A
CLASS_X CATTR RMODE(ANY) Define class CLASS_X
    - - Statements for CLASS_X
CLASS_Y CATTR RMODE(24) Define class CLASS_Y
    - - Statements for CLASS_Y
```

The statements following the first CATTR instruction will be assigned to an element defined by the section name SECT_A and the class name CLASS_X. Similarly, the statements following the second CATTR instruction will be assigned to an element defined by the section name SECT_A and the class name CLASS_Y. CLASS_Y will be loaded below 16Mb, and CLASS_X may be loaded anywhere below 2 Gb .

Class names are rarely referenced, because the attributes of the class, such as RMODE, are much more important.

You can resume a class by providing additional CATTR statements with the class name in the name entry. No attributes of the class may be specified after the first CATTR statement declaring the class.

Resuming a section will cause subsequent text to be placed in the B_TEXT class if there is no intervening CATTR statement defining or resuming a different class:

| SECT_A | CSECT , | Define section SECT_A |
| :--- | :--- | :--- |
| CLASS_X | CATTR RMODE (ANY) | Define class CLASS_X |
|  | --- | Statements for CLASS_X |
| CLASS_Y | CATTR RMODE (24) | Define class CLASS_Y |
|  | --- | Statements for CLASS_Y |
| SECT_A | CSECT , | Resume section SECT_A |
|  | --- | Statements for class B_TEXT |
| CLASS_X CATTR , | Resume class CLASS_X |  |
|  | --- | More statements for CLASS_X |

```

\section*{Class Binding and Loading Attributes}
```

Each class is bound into a separately relocatable loadable segment, using one of two binding attributes.

- Classes containing parts use merge binding (described at "Parts (MVS and CMS)" on page 61). Parts are the smallest independently relocatable components of a merge class.
- Classes not containing parts use concatenation binding, in which elements, after suitable alignment, are placed one after another. Zero-length elements are retained but take no space in the program object. Elements are the smallest independently relocatable components of a concatenation class
Each class must have uniform binding and loading attributes. More than one class may have identical attributes, and the binder may put such classes into one segment. The most usual class attributes are RMODE, alignment, and Loadability; see "CATTR Instruction (MVS and CMS)" on page 112 for further information.
Class loading attributes determine the load-time placement of segments in virtual storage. Loadable segments are loaded as separately relocated non-contiguous entities at different origin addresses.

```

\section*{Default Class Assignments}
```

High Level Assembler provides compatible behavior with "load module" model object files generated when the NOGOFF option is active. When the GOFF option is specified, the assembler automatically follows each CSECT, RSECT, and START statement by defining two classes: B_TEXT and B_PRV.

- B_TEXT contains the machine language text associated with the section name, and is assigned the RMODE of the section name. The section name is assigned to an entry point at the origin of the class. If a subsequent CATTR statement declares a class name before any other statements have defined storage, the element defined by the section name and the B_TEXT class name will be empty.
- B_PRV contains any external dummy sections defined by DXD instructions, or by DSECTs named in Q-type address constants. If none are defined, the elements in this class will be empty. ("PRV" is the binder's term for a "Pseudo Register Vector," the cumulative collection of external dummy sections.)
- High Level Assembler assigns the name of the section as an entry name at the initial byte of B_TEXT, and assigns to it the AMODE of the section name.
These two classes are bound in the same way as ordinary control sections and dummy external sections are bound in the load module model, and can be used to generate a load module if certain restrictions are satisfied.

```

\section*{Location Counter Setting}

\section*{Parts (MVS and CMS)} information.) declaring the class.

You may declare other classes in addition to the defaults, but the resulting program object will not be convertible to a load module.

Parts are the smallest externally named and independently relocatable subdivisions of elements in a merge class. A class containing parts may contain only parts, and a class containing anything other than parts may not contain any parts.

ENTRY statements may not define an entry point in a part.
You define a part with the CATTR instruction, which must follow the initiation of an executable section. The name of the class to which the part belongs is provided in the name entry of the CATTR instruction, and the name of the part is specified as an operand. The first definition of a class name may also specify the attributes of the class. (See "CATTR Instruction (MVS and CMS)" on page 112 for further

For example, suppose you define two parts in a class:
```

SECT_B CSECT , Define section SECT_B
PClass CATTR Part(Part_R),RMODE(ANY) Define class PClass, part Part_R
- - Statements included in Part_R
PClass CATTR Part(Part_S) Define part Part_S in class PClass
- - - Statements inclu\overline{ded in Part_S}
PClass CATTR Part(Part_R) Resume class PClass and part Part_R
- - More statements included in Part_\overline{R}

```

These statements define a "merge" class PClass containing two parts, Part_R and Part_S. If other classes or other object files declare parts with the same names in the same class, the binder will merge their contents to determine the final part definition in the program object.

You may provide additional statements for a part by specifying a CATTR statement with the class name in the name entry and the part name specified as the operand. No other class attributes may be specified following the first CATTR statement

Parts are automatically assigned a "merge" attribute, meaning that more than one identically named part may appear in a class defined in other assemblies or compilations. The binder will assign the longest length and strictest alignment of all such identically-named parts, and will merge the machine language text contributions of each to form the final text belonging to that part. The order of text merging depends on the sequence of parts processing by the binder.

Note: During linking, parts may be arranged in any order, depending on their priority attribute. Do not assume any ordering relationship among parts.

The assembler maintains a separate location counter for each section, element, and part. The location counter setting for the first section starts at 0 , except when an initial section is started with a START instruction that specifies a nonzero location counter value. The location values assigned to the instructions and other data in a section, element, or part are, therefore, relative to the location counter setting at the beginning of that section, element, or part.

For executable sections, the location values that appear in the listings depend on the THREAD option:
- If you specify NOTHREAD, the location counter values for each section, element, or part restart at 0, except possibly those associated with a first section initiated by a START instruction with a nonzero address.
- If you specify THREAD, location counter values do not restart at 0 for each subsequent section, element, or part. They continue, after suitable alignment, from the end of the previous section, element, or part.

For reference control sections, the location values that appear in the listings always start from 0.

You can continue a control section, element, or part that has been discontinued and, thereby, intersperse code sequences from different control sections, elements, or parts. Note that the location values that appear in the listings for such discontiguous sequences are divided into segments that follow from the end of one segment to the beginning of the subsequent segment.

The location values, listed for the next defined control section, element, or part, begin after the last location value assigned to the preceding such item.

On VSE, or when you specify the NOGOFF assembler option on MVS and CMS, the maximum value of the location counter and the maximum length of a control section is \(2^{24}-1\), or \(\mathrm{X}^{\prime}\) FFFFFF' bytes. If LIST(133) is in force, then the high-order is shown as zero.

CMS, MVS When you specify the GOFF assembler option, the maximum value of the location counter and the maximum length of an element or part is \(2^{231}-1\), or X'7FFFFFFF' bytes. CMS, MVS

\section*{Location Counter and Length Limits}

The assembler also maintains a length counter for each individually relocatable component of the program: executable and reference control sections, elements, and parts.

If any location counter overflows its maximum value, High Level Assembler issues the severe error message:

ASMA039S Location counter error
and continues assembling with the location counter value "wrapping" around to zero.

The length of a section, element, or part cannot exceed the maximum allowed length described above. If the length counter reaches this maximum value, it stays fixed at that value without an error condition or error message. Exceeding the length counter will cause overflow of the location counter, producing the ASMA039S message.

The location counter setting is relative to the beginning of the location it represents, and the length counter represents the cumulative length of the control section. This means that the length counter is nearly always greater than the location counter, and can exceed its maximum value before the location counter. Even if the location counter overflows, the length counter value may be correct, and
reassembling with the NOTHREAD option may avoid the location counter overflow condition.

\section*{Use of Multiple Location Counters}

High Level Assembler lets you use multiple location counters for each individual control section. Use the LOCTR instruction (see "LOCTR Instruction" on page 191) to assign different location counters to different parts of a control section. The assembler then rearranges and assembles the coding together, according to the different location counters you have specified: all coding using the first location counter is assembled together, then the coding using the second location counter is assembled together, and so forth.

An example of the use of multiple location counters is shown in Figure 19. In the example, executable instructions and data areas have been interspersed throughout the coding in their logical sequence, each group of instructions preceded by a LOCTR instruction identifying the location counter under which it is to be assembled. The assembler rearranges the control section so that the executable instructions are grouped together and the data areas are grouped together. Symbols are not resolved in the order they appear in the source program, but in location counter sequence.


Figure 19. Use of Multiple Location Counters
The interactions of the LOCTR instruction with sections, classes, and parts is described at "LOCTR Instruction" on page 191.

\section*{Addressing}

This part of the chapter describes the techniques and introduces the instructions that let you use symbolic addresses when referring to instructions and data. You can address code and data that is defined within the same source module, or code and data that is defined in another source module. Symbolic addresses are more meaningful and easier to use than the corresponding object code addresses required for machine instructions. The assembler can convert the symbolic addresses you specify into their object code form.

The System/390® and z/Architecture architectures have two ways of resolving addresses in your program, depending on the machine instruction type:
- base-displacement, where the address is computed by adding the displacement to the contents of a base register.
- relative-immediate, where the address is computed by adding \(2 \times\) the signed immediate operand field to the instruction's address (refer to "RI Format" on page 91 and "RSI Format" on page 95.

\section*{Addressing within Source Modules: Establishing Addressability}

You can use symbolic addresses in machine instructions and certain assembler instructions. This is much easier than explicitly coding the addresses in the form required by the hardware. Symbolic addresses you code in the instruction operands are implicit addresses, and addresses in which you specify the base-displacement or intermediate form are explicit addresses.

The assembler converts your implicit addresses into the explicit addresses required for the assembled object code of the machine instruction. However, for base-displacement operands, you must first establish addressability, as described below.

Base Address Definition: The term base address is used throughout this manual to mean the location counter value within a control section, element, or part from which the assembler can compute displacements to locations, or addresses. The base address need not always be the storage address of a control section, element, or part when it is loaded into storage at execution time.

\section*{How to Establish Addressability}

To establish the addressability of a control section, element, or part (see" "Sections, Elements, and Parts" on page 52, you must:
- Specify a base address from which the assembler can compute displacements to the addresses within the control section, element, or part.
- Assign the base registers to contain the base addresses.
- Write the instructions that load the base registers with the base addresses.

The following example shows the base address at MYPROG, that is assigned by register 12. Register 12 is loaded with the value in register 15 , which by convention usually contains the storage address (set by the operating system) of the control section (CSECT) when the program is loaded into storage at execution time.
```

MYPROG CSECT The base address
USING MYPROG,12 Assign the base register
LR 12,15 Load the base address

```

Similarly, you can use a BASR or similar instruction to put the address of the following instruction into register 12.
\[
\begin{aligned}
& \text { BASR } 12,0 \\
& \text { USING *,12 }
\end{aligned}
\]

The USING instruction indicates that register 12 may be used as a base register containing that address.

During assembly, the implicit addresses you code are converted into their explicit base-displacement form; then, they are assembled into the object code of the machine instructions in which they have been coded.

During execution, the base address is loaded into the base register.
\begin{tabular}{ll} 
CMS, MVS & If you specify multiple classes, you must provide addressability for \\
each element. For example, suppose you define two classes that must reference \\
positions in the other: \\
MYPROG CSECT
\end{tabular}

Parts must always be referenced using Q-type address constants using the techniques shown in this example, whether or not they reside in deferred load classes. This is because parts are subject to reordering during binding.

\section*{Base Register Instructions}

The USING and DROP assembler instructions enable you to use expressions representing implicit addresses as operands of machine instruction statements, leaving the assignment of base registers and the calculation of displacements to the assembler.

In order to use symbols in implicit addresses in the operand field of machine instruction statements, you must:
- Code a USING instruction to assign one or more base registers to a base address or sequence of base addresses
- Code machine instructions to load each base register with the base address

Having the assembler determine base registers and displacements relieves you of the need to separate each address into an explicit displacement value and an explicit base register value. This feature of the assembler eliminates a likely source of programming errors, thus reducing the time required to write and test programs. You use the USING and DROP instructions to take advantage of this feature. For information about how to use these instructions, see "USING Instruction" on page 218 and "DROP Instruction" on page 172

\section*{Qualified Addressing}

Qualified addressing lets you use the same symbol to refer to data in different storage locations. Qualified symbols are simply ordinary symbols prefixed by a symbol qualifier and a period. A symbol qualifier is used to specify which base register the assembler should use when converting an implicit address into its explicit base-displacement form. Before you use a symbol qualifier, you must have previously defined it in the name entry of a labeled USING instruction. For information about labeled USING instructions, see "USING Instruction" on page 218 When defined, you can use a symbol qualifier to qualify any symbol that names a storage location within the range of the labeled USING. Qualified symbols may be used anywhere a relocatable term may be used.

The following examples show the use of qualified symbols. SOURCE and TARGET are both symbol qualifiers previously defined in two labeled USING instructions. \(X\) and \(Y\) are both symbols that name storage locations within the range of both labeled USING instructions.
\begin{tabular}{ll} 
MVC & TARGET. \(X\), SOURCE. \(X\) \\
MVC & TARGET. \(Y+5(3)\), SOURCE. \(Y+5\) \\
XC & TARGET. \(X+10\left(L^{\prime} X-10\right)\), TARGET. \(X+10\) \\
LA & 2, SOURCE. \(Y\)
\end{tabular}

\section*{Dependent Addressing}

Dependent addressing lets you minimize the number of base registers required to refer to data by making greater use of established addressability. For example, you may want to describe the format of a table of data defined in your source module with a dummy control section (see"Dummy Control Sections" on page 56. To refer to the data in the table using the symbols defined in the dummy section, you need to establish the addressability of the dummy section. To do this you must:
- Code a USING instruction to assign one or more base registers to a base address
- Code machine instructions to load each base register with the base address

However, dependent addressing offers an alternative means of establishing addressability of the dummy section.

When you have established addressability of the control section in which the table is defined, you can establish addressability of the dummy section by simply coding a USING statement which specifies the name of the dummy section and the address of the table. When you subsequently refer to the symbols in the dummy section, the assembler makes use of the already established addressability of the control section when converting the symbolic addresses into their base-displacement form.

\section*{Relative Addressing}

Relative addressing is the technique of addressing instructions and data areas by designating their location in relation to the location counter or to some symbolic location. This type of addressing is always in bytes-never in bits, words, or instructions. Thus, the expression \(*+4\) specifies an address that is 4 bytes greater than the current value of the location counter. In the sequence of instructions in the following example, the location of the CR machine instruction can be expressed in two ways, ALPHA+2, or BETA-4, because all the machine instructions in the example are for 2-byte instructions.

\section*{Addressing}
\begin{tabular}{lll} 
ALPHA & LR & 3,4 \\
& CR & 4,6 \\
& BCR & 1,14 \\
BETA & AR & 2,3
\end{tabular}

\section*{Literal Pools}

Literals, collected into pools by the assembler, are assembled as part of the executable control section to which the pools belong. If a LTORG instruction is specified at the end of each control section or element, the literals specified for that section or element are assembled into the pool starting at the LTORG instruction. If no LTORG instruction is specified, a literal pool containing all the literals used in the whole source module is assembled at the end of the first control section or at the end of the B_TEXT class belonging to the first section. This literal pool appears in the listings after the END instruction. For more information about the LTORG instruction, see "LTORG Instruction" on page 193.

Independently Addressed Segments: If any control section is divided into independently addressed segments, a LTORG instruction should be specified at the end of each segment to create a separate literal pool for that segment.

\section*{Establishing Residence and Addressing Mode}

The AMODE and RMODE instructions specify the addressing mode (AMODE) and the residence mode (RMODE) to be associated with control sections in the object deck. These modes may be specified for the following types of control sections:
- Control section (for example START, CSECT)
- Unnamed control section
- Common control section (COM instruction)

The assembler sets the AMODE and RMODE indicators in the ESD record for each applicable control section in an assembly. The linker stores the AMODE and RMODE values in the load module. They are subsequently used by the loader program that brings the load module into storage. The loader program uses the RMODE value to determine where it loads the load module, and passes the AMODE value to the operating system to establish the addressing mode.

CMS, MVS When you specify the GOFF option, the RMODE value specified for a section is by default assigned to the B_TEXT class, and the AMODE specified for the section is assigned to an entry point having the section name and the location of the first byte of class B_TEXT. If the source program defines additional classes, each class may be assigned its own RMODE, and an entry point in any class may be assigned its own AMODE. CMS, MVS

For more information about the AMODE and RMODE instructions, see "AMODE Instruction" on page 110 and "RMODE Instruction" on page 211

\section*{Symbolic Linkages}

Symbols can be defined in one module and referred to in another, which results in symbolic linkages between independently assembled program sections. These linkages can be made only if the assembler can provide information about the linkage symbols to the linker, which resolves the linkage references at link-edit time.

\section*{Establishing symbolic linkage}

You must establish symbolic linkage between source modules so that you can refer to or branch to symbolic locations defined in the control sections of external source modules. You do this by using external symbol definitions, and external symbol references. To establish symbolic linkage with an external source module, you must do the following:
- In the current source module, you must identify the symbols that are not defined in that source module, if you want to use them in instruction operands. These symbols are called external symbols, because they are defined in another (external) source module. You identify external symbols in the EXTRN or WXTRN instruction, or the V-type address constant. For more information about the EXTRN and WXTRN instructions, see "EXTRN Instruction" on page 189 and "WXTRN Instruction" on page 229
- In the external source modules, you must identify the symbols that are defined in those source modules, and that you refer to from the current source module. The two types of definitions that you can use are control section names (defined by the CSECT, RSECT, and START instructions), and entry symbols. Entry symbols are so called because they provide points of entry to a control section in a source module. You identify entry symbols with the ENTRY instruction. For more information about the ENTRY instruction, see "ENTRY Instruction" on page 183
- To reference the external symbols, you must either
- provide the A-type or V-type address constants needed by the assembler to reserve storage for the addresses represented by the external symbols, or
- reference an external symbol in the same class in a relative branch instruction.

The assembler places information about entry and external symbols in the external symbol dictionary. The linker uses this information to resolve the linkage addresses identified by the entry and external symbols.

\section*{Referring to external data}

Use the EXTRN instruction to identify the external symbol that represents data in an external source module, if you want to refer to this data symbolically.

For example, you can identify the address of a data area as an external symbol and load the A-type address constant specifying this symbol into a base register. Then, you use this base register when establishing the addressability of a dummy section that describes this external data. You can now refer symbolically to the data that the external area contains.

You must also identify, in the source module that contains the data area, the address of the data as an entry symbol.

\section*{Branching to an external address}

Use the V-type address constant to identify the external symbol that represents the address in an external source module that you want to branch to.

For example, you can load into a register the V-type address constant that identifies the external symbol. Using this register, you can then branch to the external address represented by the symbol.

If the symbol is the name entry of a START, CSECT, or RSECT instruction in the other source module, and thus names an executable control section, it is automatically identified as an entry symbol. If the symbol represents an address in the middle of a control section, you must identify it as an entry symbol for the external source module.

You can also use a combination of an EXTRN instruction to identify, and an A-type address constant to contain, the external branch address. However, the V-type address constant is more convenient because:
- You do not have to use an EXTRN instruction.
- The external symbol you specify, can be used in the name entry of any other statement in the same source program.
- It will work correctly even if the program is linked as an overlay module, so long as the reference is not to a symbol in an exclusive segment. See Z/OS MVS Program Management: User's Guide and Reference SA22-7643 for further information.

The following example shows how you use an A-type address constant to contain the address of an external symbol that you identify in an EXTRN instruction. You cannot use the external symbol name EXMOD1 in the name entry of any other statement in the source program.
\begin{tabular}{|c|c|c|c|}
\hline & \[
\begin{aligned}
& \mathrm{L} \\
& \text { BASR }
\end{aligned}
\] & \[
\begin{aligned}
& \text { 15,EX_SYM } \\
& 14,15
\end{aligned}
\] & Load address of external symbol Branch to it \\
\hline & - & & \\
\hline \multirow[t]{4}{*}{EX_SYM} & \({ }_{\text {D }}\) & A(EXMOD1) & Address of external symbol \\
\hline & EXTRN & EXMOD1 & Identify EXMOD1 as external symbol \\
\hline & - & & \\
\hline & & & \\
\hline
\end{tabular}

The following example shows how you use the symbol EXMOD1 as both the name of an external symbol and a name entry on another statement.
\begin{tabular}{|c|c|c|c|}
\hline & L & 15,EX_SYM & Load address of external symbol \\
\hline & BASR & 14,15 & Branch to it \\
\hline & - & & \\
\hline EXMOD1 & DS & 0H & Using EXMOD1 as a name entry \\
\hline & - & & \\
\hline EX_SYM & \({ }_{\text {D }}\) & V (EXMOD1) & Address of external symbol \\
\hline & . & & \\
\hline & . & & \\
\hline
\end{tabular}

If the external symbol that represents the address to which you want to branch is to be part of an overlay-structured module, you should identify it with a V-type address constant, not with an EXTRN instruction and an A-type address constant. You can use the supervisor CALL macro instruction to branch to the address represented by the external symbol. The CALL macro instruction generates the necessary V-type address constant.

CMS, MVS You may branch to external symbols in the same class using relative branch instructions.
```

MYPROG CSECT , Define section MYPROG
CLASS_A CATTR RMODE(31)
- - -
BRAS 14,ENTRYB Branch to external symbol
- - -
HISPROG CSECT , Define section HISPROG
CLASS_A CATTR RMODE(31) Define class CLASS_A
ENTRYB STM 14,12,12(13) Entry point referenced externally
- - -
END

```

You may also use a relative branch instruction to branch to an externally defined symbol:
```

MYPROG CSECT , Define section MYPROG
MYCLASS CATTR RMODE(31) Define class MYCLASS
EXTRN TARGET Declare external symbol TARGET
BRAS 14,TARGET Branch to external symbol
END

```

A separate source module must define the entry point TARGET in class MYCLASS.
CMS, MVS

\section*{Establishing an external symbol alias}

You can instruct the assembler to use an alias for an external symbol in place of the external symbol itself, when it generates the object module. To do this you must code an ALIAS instruction which specifies the external symbol and the alias you want the assembler to use. The external symbol must be defined in a START, CSECT, RSECT, ENTRY, COM, DXD, external DSECT, EXTRN, or WXTRN instruction, or in a V-type address constant.

The following example shows how you use the ALIAS instruction to specify an alias for the external symbol EXMOD1.


See "ALIAS Instruction" on page 109 for information about the ALIAS instruction.

\section*{External Symbol Dictionary Entries}

For each section, class, part, entry, external symbol, and dummy external control section, the assembler keeps a record of the following external symbol dictionary (ESD) information:
- Symbolic name, if one is specified
- Type code
- Individual identification number (ESDID)
- Starting address
- Length
- Owning ESDID, if any
- Symbol attributes
- Alias, if one is specified

Figure 20 lists the assembler instructions that define control sections and dummy control sections, classes and parts, or identify entry and external symbols, and tells their associated type codes. You can define up to 65535 individual control sections and external symbols in a source module if the NOGOFF option is specified, or up to 999999 external symbols if the GOFF option is specified.

Figure 20. Defining External Symbols
\begin{tabular}{|c|c|c|c|}
\hline \multirow[t]{2}{*}{Name Entry} & \multirow[t]{2}{*}{Instruction} & \multicolumn{2}{|l|}{Coding Entered into External Symbol Dictionary} \\
\hline & & NOGOFF option & GOFF option \\
\hline If present & START, CSECT, or RSECT & SD & SD, ED, LD \\
\hline If omitted & START, CSECT, or RSECT & PC & SD \\
\hline Instructiondependent & Any instruction that initiates the unnamed section & PC & SD \\
\hline Optional & COM & CM & CM \\
\hline Optional & DSECT & None & None \\
\hline Mandatory & DXD or external DSECT & XD & XD \\
\hline Mandatory & CATTR & Not applicable & ED \\
\hline Mandatory & CATTR PART(name) & Not applicable & PD \\
\hline Not applicable & ENTRY & LD & LD \\
\hline Not applicable & EXTRN & ER & ER \\
\hline Not applicable & DC (V-type address constant) & ER & ER \\
\hline Not applicable & WXTRN & WX & WX \\
\hline
\end{tabular}

Refer to Appendix C Object Deck Output in the HLASM Programmer's Guide, SC26-4941 for details about the ESD entries produced when you specify the NOGOFF assembler option.

CMS, MVS Refer to Z/OS MVS Program Management: Advanced Facilities,
SA22-7644 for details about the ESD entries produced when you specify the GOFF assembler option.

\section*{Summary of Source and Object Program Structures}
\begin{tabular}{|c|c|c|}
\hline Property & "Load Module" Model & "Program Object" Model \\
\hline Form of object program & One-dimensional module & Two-dimensional module \\
\hline Smallest indivisible independently relocatable component & Control section & Element and part \\
\hline Residence Mode & Only one & One per class \\
\hline Addressing Mode & Only one & One per entry point \\
\hline Compatibility & Can be converted to program object & Can be converted to load module with limitations \\
\hline Assembler Option & NOGOFF or GOFF & GOFF only \\
\hline Assembler statements & CSECT, RSECT, START & CSECT, RSECT, START, CATTR, XATTR \\
\hline Assignable loadable-program attributes & RMODE & RMODE, alignment, load type \\
\hline External symbol types & SD/CM, LD, ER/WX, PR & SD, ED, LD, ER/WX, PR, PD \\
\hline External symbol maximum length & 8 characters & 256 characters \\
\hline External symbol scope & Module (WX), Library (ER) & Section, Module, Library, Import/Export \\
\hline External symbol attributes & AMode, RMode & AMode, RMode, scope, PSect name, linkage type, reference type, extended attributes \\
\hline Object module record types & ESD, TXT, RLD, END, SYM & HDR, ESD, TXT, RLD, END, LEN \\
\hline Address constant types & A, V, Q, CXD & A, V, Q, J, R, CXD \\
\hline Binding attributes & \begin{tabular}{l}
Catenate (SD), \\
Merge-like (CM,PR)
\end{tabular} & Catenate (non-Merge classes), Merge classes (Parts, Pseudo-Registers) \\
\hline Text types & Byte stream & Byte stream, records (structured and unstructured) \\
\hline Maximum contiguous text length & 16MB & 1GB \\
\hline
\end{tabular}

Figure 21. Object Program Structure Comparison

\section*{Addressing}

\section*{Part 2. Machine and Assembler Instruction Statements}
Chapter 4. Machine Instruction Statements ..... 78
General Instructions ..... 78
Decimal Instructions ..... 79
Floating-Point Instructions ..... 79
Control Instructions ..... 79
Input/Output Operations ..... 80
Branching with Extended Mnemonic Codes ..... 80
Alternative Mnemonics for some Branch Relative Instructions ..... 83
Statement Formats ..... 83
Symbolic Operation Codes ..... 84
Operand Entries ..... 85
Registers ..... 86
Register Usage by Machine Instructions ..... 86
Register Usage by System ..... 87
Addresses ..... 87
Implicit Address ..... 88
Explicit Address ..... 88
Relative Address ..... 88
Relocatability of Addresses ..... 89
Machine or Object Code Format ..... 89
Lengths ..... 90
Immediate Data ..... 91
Examples of Coded Machine Instructions ..... 91
RI Format ..... 91
RR Format ..... 93
RS Format ..... 94
RSI Format ..... 95
RX Format ..... 95
SI Format ..... 97
SS Format ..... 97
Chapter 5. Assembler Instruction Statements ..... 100
64-bit Addressing Mode ..... 101
*PROCESS Statement ..... 102
ACONTROL Instruction ..... 103
ADATA Instruction ..... 107
AINSERT Instruction ..... 108
ALIAS Instruction ..... 109
AMODE Instruction ..... 110
CATTR Instruction (MVS and CMS) ..... 112
CCW and CCWO Instructions ..... 115
CCW1 Instruction ..... 116
CEJECT Instruction ..... 118
CNOP Instruction ..... 119
COM Instruction ..... 121
COPY Instruction ..... 122
CSECT Instruction ..... 123
CXD Instruction ..... 125
DC Instruction ..... 126
Rules for DC Operand ..... 128
General Information About Constants ..... 129
Length Attribute Value of Symbols Naming Constants ..... 129
Alignment of Constants ..... 129
Padding and Truncation of Values ..... 130
Padding ..... 131
Truncation ..... 132
Subfield 1: Duplication Factor ..... 132
Subfield 2: Type ..... 133
Subfield 3: Type Extension ..... 134
Subfield 4: Program type ..... 135
Subfield 5: Modifier ..... 136
Length Modifier ..... 136
Scale Modifier ..... 138
Exponent Modifier ..... 139
Subfield 6: Nominal Value ..... 140
Binary Constant-B ..... 141
Character Constant-C ..... 142
Graphic Constant-G ..... 145
Hexadecimal Constant-X ..... 147
Fixed-Point Constants-F and H ..... 148
Decimal Constants-P and Z ..... 151
Address Constants ..... 153
Offset Constant-Q ..... 159
Length Constant-J ..... 160
PSECT Reference-R ..... 160
Hexadecimal Floating-Point Constants-E, EH, D, DH, L, LH, LQ ..... 161
Binary Floating-Point Constants-EB, DB, LB ..... 167
Floating-Point Special Values ..... 170
Literal Constants ..... 171
DROP Instruction ..... 172
DS Instruction ..... 174
How to Use the DS Instruction ..... 176
DSECT Instruction ..... 178
DXD Instruction ..... 180
EJECT Instruction ..... 181
END Instruction ..... 182
ENTRY Instruction ..... 183
EQU Instruction ..... 184
Using Conditional Assembly Values ..... 187
EXITCTL Instruction ..... 187
EXTRN Instruction ..... 189
ICTL Instruction ..... 189
ISEQ Instruction ..... 190
LOCTR Instruction ..... 191
LTORG Instruction ..... 193
Literal Pool ..... 194
Addressing Considerations ..... 195
Duplicate Literals ..... 195
MNOTE Instruction ..... 196
OPSYN Instruction ..... 198
Redefining Conditional Assembly Instructions ..... 200
ORG Instruction ..... 200
POP Instruction ..... 204
PRINT Instruction ..... 204
Process Statement ..... 208
PUNCH Instruction ..... 208
PUSH Instruction ..... 209
REPRO Instruction ..... 210
RMODE Instruction ..... 211
RSECT Instruction ..... 212
SPACE Instruction ..... 213
START Instruction ..... 214
TITLE Instruction ..... 215
Deck ID in Object Records ..... 215
Printing the Heading ..... 216
Printing the TITLE Statement ..... 216
Sample Program Using the TITLE Instruction ..... 216
Page Ejects ..... 217
Valid Characters ..... 217
USING Instruction ..... 218
How to Use the USING Instruction ..... 219
Base Registers for Absolute Addresses ..... 220
Ordinary USING Instruction ..... 220
Range of an Ordinary USING Instruction ..... 222
Domain of an Ordinary USING Instruction ..... 223
Labeled USING Instruction ..... 223
Range of a Labeled USING Instruction ..... 225
Domain of a Labeled USING Instruction ..... 226
Dependent USING Instruction ..... 226
Range of a Dependent USING Instruction ..... 228
Domain of a Dependent USING Instruction ..... 228
WXTRN Instruction ..... 229
XATTR Instruction (MVS and CMS) ..... 230
Association of Code and Data Areas (MVS and CMS) ..... 232

\section*{Chapter 4. Machine Instruction Statements}

This chapter introduces a sample of the more common instruction formats and provides general rules for coding them in their symbolic assembler language format.

For the complete specifications of machine instructions, their object code format, their coding specifications, and their use of registers and virtual storage areas, see the applicable Principles of Operation manual for your processor. If your program requires vector facility instructions, see the applicable Vector Operations manual for the complete specifications of vector-facility instructions.

At assembly time, the assembler converts the symbolic assembler language representation of the machine instructions to the corresponding object code. The computer processes this object code at execution time. Thus, the functions described in this section can be called execution-time functions.

Also at assembly time, the assembler creates the object code of the data constants and reserves storage for the areas you specify in your data definition assembler instructions, such as DC and DS (see Chapter 5, "Assembler Instruction Statements"). At execution time, the machine instructions can refer to these constants and areas, but the constants themselves are not normally processed.

As defined in the applicable Principles of Operation manual, there are five categories of machine instructions:
- General instructions
- Decimal instructions
- Floating-Point instructions
- Control instructions
- Input/Output operations

Each is discussed in the following sections.

\section*{General Instructions}

Use general instructions to manipulate data that resides in general registers or in storage, or that is introduced from the instruction stream. General instructions include fixed-point, logical, and branching instructions. In addition, they include unprivileged status-switching instructions. Some general instructions operate on data that resides in the PSW or the TOD clock.

The general instructions treat data as four types: signed binary integers, unsigned binary integers, unstructured logical data, and decimal data. Data is treated as decimal by the conversion, packing, and unpacking instructions.

For further information, see "General Instructions" in the applicable Principles of Operation manual.

\section*{Decimal Instructions}

Use the decimal instructions when you want to do arithmetic and editing operations on data that has the binary equivalent of decimal representation.

Decimal data may be represented in either zoned or packed format. In the zoned format, the rightmost four bits of a byte are called the numeric bits and normally consist of a code representing a decimal digit. The leftmost four bits of a byte are called the zone bits, except for the rightmost byte of a decimal operand, where these bits may be treated either as a zone or as a sign.

In the packed format, each byte contains two decimal digits, except for the rightmost byte, which contains a sign to the right of a decimal digit.

Decimal instructions treat all numbers as integers. For example, 3.14, 31.4, and 314 are all processed as 314. You must keep track of the decimal point yourself. The integer and scale attributes discussed in "Data Attributes" on page 324 can help you do this.

Additional operations on decimal data are provided by several of the instructions in "General Instructions" in the applicable Principles of Operation manual. Decimal operands always reside in storage.

For further information, see "Decimal Instructions" in the applicable Principles of Operation manual.

\section*{Floating-Point Instructions}

Use floating-point instructions when you want to do arithmetic operations on data in the floating-point representation. Thus, you do not have to keep track of the decimal point in your computations. Floating-point instructions also let you do arithmetic operations on both very large numbers and very small numbers, usually providing greater precision than fixed-point decimal instructions.

For further information, see "Floating-Point Instructions" in the applicable Principles of Operation manual.

\section*{Control Instructions}

Control instructions include all privileged and semiprivileged machine instructions, except the input/output instructions described on page 80 .

Privileged instructions may be processed only when the processor is in the supervisor state. An attempt to process an installed privileged instruction in the problem state generates a privileged-operation exception.

Semiprivileged instructions are those instructions that can be processed in the problem state when certain authority requirements are met. An attempt to process an installed semiprivileged instruction in the problem state when the authority requirements are not met generates a privileged-operation exception or some other program-interruption condition depending on the particular requirement that is violated.

\section*{Input/Output Operations}

For further details, see "Control Instructions" in the applicable Principles of Operation manual.

\section*{Input/Output Operations}

Use the input/output instructions (instead of the IBM-supplied system macro instructions) when you want to control your input and output operations more closely.

The input or output instructions let you identify the channel or the device on which the input or output operation is to be done. For information about how and when you can use these instructions, see the applicable system manual.

For more information, see "Input/Output Operations" in the applicable Principles of Operation manual and the applicable system manuals.

\section*{Branching with Extended Mnemonic Codes}

Branch instructions let you specify an extended mnemonic code for the condition on which a branch is to occur. Thus, you avoid having to specify the mask value, that represents the condition code, required by the \(B C, B C R\), and \(B R C\) machine instructions. The assembler translates the extended mnemonic code into the mask value, and then assembles it into the object code of the \(B C, B C R\), or \(B R C\) machine instruction.

The extended branch mnemonics for the BC instruction require a base register; the extended mnemonics for the BCR and BRC instructions do not. The extended mnemonics for the BRC instruction begin with the letter "J," and are sometimes called "Jump" instructions, as indicated in Figure 22.

The extended mnemonic codes are given in Figure 22 on page 81. They can be used as operation codes for branching instructions, replacing the BC, BCR, and BRC machine instruction codes (see 1 in Figure 22). Note that the first operand (see 2 in Figure 22) of the BC, BCR, and BRC instructions must not be present in the operand field (see 3 in Figure 22) of the extended mnemonic branching instructions.


Figure 22 (Part 1 of 3). Extended Mnemonic Codes

Branching with Extended Mnemonic Codes

\section*{Used After Test Under Mask Instructions}
\begin{tabular}{|c|c|c|c|c|c|}
\hline B0 & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Ones & RX & BC & 1, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BOR & R2 & & RR & BCR & 1, R2 \\
\hline BM & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Mixed & RX & BC & 4, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BMR & R2 & & RR & BCR & 4, R2 \\
\hline BZ & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Zero & RX & BC & 8, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BZR & R2 & & RR & BCR & 8, R2 \\
\hline BNO & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Not Ones & RX & BC & 14, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BNOR & R2 & & RR & BCR & 14, R2 \\
\hline BNM & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Not Mixed & RX & BC & 11, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BNMR & R2 & & RR & BCR & 11, R2 \\
\hline BNZ & \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) & Branch if Not Zero & RX & BC & 7, \(\mathrm{D}_{2}\left(\mathrm{X}_{2}, \mathrm{~B}_{2}\right)\) \\
\hline BNZR & R2 & & RR & BCR & 7, R2 \\
\hline
\end{tabular}
| Branch Relative on Condition Long
\begin{tabular}{|c|c|c|c|c|c|}
\hline BRUL & label & Unconditional Br Rel Long & RIL & BRCL & 15, 1 abel \\
\hline BRHL & label & Br Rel Long on High & RIL & BRCL & 2,1abel \\
\hline BRLL & label & Br Rel Long on Low & RIL & BRCL & 4,1abel \\
\hline BREL & label & Br Rel Long on Equal & RIL & BRCL & 8,1abel \\
\hline BRNHL & label & Br Rel Long on Not High & RIL & BRCL & 13, label \\
\hline BRNLL & label & Br Rel Long on Not Low & RIL & BRCL & 11, 1 abel \\
\hline BRNEL & label & Br Rel Long on Not Equal & RIL & BRCL & 7,label \\
\hline BRPL & label & Br Rel Long on Plus & RIL & BRCL & 2,1abel \\
\hline BRML & label & Br Rel Long on Minus & RIL & BRCL & 4,1abel \\
\hline BRZL & label & Br Rel Long on Zero & RIL & BRCL & 8,1abel \\
\hline BROL & label & Br Rel Long on Overflow & RIL & BRCL & 1, 1 abel \\
\hline BRNPL & label & Br Rel Long on Not Plus & RIL & BRCL & 13, 1 abe 1 \\
\hline BRNML & label & Br Rel Long on Not Minus & RIL & BRCL & 11,1abel \\
\hline BRNZL & label & Br Rel Long on Not Zero & RIL & BRCL & 7,label \\
\hline BRNOL & label & Br Rel Long on Not Overflow & RIL & BRCL & 14,1abel \\
\hline Branch & \multicolumn{5}{|l|}{Relative on Condition} \\
\hline BRO & label & Branch on Overflow & RI & BRC & 1, label \\
\hline BRP & label & Branch on Plus & RI & BRC & 2,1abel \\
\hline BRH & label & Branch on High & RI & BRC & 2,1abel \\
\hline BRL & label & Branch on Low & RI & BRC & 4, label \\
\hline BRM & label & Branch on Minus & RI & BRC & 4,label \\
\hline BRNE & label & Branch on Not Equal & RI & BRC & 7,label \\
\hline BRNZ & label & Branch on Not Minus & RI & BRC & 7,label \\
\hline BRE & label & Branch on Equal & RI & BRC & 8,1abel \\
\hline BRZ & label & Branch on Zero & RI & BRC & 8, label \\
\hline BRNL & label & Branch on Not Low & RI & BRC & 11, 1 abel \\
\hline BRNM & label & Branch on Not Minus & RI & BRC & 11, 1 abel \\
\hline BRNH & label & Branch on Not High & RI & BRC & 13, 1 abe 1 \\
\hline BRNP & label & Branch on Not Plus & RI & BRC & 13, 1 abel \\
\hline BRNO & label & Branch on No Overflow & RI & BRC & 14, 1 abel \\
\hline BRU & label & Unconditional Branch & RI & BRC & 15, 1 abel \\
\hline
\end{tabular}

Figure 22 (Part 2 of 3). Extended Mnemonic Codes

\section*{| Jump on Condition Long}
\begin{tabular}{llllll} 
JLU & label & Unconditional Jump Long & RIL & BRCL & 15, label \\
JLNOP & label & No operation & RIL & BRCL 0,1 label
\end{tabular}

Notes:
1. \(D_{2}=\) displacement, \(X_{2}=\) index register, \(B_{2}=\) base register, \(R_{2}=\) register containing branch address
2. The addresses represented are explicit address (see 4 ). However, implicit addresses can also be used in this type of instruction.
3. Avoid using BM, BNM, JM, and JNM after the TMH or TML instruction.

Figure 22 (Part 3 of 3). Extended Mnemonic Codes

\section*{Alternative Mnemonics for some Branch Relative Instructions}

For some branch relative statements, there are alternative mnemonics. These are:
\begin{tabular}{lll} 
Instruction & Alternative & Description \\
\hline BRAS & JAS & Branch Relative and Save \\
BRASL & JASL & Branch Relative and Save Long \\
BRCT & JCT & Branch Relative on Count \\
BRCTG & JCTG & Branch Relative on Count \\
BRXH & JXH & Branch Relative on Index High \\
BRXHG & JXHG & Branch Relative on Index High \\
BRXLE & JXLE & Branch Rel. on Index Low or Equal \\
BRXLG & JXLEG & Branch Rel. on Index Low or Equal
\end{tabular}

\section*{Statement Formats}

Machine instructions are assembled into 2, 4, or 6 bytes of object code according to the format of each instruction. Machine instruction formats include the following (ordered by length attribute):
```

Length Attribute Basic Formats
2 RR
4 RI, RS, RSI, RX, SI
6 SS

```

See the applicable Principles of Operation manual for complete details about machine instruction formats. See also "Examples of Coded Machine Instructions" on page 91

When you code machine instructions, you use symbolic formats that correspond to the actual machine language formats. Within each basic format, you can also code
variations of the symbolic representation, divided into groups according to the basic formats shown below.

The assembler converts only the operation code and the operand entries of the assembler language statement into object code. The assembler assigns to a name entry symbol the value of the address of the first byte of the assembled instruction. When you use this same symbol in the operand of an assembler language statement, the assembler uses this address value in converting the symbolic operand into its object code form. The length attribute assigned to the symbol depends on the basic machine language format of the instruction in which the symbol appears as a name entry.

A remarks entry is not converted into object code.
An example of a typical assembler language statement follows:
```

LABEL L 4,256(5,10) LOAD INTO REG4

```
where:
```

LABEL is the name entry
L is the operation code mnemonic (converted to hex 58)
4 is the register operand (converted to hex 4)
256(5,10) are the storage operand entries (converted to hex 5A100)
LOAD INTO REG4 are remarks not converted into object code

```

The object code of the assembled instruction, in hexadecimal, is:
```

5845A100 (4 bytes in RX format)

```

\section*{Symbolic Operation Codes}

You must specify an operation code for each machine instruction statement. The symbolic operation code, or mnemonic code as it is also called, indicates the type of operation to be done; for example, A indicates the addition operation. See the applicable Principles of Operation for a complete list of symbolic operation codes and the formats of the corresponding machine instructions.

The general format of the machine instruction operation code is:


Verb: The verb must always be present. It usually consists of one or two characters and specifies the operation to be done. The verb is underscored in the following examples:
\begin{tabular}{lll} 
A & 3, AREA & A indicates an add operation \\
\(\underline{M V C}\) & TO, FROM & MV indicates a move operation
\end{tabular}

The other items in the operation code are not always present. They include the following (underscores are used to indicate modifiers, data types, and machine formats in the examples below):

Modifier: Modifier, which further defines the operation:
\[
\text { AL 3,AREA } \quad L \text { indicates a logical operation }
\]

Data Type: Type qualifier, which indicates the type of data used by the instruction in its operation:
\begin{tabular}{ll} 
CVㅡB 3, BINAREA & B indicates binary data \\
MVC TO, FROM & C indicates character data \\
AE 2,FLTSHRT & \begin{tabular}{l} 
E indicates normalized short \\
floating-point data
\end{tabular} \\
AD 2,FLTLONG & \begin{tabular}{l} 
D indicates normalized long \\
floating-point data
\end{tabular}
\end{tabular}

Machine Format: Format qualifier, R indicating a register operand, or I indicating an immediate operand. For example:
\begin{tabular}{ll} 
ADR 2,4 & \(R\) indicates a register operand \\
MVI FIELD, X'A1' & I indicates an immediate operand \\
AH \(\underline{\text { 7,123 }}\) &
\end{tabular}

\section*{Operand Entries}

You may specify one or more operands in each machine instruction statement to provide the data or the location of the data upon which the machine operation is to be done. The operand entries consist of one or more fields or subfields, depending on the format of the instruction being coded. They can specify a register, an address, a length, or immediate data. You can omit length fields or subfields, which the assembler computes for you from the other operand entries. You can code an operand entry either with symbols or with self-defining terms.

The rules for coding operand entries are:
- A comma must separate operands.
- Parentheses must enclose subfields.
- A comma must separate subfields enclosed in parentheses.
- If a subfield is omitted because it is implicit in a symbolic address, the parentheses that would have enclosed the subfield must be omitted.
- If two subfields are enclosed in parentheses and separated by commas, the following applies:
- If both subfields are omitted because they are implicit in a symbolic entry, the separating comma and the parentheses that would have been needed must also be omitted.
- If the first subfield is omitted, the comma that separates it from the second subfield must be written, as well as the enclosing parentheses.
- If the second subfield is omitted, the comma that separates it from the first subfield must be omitted; however, the enclosing parentheses must be written.
- Spaces must not appear within the operand field, except as part of a character self-defining term, or in the specification of a character literal.

\section*{Registers}

You can specify a register in an operand for use as an arithmetic accumulator, a base register, an index register, and as a general depository for data to which you want to refer repeatedly.

You must be careful when specifying a register whose contents have been affected by the execution of another machine instruction, the control program, or an IBM-supplied system macro instruction.

For some machine instructions, you are limited in which registers you can specify in an operand.

The expressions used to specify registers must have absolute values; in general, registers 0 through 15 can be specified for machine instructions. However, the following restrictions on register usage apply:
- If the NOAFPR assembler option is specified, then only the floating-point registers \((0,2,4\), or 6\()\) may be specified for floating-point instructions.
- The even-numbered registers \((0,2,4,6,8,10,12,14)\) must be specified for the following groups of instructions:
- The double-shift instructions
- The fullword multiply and divide instructions
- The move long and compare logical long instructions
- If the AFPR assembler option is specified, then one of the floating-point registers \(0,1,4,5,8,9,12\) or 13 can be specified for the instructions that use extended floating-point data in pairs of registers, such as AXR, SXR, LTXBR, and SQEBR.
- If the NOAFPR assembler option is specified, then either floating-point register 0 or 4 must be specified for these instructions.
- For a processor with a vector facility, the even-numbered vector registers (0, 2, \(4,6,8,10,12,14)\) must be specified in vector-facility instructions that are used to manipulate long floating-point data or 64-bit signed binary data in vector registers.

The assembler checks the registers specified in the instruction statements of the above groups. If the specified register does not comply with the stated restrictions, the assembler issues a diagnostic message and does not assemble the instruction. Binary zeros are generated in place of the machine code.

\section*{Register Usage by Machine Instructions}

Registers that are not explicitly coded in symbolic assembler language representation of machine instructions, but are nevertheless used by assembled machine instructions, are divided into two categories:
- Base registers that are implicit in the symbolic addresses specified. (See "Addresses" on page 87) The registers can be identified by examining the object code or the USING instructions that assign base registers for the source module.
- Registers that are used by machine instructions, but don't appear in assembled object code.
- For double shift and fullword multiply and divide instructions, the odd-numbered register, whose number is one greater than the even-numbered register specified as the first operand.
- For Move Long and Compare Logical Long instructions, the odd-numbered registers, whose number is one greater than even-numbered registers specified in the two operands.
- For Branch on Index High (BXH) and the Branch on Index Low or Equal (BXLE) instructions, if the register specified for the second operand is an even-numbered register, the next higher odd-numbered register is used to contain the value to be used for comparison.
- For Load Multiple (LM, LAM) and Store Multiple (STM, STAM) instructions, the registers that lie between the registers specified in the first two operands.
- For extended-precision floating point instructions, the second register of the register pair.
- For Compare and Form Codeword (CFC) instruction, registers 1, 2 and 3 are used.
- For Translate and Test (TRT) instruction, registers 1 and 2 are used.
- For Update Tree (UPT) instruction, registers 0-5 are used.
- For Edit and Mark (EDMK) instruction, register 1 is used.
- For certain control instructions, one or more of registers 0-4 and register 14 are used. See "Control Instructions" in the applicable Principles of Operation manual.
- For certain input/output instructions, either or both registers 1 and 2 are used. See "Input/Output Instructions" in the applicable Principles of Operation manual.
- On a processor with a vector facility:
1. For instructions that manipulate long floating-point data in vector registers, the odd-numbered vector registers, whose number is one greater than the even-numbered vector registers specified in each operand.
2. For instructions that manipulate 64-bit signed binary data in vector registers, the odd-numbered vector registers, whose number is one greater than the even-numbered vector registers specified in each operand.

\section*{Register Usage by System}

The programming interface of the system control programs uses registers \(0,1,13\), 14 , and 15.

\section*{Addresses}

You can code a symbol in the name field of a machine instruction statement to represent the address of that instruction. You can then refer to the symbol in the operands of other machine instruction statements. The object code requires that addresses be assembled in a numeric relative-offset or base-displacement format. This format lets you specify addresses that are relocatable or absolute Chapter 3,
"Program Structures and Addressing" on page 50 describes how you use symbolic addresses to refer to data in your assembler language program.

Defining Symbolic Addresses: Define relocatable addresses by either using a symbol as the label in the name field of an assembler language statement, or equating a symbol to a relocatable expression.

Define absolute addresses (or values) by equating a symbol to an absolute expression.

Referring to Addresses: You can refer to relocatable and absolute addresses in the operands of machine instruction statements. (Such address references are also called addresses in this manual.) The two ways of coding addresses are:
- Implicitly-in a form that the assembler must first convert into an explicit relative-offset or base-displacement form before it can be assembled into object code.
- Explicitly—in a form that can be directly assembled into object code.

\section*{Implicit Address}

An implicit address is specified by coding one expression. The expression can be relocatable or absolute. The assembler converts all implicit addresses into their relative-offset or base-displacement form before it assembles them into object code. The assembler converts implicit addresses into explicit base-displacement addresses only if a USING instruction has been specified, or for small absolute expressions, where the address is resolved without a USING. The USING instruction assigns both a base address, from which the assembler computes displacements, and a base register, which is assumed to contain the base address. The base register must be loaded with the correct base address at execution time. For more information, refer to "Addressing" on page 63 .

\section*{Explicit Address}

An explicit address is specified by coding two absolute expressions as follows:
- The first is an absolute expression for the displacement, whose value must lie in the range 0 through 4095 (4095 is the maximum value that can be represented by the 12 binary bits available for the displacement in the object code).
- The second (enclosed in parentheses) is an absolute expression for the base register, whose value must lie in the range 0 through 15.

An explicit base register designation must not accompany an implicit address. However, in RX-format instructions, an index register can be coded with an implicit address as well as with an explicit address. When two addresses are required, each address can be coded as an explicit address or as an implicit address.

\section*{Relative Address}

A relative address is specified by coding one expression. The expression may be relocatable or absolute. If a relocatable expression is used, then the assembler converts the value to a signed number of halfwords relative to the current location counter, and then uses that value in the object code. An absolute value may be used for a relative address, but the assembler issues a warning message, as it uses the supplied value, and this may cause unpredictable results.

\section*{Relocatability of Addresses}

If the value of an address expression changes when the assumed origin of the program is changed, and changes by the same amount, then the address is simply relocatable. If the addressing expression does not change when the assumed origin of the program is changed, then that address is absolute. If the addressing expression changes by some other amount, the address may be complexly relocatable.

Addresses in the relative-offset or base-displacement form are relocatable, because:
- Each relocatable address is assembled as a signed relative offset from the instruction, or as a displacement from a base address and a base register.
- The base register contains the base address.
- If the object module assembled from your source module is relocated, only the contents of the base register need reflect this relocation. This means that the location in virtual storage of your base has changed, and that your base register must contain this new base address.
- Addresses in your program have been assembled as relative to the base address; therefore, the sum of the displacement and the contents of the base register point to the correct address after relocation.

Absolute addresses are also assembled in the base-displacement form, but always indicate a fixed location in virtual storage. This means that the contents of the base register must always be a fixed absolute address value regardless of relocation.

\section*{Machine or Object Code Format}

Addresses assembled into the object code of machine instructions have the format given in Figure 23 on page 90 . Not all of the instruction formats are shown in Figure 23.

The addresses represented have a value that is the sum of a displacement (see \(\mathbf{1}\) in Figure 23) and the contents of a base register (see \(\mathbf{2}\) in Figure 23).

Index Register: In RX-format instructions, the address represented has a value that is the sum of a displacement, the contents of a base register, and the contents of an index register (see 3 in Figure 23).

Operand Entries


Figure 23. Format of Addresses in Object Code

\section*{Lengths}

You can specify the length field in an SS-format instruction. This lets you indicate explicitly the number of bytes of data at a virtual storage location that is to be used by the instruction. However, you can omit the length specification, because the assembler computes the number of bytes of data to be used from the expression that represents the address of the data.

See page 97 for more information about SS-format instructions.
Implicit Length: When a length subfield is omitted from an SS-format machine instruction, an implicit length is assembled into the object code of the instruction. The implicit length is either of the following:
- For an implicit address, it is the length attribute of the first or only term in the expression representing the implicit address.
- For an explicit address, it is the length attribute of the first or only term in the expression representing the displacement.

Explicit Length: When a length subfield is specified in an SS-format machine instruction, the explicit length always overrides the implicit length.

An implicit or explicit length is the effective length. The length value assembled is
always one less than the effective length. If you want an assembled length value of 0 , an explicit length of 0 or 1 can be specified.

In the SS-format instructions requiring one length value, the allowable range for explicit lengths is 0 through 256. In the SS-format instructions requiring two length values, the allowable range for explicit lengths is 0 through 16.

\section*{Immediate Data}

In addition to registers, numeric values, relative addresses, and lengths, some machine instruction operands require immediate data. Such data is assembled directly into the object code of the machine instructions. Use immediate data to specify the bit patterns for masks or other absolute values you need.

Specify immediate data only where it is required. Do not confuse it with address references to constants and areas, or with any literals you specify as the operands of machine instructions.

Immediate data must be specified as absolute expressions whose range of values depends on the machine instruction for which the data is required. The immediate data is assembled into its binary representation.

\section*{Examples of Coded Machine Instructions}

The examples that follow are grouped according to machine instruction format, and the groups are shown in order of the instruction length. They show the various ways in which you can code the operands of machine instructions. Both symbolic and numeric representation of fields and subfields are shown in the examples. Therefore, assume that all symbols used are defined elsewhere in the same source module.

The object code assembled from at least one coded statement per group is also included. A complete summary of machine instruction formats with the coded assembler language variants can be found in the applicable Principles of Operation manual.

The examples that follow show the various instruction formats, and are not meant to show how the machine instructions should be used.

\section*{RI Format}

The operand fields of RI-format instructions designate a register and an immediate operand, with the following exception:
- In BRC branching instructions, a 4-bit branching mask with a value between 0 and 15 inclusive replaces the register designation.

Symbols used to represent registers (such as REG1 in the example) are assumed to be equated to absolute values between 0 and 15. The 16-bit immediate operand has two different interpretations, depending on whether the instruction is a branching instruction or not.

There are two types of non-branching RI-format instructions.
- For most, the immediate value is treated as a signed binary integer (a value between -32768 and +32767 ). This value may be specified using self-defining terms or equated symbols.
\begin{tabular}{|l|l|l|lll|}
\hline Op Code & R1 & OpCd & \(\mathrm{I}_{2}\) & \\
\hline 0 & 8 & 12 & 16 & & 31
\end{tabular}
- For logical instructions such as TMH, the immediate field is a 16 bit mask.
\begin{tabular}{|l|l|l|ll|}
\hline Op Code & \(M_{1}\) & OpCd & \(I_{2}\) & \\
\hline
\end{tabular}

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & AHI & REG1,2000 \\
ALPHA2 & MHI & 3,1234 \\
BETA1 & TMH & \(7, X^{\prime} 80011^{\prime}\)
\end{tabular}

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

A7708001
where:
A7.0 is the operation code
7 is register \(R_{1}\)
8001 is the immediate data I2
For branching RI-format instructions, the immediate value is treated as a signed binary integer representing the number of halfwords to branch relative to the current location.

The branch target may be specified as a relocatable expression, in which case the assembler performs some checking, and calculates the immediate value.

The branch target may also be specified as an absolute value in which case the assembler issues a warning before it assembles the instruction.

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & BRAS & 1,BETA1 \\
ALPHA2 & BRC & 3,ALPHA1 \\
BETA1 & BRCT & 7,ALPHA1
\end{tabular}

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

A776FFFC
where:
A7. 6 is the operation code
7 is register \(R_{1}\)
FFFC is the immediate data I2; a value of -4 decimal
If the GOFF assembler option is active, then it is possible to specify the target address as one or more external symbols (with or without offsets).

If an offset is specified it may be specified as a relocatable expression or an absolute value. If the offset is specified as a relocatable expression, the assembler performs some checking and calculates the immediate value. If the offset is an absolute expression the assembler issues warning message ASMA056W.

Examples:
\begin{tabular}{lll} 
ALPHA1 BRAS & \(14, A-B+C+10\) \\
ALPHA2 & wRASL & \(14, A-B+C+10\) \\
BETA1 \(B R C\) & \(15, A-B+C+10\)
\end{tabular}

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

A7F40005
where:
A7.4 is the operation code
\(F \quad\) is the condition code 0005 is the immediate data I2; a value of 5 decimal.

In addition GOFF Relocation Dictionary Data Items are generated for the external symbols \(A, B\) and \(C\).

\section*{RR Format}

The operand fields of RR-format instructions designate two registers, with the following exceptions:
- In BCR branching instructions, when a 4-bit branching mask replaces the first register specification (see 8 in the instruction labeled GAMMA1 below)
- In SVC instructions, where an immediate value (between 0 and 255) replaces both registers (see 200 in the instruction labeled DELTA1 below)


Symbols used to represent registers in RR-format instructions (see INDEX and REG2 in the instruction labeled ALPHA2 below) are assumed to be equated to absolute values between 0 and 15 .

Symbols used to represent immediate values in SVC instructions (see TEN in the instruction labeled DELTA2 below) are assumed to be equated to absolute values between 0 and 255.

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & LR & 1,2 \\
ALPHA2 & LR & INDEX, REG2 \\
GAMMA1 & BCR & 8,12 \\
DELTA1 & SVC & 200 \\
DELTA2 & SVC & TEN
\end{tabular}

When assembled, the object code of the instruction labeled ALPHA1, in hexadecimal, is:
where:
18 is the operation code
1 is register \(R_{1}\)
2 is register \(R_{2}\)

\section*{RS Format}

The operand fields of RS-format instructions designate two registers, and a virtual storage address (coded as an implicit address or an explicit address).
\begin{tabular}{|l|l|l|l|l|l|}
\hline Op Code & \(R_{1}\) & \(R_{3}\) & \(B_{2}\) & \(D_{2}\) & \\
\hline
\end{tabular}

In the Insert Characters under Mask (ICM) and the Store Characters under Mask (STCM) instructions, a 4-bit mask (see X'E' and MASK in the instructions labeled DELTA1 and DELTA2 below), with a value between 0 and 15, replaces the second register specifications.
\begin{tabular}{|l|c|c|c|l|l|}
\hline Op Code & \(R_{1}\) & \(M_{3}\) & \(B_{2}\) & & \(D_{2}\) \\
\hline
\end{tabular}

Symbols used to represent registers (see REG4, REG6, and BASE in the instruction labeled ALPHA2 below) are assumed to be equated to absolute values between 0 and 15.

Symbols used to represent implicit addresses (see AREA and IMPLICIT in the instructions labeled BETA1 and DELTA2 below) can be either relocatable or absolute.

Symbols used to represent displacements (see DISPL in the instruction labeled BETA2 below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & LM & \(4,6,20(12)\) \\
ALPHA2 & LM & REG4, REG6,20(BASE) \\
BETA1 & STM & 4,6, AREA \\
BETA2 & STM & 4,6, DISPL(BASE) \\
GAMMA1 & SLL & 2,15 \\
DELTA1 & ICM & \(3, X^{\prime} E^{\prime}, 1024(10)\) \\
DELTA2 & ICM & REG3,MASK, IMPLICIT
\end{tabular}

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:
9846C014
where:
98 is the operation code
4 is register \(R_{1}\)
6 is register \(R_{3}\)
\(C\) is base register \(B_{1}\)
014 is displacement \(D_{1}\) from base register \(B_{1}\)

When assembled, the object code for the instruction labeled DELTA1, in hexadecimal, is:
BF3EA400
where:
\begin{tabular}{ll} 
BF & is the operation code \\
3 & is register \(R_{1}\) \\
\(E\) & is mask \(M_{3}\) \\
A & is base register \(B_{1}\) \\
400 & is displacement \(D_{1}\) from base register \(B_{1}\)
\end{tabular}

\section*{RSI Format}

The operand fields of RSI-format instructions designate two registers and a 16-bit immediate operand.
\begin{tabular}{l}
\begin{tabular}{|l|c|c|c|c|}
\hline Op Code & \(R_{1}\) & \(R_{3}\) & \(I_{2}\) & \\
\hline
\end{tabular} \\
\hline
\end{tabular}

Symbols used to represent registers (See REG1 below) are assumed to be equated to absolute values between 0 and 15.

The immediate value is treated as a signed binary integer representing the number of halfwords to branch relative to the current location.

The branch target may be specified as a label in which case the assembler calculates the immediate value and performs some checking of the value.

The branch target may also be specified as an absolute value in which case the assembler issues a warning before it assembles the instruction.

\section*{Examples:}
```

ALPHA1 BRXH
REG1,REG3,BETA1
BETA1 BRXLE
1,2,ALPHA1

```

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is
84130002
where:
```

84 is the operation code
1 is register REG1
3 is register REG3
0002 is the immediate data I2

```

The operand fields of RX-format instructions designate one or two registers, including an index register, and a virtual storage address (coded as an implicit address or an explicit address), with the following exception:

In BC branching instructions, a 4-bit branching mask (see 7 and TEN in the instructions labeled LAMBDAn below) with a value between 0 and 15, replaces the first register specification.
\begin{tabular}{|l|l|l|l|l|l|}
\hline Op Code & \(R_{1}\) & \(X_{2}\) & \(B_{2}\) & \(D_{2}\) & \\
\hline 0 & 8 & 12 & 16 & 20 & 31
\end{tabular}

Symbols used to represent registers (see REG1, INDEX, and BASE in the ALPHA2 instruction below) are assumed to be equated to absolute values between 0 and 15.

Symbols used to represent implicit addresses (see IMPLICIT in the instructions labeled GAMMAn below) can be either relocatable or absolute.

Symbols used to represent displacements (see DISPL in the instructions labeled BETA2 and LAMBDA1 below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

\section*{Examples:}
\begin{tabular}{|c|c|c|}
\hline ALPHA1 & L & 1,200(4,10) \\
\hline ALPHA2 & L & REG1,200(INDEX, BASE) \\
\hline BETA1 & L & 2,200(,10) \\
\hline BETA2 & L & REG2,DISPL(,BASE) \\
\hline GAMMA1 & L & 3,IMPLICIT \\
\hline GAMMA2 & L & 3,IMPLICIT(INDEX) \\
\hline DELTA1 & L & 4, =F'33' \\
\hline LAMBDA1 & BC & 7,DISPL(,BASE) \\
\hline LAMBDA2 & BC & TEN,ADDRESS \\
\hline
\end{tabular}

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:

5814A0C8
where:
58 is the operation code
1 is register \(R_{1}\)
4 is index register \(X_{2}\)
\(A \quad\) is base register \(B_{2}\)
\(0 C 8\) is displacement \(D_{2}\) from base register \(B_{2}\)
When assembled, the object code for the instruction labeled GAMMA1, in hexadecimal, is:
5824xyyy
where:
58 is the operation code
2 is register \(R_{1}\)
4 is the index register \(X_{2}\)
\(x \quad\) is base register \(B_{2}\)
yyy is displacement \(D_{2}\) from base register \(B_{2}\)

\section*{SI Format}

The operand fields of SI-format instructions designate immediate data and a virtual storage address.
\begin{tabular}{|l|l|l|ll|}
\hline Op Code & & \(I_{2}\) & \(\mathrm{~B}_{1}\) & \\
\hline \multicolumn{6}{|c|}{} & \(\mathrm{D}_{1}\) & \\
\hline 0 & 8 & & 16 & 20 \\
\hline
\end{tabular}

Symbols used to represent immediate data (see HEX40 and TEN in the instructions labeled ALPHA2 and BETA1 below) are assumed to be equated to absolute values between 0 and 255.

Symbols used to represent implicit addresses (see IMPLICIT and KEY in the instructions labeled BETA1 and BETA2) can be either relocatable or absolute.

Symbols used to represent displacements (see DISPL40 in the instruction labeled ALPHA2 below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & CLI & \(40(9), X^{\prime} 40^{\prime}\) \\
ALPHA2 & CLI & DISPL40(NINE), HEX40 \\
BETA1 & CLI & IMPLICIT,TEN \\
BETA2 & CLI & KEY,C'E'
\end{tabular}

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:
95409028
where
95 is the operation code.
40 is the immediate data.
9 is the base register.
028 is the displacement from the base register

\section*{SS Format}

The operand fields and subfields of SS-format instructions designate two virtual storage addresses (coded as implicit addresses or explicit addresses) and, optionally, the explicit data lengths you want to include. However, note that, in the Shift and Round Decimal (SRP) instruction, a 4-bit immediate data field (see 3 in SRP instruction below), with a value between 0 and 9 , is specified as a third operand.


Symbols used to represent base registers (see BASE8 and BASE7 in the instruction labeled ALPHA2 below) in explicit addresses are assumed to be equated to absolute values between 0 and 15 .

Symbols used to represent explicit lengths (see NINE and SIX in the instruction labeled ALPHA2 below) are assumed to be equated to absolute values between 0 and 256 for SS-format instructions with one length specification, and between 0 and 16 for SS-format instructions with two length specifications.

Symbols used to represent implicit addresses (see FIELD1 and FIELD2 in the instruction labeled ALPHA3, and FIELD1, X'8' in the SRP instructions below) can be either relocatable or absolute.

Symbols used to represent displacements (see DISP40 and DISP30 in the instruction labeled ALPHA5 below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

See page 90 for more information about the lengths of SS-format instructions.

\section*{Examples:}
\begin{tabular}{lll} 
ALPHA1 & AP & \(40(9,8), 30(6,7)\) \\
ALPHA2 & AP & \(40(\) NINE,BASE8), \(30(\) SIX, BASE7 \()\) \\
ALPHA3 & AP & FIELD1,FIELD2 \\
ALPHA4 & AP & AREA(9),AREA2(6) \\
ALPHA5 & AP & DISP40(,8), DISP30(,7) \\
BETA1 & MVC & \(0(80,8), 0(7)\) \\
BETA2 & MVC & DISP0(,8), DISP0(7) \\
BETA3 & MVC & TO,FROM \\
& SRP & FIELD1, ' \(^{\prime}, 3\)
\end{tabular}

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:

\section*{Examples of Coded Machine Instructions}
where:
FA is the operation code.
8 is length \(L_{1}\)
5 is length \(L_{2}\)
8 is base register \(B_{1}\)
028 is displacement \(D_{1}\) from base register \(B_{1}\)
7 is base register \(B_{2}\)
01 E is displacement \(D_{2}\) from base register \(B_{2}\)
When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is:

D24F80007000
where:
D2 is the operation code
4 F is length \(L\)
8 is base register \(B_{1}\)
000 is displacement \(D_{1}\) from base register \(B_{1}\)
7 is base register \(B_{2}\)
000 is displacement \(D_{2}\) from base register \(B_{2}\)

\section*{Chapter 5. Assembler Instruction Statements}

This chapter describes, in detail, the syntax and usage rules of each assembler instruction. There is also information about assembly instructions on "Conditional Assembly Instructions" on page 256 The following table lists the assembler instructions by type, and provides the page number where the instruction is described.

Figure 24 (Page 1 of 2). Assembler Instructions
\begin{tabular}{llr|}
\hline Type of Instruction & Instruction & Page No. \\
\hline Program Control & AINSERT & 108 \\
\cline { 2 - 3 } & CNOP & 119 \\
\cline { 2 - 3 } & COPY & 122 \\
\cline { 2 - 3 } & END & 182 \\
\hline & EXITCTL & 187 \\
\hline & ICTL & 189 \\
\hline & ISEQ & 190 \\
\hline & ORG & 193 \\
\hline & POP & 200 \\
\hline & PUNCH & 204 \\
\hline & PUSH & 208 \\
\hline & REPRO & 209 \\
\hline Operation Code Definition & CEJECT & 210 \\
\hline
\end{tabular}

Figure 24 (Page 2 of 2). Assembler Instructions
\begin{tabular}{|c|c|c|}
\hline Type of Instruction & Instruction & Page No. \\
\hline \multirow[t]{16}{*}{Program Section and Linking} & ALIAS & 109 \\
\hline & AMODE & 110 \\
\hline & CATTR (MVS and CMS) & 112 \\
\hline & COM & 121 \\
\hline & CSECT & 123 \\
\hline & CXD & 125 \\
\hline & DSECT & 178 \\
\hline & DXD & 180 \\
\hline & ENTRY & 183 \\
\hline & EXTRN & 189 \\
\hline & LOCTR & 191 \\
\hline & RMODE & 211 \\
\hline & RSECT & 212 \\
\hline & START & 214 \\
\hline & WXTRN & 229 \\
\hline & XATTR (MVS and CMS) & 230 \\
\hline \multirow[t]{2}{*}{Base Register} & DROP & 172 \\
\hline & USING & 218 \\
\hline \multirow[t]{5}{*}{Data Definition} & CCW & 115 \\
\hline & cCWo & 115 \\
\hline & CCW1 & 116 \\
\hline & DC & 126 \\
\hline & DS & 174 \\
\hline Symbol Definition & EQU & 184 \\
\hline Associated Data & ADATA & 107 \\
\hline \multirow[t]{2}{*}{Assembler Options} & *PROCESS & 102 \\
\hline & ACONTROL & 103 \\
\hline
\end{tabular}

\section*{64-bit Addressing Mode}

Some instructions have an operand or operands that pertain to 64-bit addressing mode (for example, 64 for AMODE). This operand is accepted and processed by the assembler. However, other operating system components and utility programs may not be able to accept and process information related to this operand.

\section*{*PROCESS Statement}

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

To ensure that certain assembler options cannot be changed for a given source file, put the OVERRIDE keyword as the first and only keyword on the process statement, followed by a list of options. This means that default and invocation options cannot override the specified options.

You can specify up to 10 process statements in each source program. Except for the ICTL instruction, process statements must be the first statements in your source program. If you include process statements anywhere else in your source program the assembler treats them as comments.

A process statement has a special coding format, unlike any other assembler instruction, although it is affected by the column settings of the ICTL instruction. You must code the characters *PROCESS starting in the begin column of the source statement, followed by one or more spaces. You can code as many assembler options that can fit in the remaining columns up to, and including the end column of the source statement.

You cannot continue a process statement on to the next statement.

assembler_option
is any assembler option.
A number of options are not accepted from a process statement. If the option is specified on a process override statement and differs from the option in effect at the time of processing the statement, the assembler issues a warning message. These options are:
\begin{tabular}{lll} 
ADATA & LANGUAGE & SYSPARM \\
ASA & LINECOUNT & TERM \\
DECK & LIST & TRANSLATE \\
EXIT & OBJECT & XOBJECT \\
GOFF & SIZE &
\end{tabular}

When the assembler detects an error in a process statement, it produces an error message in the High Level Assembler Option Summary section of the assembler listing. If the installation default option PESTOP is set then the assembler stops after it finishes processing any remaining process statements.

The assembler lists the options from process statements in the High Level Assembler Option Summary section of the assembler listing. The process statements are also shown as comment lines in the Source and Object section of the assembler listing.

\section*{ACONTROL Instruction}

The ACONTROL instruction can change these HLASM options within a program:
- AFPR
- COMPAT
- FLAG (except the RECORD/NORECORD and the PUSH/NOPUSH suboptions)
- LIBMAC
- RA2
- TYPECHECK

Note: The AFPR option is not available as an assembler option at invocation of the assembler.

The selections which can be specified are documented here for completeness.

sequence_symbol
is a sequence symbol.
selection
is one or more selections from the group of selections described below.
Because ACONTROL is making changes to existing values, there are no default values for the ACONTROL instruction.


\section*{AFPR}
instructs the assembler that the additional floating point registers 1, 3, 5 and 7 through 15 may be specified in the program.

Note: The assembler starts with AFPR enabled.

\section*{NOAFPR}
instructs the assembler that no additional floating point registers, that is, only floating point registers \(0,2,4\) and 6 may be specified in the program.

keyword:


\section*{COMPAT(CASE), abbreviation CPAT(CASE)}
instructs the assembler to maintain uppercase alphabetic character set compatibility with earlier assemblers.

\section*{COMPAT(NOCASE), abbreviation CPAT(NOCASE)}
instructs the assembler to allow mixed case alphabetic character set.

\section*{COMPAT(LITTYPE), abbreviation CPAT(LIT)}
instructs the assembler to return ' U ' as the type attribute for all literals.

\section*{COMPAT(NOLITTYPE), abbreviation CPAT(NOLIT)}
instructs the assembler to return the correct type attribute for literals.

\section*{COMPAT(MACROCASE), abbreviation CPAT(MC)}
instructs the assembler to convert internally lowercase alphabetic characters in unquoted macro operands to uppercase alphabetic characters prior to macro expansion. (The source statement is unchanged).

\section*{COMPAT(NOMACROCASE), abbreviation CPAT(NOMC)}
instructs the assembler not to convert lowercase alphabetic characters (a through \(z\) ) in unquoted macro operands.

\section*{COMPAT(SYSLIST), abbreviation CPAT(SYSL)}
instructs the assembler to treat sublists in SETC symbols as compatible with earlier assemblers.

\section*{COMPAT(NOSYSLIST), abbreviation CPAT(NOSYSL)}
instructs the assembler not to treat sublists in SETC symbols as character strings, when passed to a macro definition in an operand of a macro instruction.

\section*{NOCOMPAT, abbreviation NOCPAT}
instructs the assembler to allow lowercase alphabetic characters in all language elements, to treat sublists in SETC symbols as sublists when passed to a macro definition in the operand of a macro instruction, and to return the correct type attribute for literals.

integer
specifies that error diagnostic messages with this or a higher severity code are printed in the source and object section of the assembly listing.

\section*{FLAG(ALIGN), abbreviation FLAG(AL)}
instructs the assembler to issue diagnostic message ASMA033I, ASMA212W, or ASMA213W when an inconsistency is detected between the operation code and the alignment of addresses in machine instructions.

\section*{FLAG(NOALIGN), abbreviation FLAG(NOAL)}
instructs the assembler not to issue diagnostic message ASMA033I ASMA212W, or ASMA213W when an inconsistency is detected between the operation code and the alignment of addresses in machine instructions.

\section*{FLAG(CONT)}
specifies that the assembler is to issue diagnostic messages ASMA430W through ASMA433W when an inconsistent continuation is encountered in a statement.

\section*{FLAG(NOCONT)}
specifies that the assembler is not to issue diagnostic messages ASMA430W through ASMA433W when an inconsistent continuation is encountered in a statement.

\section*{FLAG(EXLITW)}
instructs the assembler to issue diagnostic warning ASMA016W when a literal is specified as the object of an EX instruction.

\section*{FLAG(NOEXLITW)}
instructs the assembler to suppress diagnostic warning message ASMA016W when a literal is specified as the object of an EX instruction.

\section*{FLAG(IMPLEN)}
instructs the assembler to issue diagnostic message ASMA169I when an explicit length subfield is omitted from an SS-format machine instruction.

\section*{FLAG(NOIMPLEN)}
instructs the assembler not to issue diagnostic message ASMA169I when an explicit length subfield is omitted from an SS-format machine instruction.

\section*{FLAG(PAGEO)}
instructs the assembler to issue diagnostic message ASMA309W when an operand is resolved to a baseless address and a base and displacement is expected.

\section*{FLAG(NOPAGEO)}
instructs the assembler not to issue diagnostic message ASMA309W when an operand is resolved to a baseless address and a base and displacement is expected.

\section*{FLAG(SUBSTR), abbreviation FLAG(SUB)}
instructs the assembler to issue warning diagnostic message ASMA094I when the second subscript value of the substring notation indexes past the end of the character expression.

\section*{FLAG(NOSUBSTR), abbreviation FLAG(NOSUB)}
instructs the assembler not to issue warning diagnostic message ASMA094I when the second subscript value of the substring notation indexes past the end of the character expression.

\section*{FLAG(USINGO), abbreviation FLAG(USO)}
instructs the assembler to issue diagnostic warning message ASMA306W for a USING that is coincident with or overlaps the implied USING 0,0 , when the USING(WARN) suboption includes the condition numbers 1 and/or 4.

\section*{FLAG(NOUSINGO), abbreviation FLAG(NOUSO)}
instructs the assembler to suppress diagnostic warning message ASMA306W


\section*{LIBMAC, abbreviation LMAC}
specifies that, for each macro, macro definition statements read from a macro library are to be imbedded in the input source program immediately preceding the first invocation of that macro.

\section*{NOLIBMAC, abbreviation NOLMAC}
specifies that macro definition statements read from a macro library are not to be included in the input source program.


RA2
instructs the assembler to suppress error diagnostic message ASMA066W when 2-byte relocatable address constants are defined in the source

\section*{NORA2}
instructs the assembler to issue error diagnostic message ASMA066W when 2-byte relocatable address constants are defined in the source


Note:
1 Choose at least one option.

\section*{TYPECHECK(MAGNITUDE)}
specifies that the assembler performs magnitude validation of signed immediate-data fields of machine instruction operands.

\section*{TYPECHECK(NOMAGNITUDE)}
specifies that the assembler not perform magnitude validation of signed immediate-data fields of machine instruction operands.

\section*{TYPECHECK(REGISTER)}
specifies that the assembler performs type checking of register fields of machine instruction operands.

\section*{TYPECHECK(NOREGISTER)}
specifies that the assembler not perform type checking of register fields of machine instruction operands.

\section*{NOTYPECHECK}
specifies that the assembler not perform any type checking of machine instruction operands.

For further details of the TYPECHECK option, refer to the High Level Assembler Programmer's Guide.

\section*{ADATA Instruction}

The ADATA instruction writes records to the associated data file.


\section*{sequence_symbol}
is a sequence symbol.

\section*{value1-value4}
up to four values may be specified, separated by commas. If a value is omitted, the field written to the associated data file contains binary zeros. You must code a comma in the operand for each omitted value. If specified, value1 through value 4 must be a decimal self-defining term with a value in the range \(-2^{31}\) to \(+2^{31}-1\).

\section*{AINSERT Instruction}
character_string
is a character string up to 255 bytes long, enclosed in single quotes. If omitted, the length of the user data field in the associated data file is set to zero.

\section*{Notes:}
1. All operands may be omitted to produce a record containing binary zeros in all fields except the user data field.
2. The record written to the associated data file is described under "User-Supplied Information Record X'0070'," in Appendix D, "Associated Data File Output" of the HLASM Programmer's Guide.
3. If you do not specify the ADATA assembler option, or the GOFF(ADATA) or the XOBJECT(ADATA) assembler option (MVS or CMS), the assembler only checks the syntax of an ADATA instruction, and prints it in the assembler listing.
4. The assembler writes associated data records to the SYSADATA (MVS or CMS), or the SYSADAT (VSE) file if the ADATA assembler option has been specified.

\section*{AINSERT Instruction}

The AINSERT instruction inserts statements into the input stream. These statements are queued in an internal buffer until the macro generator has completed expanding the current outermost macro instruction. At that point the internal buffer queue provides the next statement or statements. An operand controls the sequence of the statements within the internal buffer queue.
Note: While inserted statements may be placed at either end of the buffer queue, the statements are removed only from the front of the buffer queue.

sequence_symbol
is a sequence symbol.

\section*{statement}
is the statement stored in the internal buffer. It may be any characters enclosed in single quotation marks.

The rules that apply to this character string are:
- Variable symbols are allowed.
- The string may be up to 80 characters in length. If the string is longer than 80 characters, only the first 80 characters are used. The rest of the string is ignored.

\section*{BACK}

The statement is placed at the back of the internal buffer.

\section*{ALIAS Instruction}

\section*{FRONT}

The statement is placed at the front of the internal buffer.

\section*{Notes:}
1. The ICTL instruction does not affect the format of the stored statements. The assembler processes these statements according to the standard begin, end and continue columns.
2. The assembler does not check the stored statements, even when the ISEQ instruction is active.

\section*{Example:}


In this example the variable \&FIRST receives the operand of the AINSERT statement created at .B. \&SECOND receives the operand of the AINSERT statement created at .D. The operand of the AINSERT statements at .A and .C are in the internal buffer in the sequence .A followed by .C and are the next statements processed when the macro generator has finished processing.

Figure 65 on page 263 shows code using AINSERT in statements 16, 22, and 23.

\section*{ALIAS Instruction}

The ALIAS instruction specifies alternate names for the external symbols that identify control sections, entry points, and external references. The instruction has nothing to do with the link-time aliases in libraries.
--symbol—ALIAS—alias_string
symbol
is an external symbol that is represented by one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

\section*{AMODE Instruction}
alias_string
is the alternate name for the external symbol, represented by one of the following:
- A character constant in the form C'aaaaaaaa', where aaaaaaaa is a string of characters each of which has a hexadecimal value of \(X^{\prime} 42^{\prime}\) to \(X^{\prime} F^{\prime}\) inclusive
- A hexadecimal constant in the form \(\mathrm{X}^{\prime} x x x x x x x x^{\prime}\), where \(x x x x x x y x\) is a string of hexadecimal digits, each pair of which is in the range \(\mathrm{X}^{\prime} 42^{\prime}\) to \(X^{\prime}\) FE' \(^{\prime}\) inclusive

The ordinary symbol denoted by symbol must also appear in one of the following in this assembly:
- The name entry field of a START, CSECT, RSECT, COM, or DXD instruction
- The name entry field of a DSECT instruction and the nominal value of a Q-type offset constant
- The operand of an ENTRY, EXTRN or WXTRN instruction
- The nominal value of a V-type address constant

The assembler uses the string denoted by alias_string to replace the external symbol denoted by symbol in the external symbol dictionary records in the object module. If the string is shorter than 8 characters, or 16 hexadecimal digits, it is padded on the right with EBCDIC spaces ( \(\mathrm{X}^{\prime} 40^{\prime}\) ). If the string is longer than 8 characters, it is truncated. Some programs that process object modules do not support external symbols longer than 8 characters.

CMS, MVS If the extended object format is being generated (GOFF assembler option), the alias_string can be up to 256 characters, or 512 hexadecimal digits. CMS,MVS

The following examples are of the ALIAS instruction, and show both formats of the alternate name denoted by alias_string.
EXTSYM1 ALIAS C'lower1'
EXTSYM2 ALIAS X'9396A68599F2'

The alias_string must not match any external symbol, regardless of case.
Aliased names are not checked against other internal or external symbols or ALIASes for possible duplicates or conflicts.

\section*{AMODE Instruction}

The AMODE instruction specifies the addressing mode associated with control sections in the object deck.

name is the name field that associates the residence mode with a control section. If there is a symbol in the name field, it must also appear in the name field of a START, CSECT, RSECT, or COM instruction in this assembly. If the name field is space-filled, there must be an unnamed control section in this assembly. If the name field contains a sequence symbol (see "Symbols" on page 29 for details), it is treated as a blank name field.

CMS, MVS If the extended object format is being generated (GOFF assembler option), name is a relocatable symbol that names an entry point specified on an ENTRY instruction. CMS, MVS

24 Specifies that 24-bit addressing mode is to be associated with a control section, or entry point.

31 Specifies that 31-bit addressing mode is to be associated with a control section, or entry point.

64 Specifies that 64-bit addressing mode is to be associated with a control section, or entry point (see "64-bit Addressing Mode" on page 101).

ANY The same as ANY31.
ANY31 The control section or entry point is not sensitive to whether it is entered in AMODE 24 or AMODE 31.

ANY64 The control section or entry point is not sensitive to whether it is entered in AMODE 24, AMODE 31, or AMODE 64.

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

If symbol denotes an ordinary symbol, the ordinary symbol associates the addressing mode with a control section. The ordinary symbol must also appear in the name field of a START, CSECT, RSECT, or COM instruction in this assembly.

If symbol is not specified, or if name is a sequence symbol, there must be an unnamed control section in this assembly.

\section*{Notes:}
1. AMODE can be specified anywhere in the assembly. It does not initiate an unnamed control section.
2. AMODE is permitted on external labels (EXTRNs) and Entry labels for both GOFF and OBJ formats and Parts for GOFF formats.
3. An assembly can have multiple AMODE instructions; however, two AMODE instructions cannot have the same name field.
4. The valid and invalid combinations of AMODE and RMODE are shown in the following table. Note that combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64-bit Addressing Mode" on page 101

\section*{CATTR Instruction}

Figure 25. AMODE/RMODE Combinations
\begin{tabular}{lccc}
\hline & RMODE 24 & RMODE 31 & RMODE 64 \\
\hline AMODE 24 & OK & invalid & invalid \\
\hline AMODE 31 & OK & OK & invalid \\
\hline AMODE ANYIANY31 & OK & OK & invalid \\
\hline AMODE 64IANY64 & OK & OK & OK \\
\hline
\end{tabular}
5. AMODE or RMODE cannot be specified for an unnamed common control section.
6. The defaults used when zero or one MODE is specified are shown in the following table. Note that combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64-bit Addressing Mode" on page 101

Figure 26. AMODE/RMODE Defaults
\begin{tabular}{ll}
\hline Specified & Default \\
\hline Neither & AMODE 24, RMODE 24 \\
\hline AMODE 24 & RMODE 24 \\
\hline AMODE 31 & RMODE 24 \\
\hline AMODE ANYIANY31 & RMODE 24 \\
\hline RMODE 24 & AMODE 24 \\
\hline RMODE 31 (was ANY) & AMODE 31 \\
\hline AMODE 64 & RMODE 31 \\
\hline AMODE ANY64 & RMODE 31 \\
\hline RMODE 64 & AMODE 64 \\
\hline
\end{tabular}

\section*{CATTR Instruction (MVS and CMS)}

The CATTR instruction establishes a program object external class name, and assigns binder attributes for the class. This instruction is valid only when you specify the GOFF or XOBJECT assembler option.

class_name
is a valid program object external class name. The class name must follow the rules for naming external symbols, except that:
- Class names are restricted to a maximum of 16 characters
- Class names with an underscore (_) in the second character are reserved for IBM use; for example \(B_{-} T E X T\). If you use a class name of this format, it might conflict with an IBM-defined binder class.
attribute
is one or more binder attributes that are assigned to the text in this class:

\section*{ALIGN(n)}

Aligns the text on a \(2^{n}\) boundary. \(n\) is an integer in the range from 0 to 12 , inclusive. If not specified, then the SECTALGN value ( 8 is the default) is used.

Note: Execution time support of the desired alignment depends on its being respected by other operating system components such as linkers and loaders.

\section*{EXECUTABLE}

The text can be branched to or executed-it is instructions, not data.

\section*{DEFLOAD}

The text is not loaded when the program object is brought into storage, but will probably be requested, and therefore should be partially loaded, for fast access.

\section*{MERGE}

The text has the merge binding property. For example, pseudo-registers or external dummy sections have the "merge" binding property.
Merge classes can contain initial text. If they do contain initial text, they must have a class name beginning with \(C_{\text {_ }}\).

\section*{MOVABLE}

The text can be moved, and is reenterable (that is, it is free of location-dependent data such as address constants, and executes normally if moved to a properly aligned boundary).

\section*{NOLOAD}

The text for this class is not loaded when the program object is brought into storage. An external dummy section is an example of a class which is defined in the source program but not loaded.

\section*{NOTEXECUTABLE}

The text cannot be branched to or executed (that is, it is data, not instructions).

\section*{NOTREUS}

The text is marked not reusable.

\section*{PART(part-name)}

Identifies or continues the part with the name part-name. The part-name can be up to 63 characters in length. An invalid part-name is ignored and diagnostic message 'ASMA062E Illegal operand format xxxxxx' is issued.
Binding attributes assigned to the class are also assigned to the part. Both the class and the part are assigned to Name Space 3 and are assigned the merge attribute.
Text within a part cannot contain an entry point. If an entry point is found within the part it is ignored and diagnostic message 'ASMA048E Entry error - xxxxxxx' is issued.

The following rules apply to the validation of the PART attribute on the CATTR instruction:
- If the PART attribute has not been specified on the first CATTR statement for the class, but is specified on subsequent CATTR
statements for the class, the attribute is ignored and diagnostic message ASMA191W is issued.
- If the PART attribute has been specified on the first CATTR statement for the class, but is not specified on subsequent CATTR statements for the class, the diagnostic message ASMA155S is issued.
- Multiple parts may be defined within a class.

\section*{PRIORITY(nnnnn)}

The binding priority to be attached to this part. The value must be specified as an unsigned decimal number and must lie between 0 and \(2^{31}-1\). An invalid priority is ignored and diagnostic message 'ASMA062E Illegal operand format \(x x x x x x^{\prime}\) is issued.

The PRIORITY attribute may be specified on the first CATTR instruction for the part. If the PRIORITY attribute is specified on second and subsequent CATTR instructions for the part it is ignored and the diagnostic message ASMA191W is issued.

The PRIORITY attribute is ignored if there is no PART attribute on the CATTR instruction and the diagnostic message 'ASMA062E Illegal operand format \(x_{x x x x x ' ~ i s ~ i s s u e d . ~}^{\text {is }}\)

\section*{READONLY}

The text is storage-protected.

\section*{REFR}

The text is marked refreshable.

\section*{RENT}

The text is marked reenterable.

\section*{REUS}

The text is marked reusable.

\section*{RMODE(24)}

The text has a residence mode of 24 .

\section*{RMODE(31)}

The text has a residence mode of 31 .

\section*{RMODE(ANY)}

The text may be placed in any addressable storage.
Refer to the z/OS MVS Program Management: User's Guide and Reference, SA22-7643 for details about the binder attributes.

Default Attributes: When you don't specify attributes on the CATTR instruction the defaults are:

\section*{ALIGN(3),EXECUTABLE,NOTREUS,RMODE(24)}

The LOAD attribute is the default if neither DEFLOAD nor NOLOAD are specified.
Where to Use the CATTR Instruction: Use the CATTR instruction anywhere in a source module after any ICTL or *PROCESS statements. The CATTR instruction must be preceded by a START, CSECT, or RSECT statement, otherwise the assembler issues diagnostic message ASMA190E.

If several CATTR instructions within a source module have the same class name, the first occurrence establishes the class and its attributes, and the rest indicate the continuation of the text for the class. If you specify attributes on subsequent CATTR instructions having the same class name as a previous CATTR instruction, the assembler ignores the attributes and issues diagnostic message ASMA191W.

If you specify conflicting attributes on the same instruction, the assembler uses the last one specified. In the following example, the assembler uses RMODE(ANY):
MYCLASS CATTR RMODE(24),RMODE(ANY)
Syntax Checking Only: If you code a CATTR instruction but don't specify the GOFF or XOBJECT option, the assembler checks the syntax of the instruction statement and does not process the attributes.

\section*{CCW and CCWO Instructions}

The CCW and CCWO instructions define and generate an 8-byte, format-0 channel command word for input/output operations. A format-0 channel command word allows a 24 -bit data address. The CCW and CCWO instructions have identical functions. If a control section has not been established, CCW and CCW0 will initiate an unnamed (private) control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol
command_code
is an absolute expression that specifies the command code. This expression's value is right-justified in byte 0 of the generated channel command word.
data_address
is a relocatable or absolute expression that specifies the address of the data to operate upon. This value is treated as a 3-byte, A-type address constant. The value of this expression is right-justified in bytes 1 to 3 of the generated channel command word.
flags
is an absolute expression that specifies the flags for bits 32 to 37 , and zeros for bits 38 and 39 , of the generated channel command word. The value of this expression is right-justified in byte 4 of the generated channel command word. Byte 5 is set to zero by the assembler.
data_count
is an absolute expression that specifies the byte count or length of data. The value of this expression is right-justified in bytes 6 and 7 of the generated channel command word.

The generated channel command word is aligned at a doubleword boundary. Any skipped bytes are set to zero.

The internal machine format of a channel command word is shown in Figure 27.

Figure 27. Channel Command Word, Format 0
\begin{tabular}{lll}
\hline Byte & Bits & Usage \\
\hline 0 & \(0-7\) & Command code \\
\hline \(1-3\) & \(8-31\) & Address of data to operate upon \\
\hline 4 & \(32-37\) & Flags \\
\hline & \(38-39\) & Must be specified as zeros \\
\hline 5 & \(40-47\) & Set to zeros by assembler \\
\hline \(6-7\) & \(48-63\) & Byte count or length of data \\
\hline
\end{tabular}

If symbol is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the generated channel command word. The length attribute value of the symbol is 8 .

The following are examples of CCW and CCW0 statements:
```

WRITE1 CCW
1,DATADR,X'48',X'50'
WRITE2 CCW0 1,DATADR,X'48',X'50'

```

The object code generated (in hexadecimal) for either of the above examples is: 01 xxxxxx 48000050
where \(x x x x x x\) contains the address of DATADR, and DATADR must reside below 16 megabytes.

MVS

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

Specifying RMODE: Use RMODE 24 with CCW or CCW0 to ensure that valid data addresses are generated. If you use RMODE ANY with CCW or CCW0, an invalid data address in the channel command word can result at execution time.

\section*{CCW1 Instruction}

The CCW1 instruction defines and generates an 8-byte format-1 channel command word for input/output operations. A format-1 channel command word allows 31-bit data addresses. A format-0 channel command word generated by a CCW or CCW0 instruction allows only a 24 -bit data address. If a control section has not been established, CCW1 will initiate an unnamed (private) control section.
\[
L_{\text {symbol }} \text { CCW1—command_code,data_address,flags,data_count } \longrightarrow
\]

\section*{symbol}
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol
command_code
is an absolute expression that specifies the command code. This expression's value is right-justified in byte 0 of the generated channel command word.

\section*{data_address}
is a relocatable or absolute expression that specifies the address of the data to operate upon. This value is treated as a 4-byte, A-type address constant. The value of this expression is right-justified in bytes 4 to 7 of the generated channel command word.

\section*{flags}
is an absolute expression that specifies the flags for bits 8 to 15 of the generated channel command word. The value of this expression is right-justified in byte 1 of the generated channel command word.
data_count
is an absolute expression that specifies the byte count or length of data. The value of this expression is right-justified in bytes 2 and 3 of the generated channel command word.

The generated channel command word is aligned at a doubleword boundary. Any skipped bytes are set to zero.

The internal machine format of a channel command word is shown in Figure 28.

Figure 28. Channel Command Word, Format 1
\begin{tabular}{lll}
\hline Byte & Bits & Usage \\
\hline 0 & \(0-7\) & Command code \\
\hline 1 & \(8-15\) & Flags \\
\hline \(2-3\) & \(16-31\) & Count \\
\hline 4 & 32 & Must be zero \\
\hline \(4-7\) & \(33-63\) & Data address \\
\hline
\end{tabular}

The expression for the data address should be such that the address is within the range 0 to \(2^{31}-1\), inclusive, after possible relocation. This is the case if the expression refers to a location within one of the control sections that are link-edited together. An expression such as *-1000000000 yields an acceptable value only when the value of the location counter \(\left(^{*}\right)\) is 1000000000 or higher at assembly time.

\section*{CEJECT Instruction}

If symbol is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the generated channel command word. The length attribute value of the symbol is 8 .

The following is an example of a CCW1 statement:
A CCW1 X'0C',BUF1, X'00', L'BUF1
The object code generated (in hexadecimal) for the above examples is:
\(0 C 00\) yyyy xxxxxxxx
where yyyy is the length of BUF1 and \(x x x x x x x x\) is the address of BUF1. BUF1 can reside anywhere in virtual storage.

\section*{CEJECT Instruction}

The CEJECT instruction conditionally stops the printing of the assembler listing on the current page, and continues the printing on the next page.

sequence_symbol
is a sequence symbol.
number_of_lines
is an absolute value that specifies the minimum number of lines that must be remaining on the current page to prevent a page eject. If the number of lines remaining on the current page is less than the value specified by number_of_lines, the next line of the assembler listing is printed at the top of a new page.

You may use any absolute expression to specify number_of_lines.
If number of lines is omitted, the CEJECT instruction behaves as an EJECT instruction.

If zero, a page is ejected unless the current line is at the top of a page.
If the line before the CEJECT statement appears at the bottom of a page, the CEJECT statement has no effect. A CEJECT instruction without an operand immediately following another CEJECT instruction or an EJECT instruction is ignored.

\section*{Notes:}
1. The CEJECT statement itself is not printed in the listing unless a variable symbol is specified as a point of substitution in the statement, in which case the statement is printed before substitution occurs.
2. The PRINT DATA and PRINT NODATA instructions can alter the effect of the CEJECT instruction, depending on the number of assembler listing lines that are required to print the generated object code for each instruction.

\section*{CNOP Instruction}

The CNOP instruction aligns any instruction or other data on a specific halfword boundary. This ensures an unbroken flow of executable instructions, since the CNOP instruction generates no-operation instructions to fill the bytes skipped to achieve specified alignment. If a control section has not been established, CNOP will initiate an unnamed (private) control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The name is assigned to the next halfword aligned location. There may be a single byte before that location, but this is skipped for alignment. There may be zero or more \(\mathrm{NOP}(\mathrm{R})\) s generated at or after that location.

\section*{byte}
is an absolute expression that specifies at which even-numbered byte in a fullword, doubleword, or quadword the location counter is set. The value of the expression must be 0 to boundary- 2 .

\section*{boundary}
is an absolute expression that specifies the byte specified by boundary is in a fullword, doubleword, or quadword. A value of 4 indicates the byte is in a fullword, a value of 8 indicates the byte is in a doubleword, and a value of 16 indicates the byte is in a quadword.

Figure 29 shows valid pairs of byte and word.

Figure 29 (Page 1 of 2). Valid CNOP Values
\begin{tabular}{ll}
\hline Values & Specify \\
\hline 0,4 & Beginning of a word \\
\hline 2,4 & Middle of a word \\
\hline 0,8 & Beginning of a doubleword \\
\hline 2,8 & Second halfword of a doubleword \\
\hline 4,8 & Middle (third halfword) of a doubleword \\
\hline 6,8 & Fourth halfword of a doubleword \\
\hline 0,16 & Beginning of a quadword \\
\hline 2,16 & Second halfword of a quadword \\
\hline 4,16 & Third halfword of a quadword \\
\hline 8,16 & Fourth halfword of a quadword \\
\hline
\end{tabular}

Figure 29 (Page 2 of 2). Valid CNOP Values
\begin{tabular}{ll}
\hline Values & Specify \\
\hline 10,16 & Sixth halfword of a quadword \\
\hline 12,16 & Seventh halfword of a quadword \\
\hline 14,16 & Eighth halfword of a quadword \\
\hline
\end{tabular}

Figure 30 shows the position in a doubleword that each of these pairs specifies. Note that both 0,4 and 2,4 specify two locations in a doubleword.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{16}{|c|}{Quadword} \\
\hline \multicolumn{8}{|c|}{Doubleword} & \multicolumn{8}{|c|}{Doubleword} \\
\hline \multicolumn{4}{|c|}{Fullword} & \multicolumn{4}{|c|}{Fullword} & \multicolumn{4}{|c|}{Fullword} & \multicolumn{4}{|c|}{Fullword} \\
\hline \multicolumn{2}{|c|}{Hal fword} & \multicolumn{2}{|c|}{Hal fword} & \multicolumn{2}{|r|}{Hal fword} & \multicolumn{2}{|c|}{Hal fword} & \multicolumn{2}{|c|}{Hal fword} & \multicolumn{2}{|c|}{Hal fword} & \multicolumn{2}{|r|}{Hal fword} & \multicolumn{2}{|c|}{Hal fword} \\
\hline Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte & Byte \\
\hline 0,4 & \multicolumn{3}{|c|}{2,4} & \multicolumn{2}{|l|}{0,4} & \multicolumn{2}{|l|}{2,4} & \multicolumn{2}{|l|}{0,4} & \multicolumn{2}{|l|}{2,4} & \multicolumn{2}{|l|}{0,4} & \multicolumn{2}{|l|}{2,4} \\
\hline 0,8 & \multicolumn{3}{|c|}{2,8} & \multicolumn{2}{|l|}{4,8} & \multicolumn{2}{|l|}{6,8} & \multicolumn{2}{|l|}{0,8} & \multicolumn{2}{|l|}{2,8} & \multicolumn{2}{|l|}{4,8} & \multicolumn{2}{|l|}{6,8} \\
\hline 0,16 & & \multicolumn{2}{|l|}{2,16} & \multicolumn{2}{|l|}{4,16} & \multicolumn{2}{|l|}{6,16} & \multicolumn{2}{|l|}{8,16} & \multicolumn{2}{|l|}{10,16} & \multicolumn{2}{|l|}{12,16} & \multicolumn{2}{|l|}{14,16} \\
\hline
\end{tabular}

Figure 30. CNOP Alignment
Use the CNOP instruction, for example, when you code the linkage to a subroutine, and you want to pass parameters to the subroutine in fields immediately following the branch and link instructions. These parameters-for example, channel command words-can require alignment on a specific boundary. The subroutine can then address the parameters you pass through the register with the return address, as in the following example:
\begin{tabular}{lll} 
& CNOP & 6,8 \\
\multirow{2}{*}{ LINK } & BALR & 2,10 \\
& CCW & 1, DATADR \(, X^{\prime} 48^{\prime}, X^{\prime} 50^{\prime}\)
\end{tabular}

Assume that the location counter is aligned at a doubleword boundary. Then the CNOP instruction causes the following no-operations to be generated, thus aligning the BALR instruction at the last halfword in a doubleword as follows:
\begin{tabular}{lll} 
& BCR & 0,0 \\
& BC & \(0, X^{\prime} 700^{\prime}\) \\
BALR & 2,10 \\
LINK & CCW & \(1, D A T A D R, X^{\prime} 48^{\prime}, X^{\prime} 50^{\prime}\)
\end{tabular}

After the BALR instruction is generated, the location counter is at a doubleword boundary, thereby ensuring that the CCW instruction immediately follows the branch and link instruction.

The CNOP instruction forces the alignment of the location counter to a halfword, fullword, doubleword or quadword boundary. It does not affect the location counter if the counter is already correctly aligned. If the specified alignment requires the location counter to be incremented, no-operation instructions are generated to fill the skipped bytes. Any single byte skipped to achieve alignment to the first no-operation instruction is filled with zeros, even if the preceding byte contains no machine language object code. A length attribute reference to the name of a CNOP instruction is always invalid. Message ASMA042E will be issued, and a default value of 1 is assigned.

\section*{COM Instruction}

The COM instruction identifies the beginning or continuation of a common control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The COM instruction can be used anywhere in a source module after the ICTL instruction.

If symbol denotes an ordinary symbol, the ordinary symbol identifies the common control section. If several COM instructions within a source module have the same symbol in the name field, the first occurrence initiates the common section and the rest indicate the continuation of the common section. The ordinary symbol denoted by symbol represents the address of the first byte in the common section, and has a length attribute value of 1 .

If symbol is not specified, or if name is a sequence symbol, the COM instruction initiates, or indicates the continuation of, the unnamed common section.

See "CSECT Instruction" on page 123 for a discussion on the interaction between COM and the GOFF assembler option.

The location counter for a common section is always set to an initial value of 0 . However, when an interrupted common control section is continued using the COM instruction, the location counter last specified in that control section is continued.

If a common section with the same name (or unnamed) is specified in two or more source modules, the amount of storage reserved for this common section is equal to that required by the longest common section specified.

The source statements that follow a COM instruction belong to the common section identified by that COM instruction.

\section*{Notes:}
1. The assembler language statements that appear in a common control section are not assembled into object code.
2. When establishing the addressability of a common section, the symbol in the name field of the COM instruction, or any symbol defined in the common section, can be specified in a USING instruction.
3. An AMODE cannot be assigned to a common section.

In the following example, addressability to the common area of storage is established relative to the named statement XYZ.
```

L 1,=A(XYZ)
USING XYZ,1
MVC
PDQ(16),=4C'ABCD'
.
COM
XYZ DS 16F
PDQ DS 16C

```

A common control section may include any assembler language instructions, but no object code is generated by the assembly of instructions or constants appearing in a common control section. Data can only be placed in a common control section through execution of the program.

If the common storage is assigned in the same manner by each independent assembly, reference to a location in common by any assembly results in the same location being referenced.

\section*{COPY Instruction}

Use the COPY instruction to obtain source statements from a source language library and include them in the program being assembled. You can thereby avoid writing the same, often-used sequence of code over and over.

sequence_symbol
is a sequence symbol.

\section*{member}
is an ordinary symbol that identifies a source language library member to be copied from either a system macro library or a user macro library. In open code it can also be a variable symbol that has been assigned a valid ordinary symbol.

The source statements that are copied into a source module:
- Are inserted immediately after the COPY instruction.
- Are inserted and processed according to the standard instruction statement coding format, even if an ICTL instruction has been specified.
- Must not contain either an ICTL or ISEQ instruction.
- Can contain other COPY statements. There are no restrictions on the number of levels of nested copy instructions. However, the COPY nesting must not be recursive. For example, assume that the source program contains the statement:

COPY A
and library member A contains the statement:
COPY B
In this case, the library member B must not contain a COPY A or COPY B statement.
- Can contain macro definitions. Note, however, that if a source macro definition is copied into a source module, both the MACRO and MEND statements that delimit the definition must be contained in the same level of copied code.

\section*{Notes:}
1. The COPY instruction can also be used to copy statements into source macro definitions.
2. The rules that govern the occurrence of assembler language statements in a source module also govern the statements copied into the source module.
3. Whenever the assembler processes a COPY statement, whether it is in open code or in a macro definition, the assembler attempts to read the source language library member specified in the COPY statement. This means that all source language library members specified by COPY statements in a source program, including those specified in macro definitions, must be available during the assembly. The HLASM Programmer's Guide describes how to specify the libraries when you run the assembler.
4. If an END instruction is encountered in a member during COPY processing, the assembly is ended. Any remaining statements in the COPY member are discarded.

\section*{CSECT Instruction}

The CSECT instruction initiates an executable control section or indicates the continuation of an executable control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The CSECT instruction can be used anywhere in a source module after any ICTL or *PROCESS statements. If it is used to initiate the first executable control section, it must not be preceded by any instruction that affects the location counter and thereby causes a control section to be initiated.

If symbol denotes an ordinary symbol, the ordinary symbol identifies the control section. If several CSECT instructions within a source module have the same
symbol in the name field, the first occurrence initiates the control section and the rest indicate the continuation of the control section. The ordinary symbol denoted by symbol represents the address of the first byte in the control section, and has a length attribute value of 1 .

If symbol is not specified, or if name is a sequence symbol, the CSECT instruction initiates, or indicates the continuation of the unnamed control section.

If the first control section is initiated by a START instruction, the CSECT instruction which continues the section must have the same name as the START instruction.

CMS, MVS When the GOFF option is not specified a control section is initiated or resumed by the CSECT, RSECT, and COM statements. Any machine language text created by statements that follow such control section declarations belongs to the control section, and is manipulated during program linking and binding as an indivisible unit.

When the GOFF option is specified, the behavior of CSECT, RSECT, and COM statements is somewhat different. By default, the assembler creates a definition of a text class named B_TEXT, to which subsequent machine language text belongs if no other classes are declared. If you specify other class names using the CATTR statement, machine language text following such CATTR statements belongs to that class.

The combination of a section name and a class name defines an element, which is the indivisible unit manipulated during linking and binding. All elements with the same section name are "owned" by that section, and binding actions (such as section replacement) act on all elements owned by a section.

When the GOFF option is specified, and if no CATTR statements are present, then all machine language text is placed in the default class B_TEXT, and the behavior of the elements in the bound module is essentially the same as the behavior of control sections when the OBJECT option is specified. However, if additional classes are declared, a section name can best be thought of as a "handle" by which elements within declared classes are owned. CMS, MVS

The beginning of a control section is aligned on a boundary determined by the SECTALGN option. However, when an interrupted control section is continued using the CSECT instruction, the location counter last specified in that control section is continued. Consider the coding in Figure 31:


Figure 31. How the Location Counter Works

The source statements following a CSECT instruction that either initiate or indicate the continuation of a control section are assembled into the object code of the control section identified by that CSECT instruction.

The end of a control section or portion of a control section is marked by:
- Any instruction that defines a new or continued control section
- The END instruction

The CSECT instruction may interact with any LOCTR instructions that are present. For more information about this interaction, see "LOCTR Instruction" on page 191

\section*{CXD Instruction}

The CXD instruction reserves a fullword area in storage. The linker or loader inserts into this area the total length of all external dummy sections specified in the source modules that are assembled and linked into one program. If a control section has not previously been established, CXD will initiate an unnamed (private) control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The linker or loader inserts into the fullword-aligned fullword area reserved by the CXD instruction the total length of storage required for all the external dummy sections specified in a program. If the GOFF assembler option is specified, CXD returns the length of the B_PRV class. If symbol denotes an ordinary symbol, the ordinary symbol represents the address of the fullword area. The ordinary symbol denoted by symbol has a length attribute value of 4 .

The following examples shows how external dummy sections may be used:
ROUTINE A
\begin{tabular}{lll} 
ALPHA & DXD & \(2 D L 8\) \\
BETA & DXD & \(4 F L 4\) \\
OMEGA & CXD & \\
& \(\cdot\) & \\
& DC & Q(ALPHA) \\
& DC & Q(BETA) \\
& \(\cdot\) & \\
& \(\cdot\) &
\end{tabular}

\section*{DC Instruction}

ROUTINE B
\begin{tabular}{lll} 
GAMMA & DXD & \(5 D\) \\
DELTA & DXD & \(10 F\) \\
ZETA & DXD & XL22 \\
& \(\cdot\) & \\
& \(\cdot\) & \\
& DC & (GAMMA) \\
& DC & Q(DELTA) \\
& \(\cdot\) & \\
& \(\cdot\) &
\end{tabular}

ROUTINE C
\begin{tabular}{lll} 
EPSILON & DXD & \(4 H\) \\
ZETA & DXD & 4 F \\
& \(\cdot\) & \\
& DC & Q(EPSILON,ZETA)
\end{tabular}

Each of the three routines is requesting an amount of work area. Routine \(A\) wants 2 doublewords and 4 fullwords; Routine B wants 5 doublewords, 10 fullwords, and 22 bytes; Routine \(C\) wants 4 halfwords and 4 fullwords. During program linking, identically named dummy sections are combined, retaining their strictest alignment and longest length. For example, Routines \(B\) and \(C\) both request storage named ZETA: the resulting allocation will be 22 bytes on a fullword boundary. When program linking is complete, the sum of these individual dummy external section lengths is placed in the location of the CXD instruction labeled OMEGA. Routine A can then allocate the amount of storage that is specified in the CXD location, and each dummy external section's offset within the allocated storage is found in the Q-type offset constant referencing its name. Q-type offset constants are described at "Offset Constant-Q" on page 159

\section*{DC Instruction}

You specify the DC instruction to define the data constants you need for program execution. The DC instruction causes the assembler to generate the binary representation of the data constant you specify into a particular location in the assembled source module; this is done at assembly time.

Note that the DC instruction's name - Define Constant - is somewhat misleading: DC simply creates initial data in an area of the program. The contents of that area may be modified during program execution, so the original data isn't truly "constant." If you want to declare values that are more likely to behave like constants, use literals "Literals" on page 40; the assembler attempts to detect and diagnose instructions that might change the contents of a field defined by a literal. If a control section has not been established previously, DC will initiate an unnamed (private) control section.

The DC instruction can generate the following types of constants:

Figure 32. Types of Data Constants
\(\left.\begin{array}{lllll}\hline \begin{array}{l}\text { Type of } \\ \text { Constant }\end{array} & \text { Function } & \text { Example } & \\ \hline \text { Address } & \begin{array}{l}\text { Defines address mainly } \\ \text { for the use of fixed-point } \\ \text { and other instructions }\end{array} & \text { ADCON } & \text { L } & \text { DC }\end{array} \begin{array}{l}\text { 5,ADCON } \\ \text { A(SOMWHERE) }\end{array}\right]\)

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If symbol denotes an ordinary symbol, the ordinary symbol represents the address of the first byte of the assembled constant. If several operands are specified, the first constant defined is addressable by the ordinary symbol. The other constants can be reached by relative addressing.

\section*{operands}

An operand of six subfields. The first five subfields describe the constant. The sixth subfield provides the nominal values for the constants.

A DC operand has this format:

duplication_factor
causes the nominal_value to be generated the number of times indicated by
this factor. See" "Subfield 1: Duplication Factor" on page 132
type
further determines the type of constant the nominal_value represents. See "Subfield 2: Type" on page 133
type_extension
determines some of the characteristics of the constant. See "Subfield 3: Type Extension" on page 134.

\section*{Rules for DC Operand}
1. The type subfield and the nominal value must always be specified unless the duplication factor is zero. If the duplication factor is zero, only the type must be specified.
2. The duplication factor, type extension, program type, and modifier subfields are optional.
3. When multiple operands are specified, they can be of different types.
4. When multiple nominal values are specified in the sixth subfield, they must be separated by commas and be of the same type. Multiple nominal values are not allowed for character or graphic constants.
5. The descriptive subfields, apart from the program type, apply to all the nominal values. The program type applies to only the symbol naming the DC instruction, if a symbol was present. Separate constants are generated for each separate operand and nominal value specified.
6. No spaces are allowed:
- Between subfields
- Between multiple operands

\section*{General Information About Constants}

Constants defined by the DC instruction are assembled into an object module at the location at which the instruction is specified. However, the type of constant being defined, and the presence or absence of a length modifier, determines whether the constant is to be aligned on a particular storage boundary or not (see "Alignment of Constants").

Symbolic Addresses of Constants: The value of the symbol that names the DC instruction is the address of the first byte (after alignment) of the first or only constant.

\section*{Length Attribute Value of Symbols Naming Constants}

The length attribute value assigned to the symbols in the name field of the constants is equal to:
- The implicit length (see "Implicit Length" in Figure 33) of the constant when no explicit length is specified in the operand of the constant, or
- The explicit length (see "Value of Length Attribute" in Figure 33) of the constant.

If more than one operand is present, the length attribute value of the symbol is the length in bytes of the first constant specified, according to its implicit or explicit length.

\section*{Alignment of Constants}

The assembler aligns constants on different boundaries according to the following:
- On boundaries implicit to the type of constant (see "Implicit Boundary Alignment" in Figure 34 on page 130 when no length is specified.
- On byte boundaries (see "Boundary Alignment" in Figure 34) when an explicit length is specified.

Bytes that are skipped to align a constant at the correct boundary are not considered part of the constant. They are filled with binary zeros.

\section*{Notes:}
1. The automatic alignment of constants and areas does not occur if the NOALIGN assembler option has been specified.
2. Alignment can be forced to any boundary by a preceding DS or DC instruction with a zero duplication factor. This occurs whether or not the ALIGN option is set.

Figure 33 (Page 1 of 2). Length Attribute Value of Symbol Naming Constants
\begin{tabular}{llll}
\hline Type of & Implicit & & \begin{tabular}{l} 
Value of \\
Length \\
constant
\end{tabular} \\
\hline Length & Examples & \begin{tabular}{l} 
Attribute \({ }^{1}\)
\end{tabular} \\
\hline as needed & DC \(\quad B^{\prime} 10010000^{\prime}\) & 1 \\
\hline
\end{tabular}

Figure 33 (Page 2 of 2). Length Attribute Value of Symbol Naming Constants
\begin{tabular}{|c|c|c|c|c|}
\hline Type of constant & Implicit Length & \multicolumn{2}{|l|}{Examples} & Value of Length Attribute \({ }^{1}\) \\
\hline \multirow[t]{2}{*}{C} & \multirow[t]{2}{*}{as needed} & DC & \(C^{\prime} A B C^{\prime}\) & 3 \\
\hline & & DC & CL8'WOW' & 8 \\
\hline \multirow[t]{2}{*}{CU} & \multirow[t]{2}{*}{as needed} & DC & CU'ABC' & 6 \\
\hline & & DC & CUL4'XX' & 4 \\
\hline \multirow[t]{2}{*}{G} & \multirow[t]{2}{*}{as needed} & DC & \(\mathrm{G}^{\prime}<\mathrm{CaD}^{\text {d }}\) > \({ }^{\prime}\) & 4 \\
\hline & & DC & GL8'<DaDb>' & 8 \\
\hline \multirow[t]{2}{*}{X} & \multirow[t]{2}{*}{as needed} & DC & X'COFFEE' & 3 \\
\hline & & DC & XL2'FFEE' & 2 \\
\hline H & 2 & DC & H'32' & 2 \\
\hline F & 4 & DC & FL3'32' & 3 \\
\hline FD & 8 & DC & FD'32' & 8 \\
\hline \multirow[t]{2}{*}{P} & \multirow[t]{2}{*}{as needed} & DC & \(P^{\prime} 123 '\) & 2 \\
\hline & & DC & PL4'123' & 4 \\
\hline \multirow[t]{2}{*}{Z} & \multirow[t]{2}{*}{as needed} & DC & Z'123' & 3 \\
\hline & & DC & ZL10'123' & 10 \\
\hline E & 4 & DC & E'565.40' & 4 \\
\hline D & 8 & DC & DL6'565.40' & 6 \\
\hline L & 16 & DC & LL12'565.40' & 12 \\
\hline LQ & 16 & DC & LQ'565.40' & 16 \\
\hline Y & 2 & DC & Y (HERE) & 2 \\
\hline A & 4 & DC & AL1 (THERE) & 1 \\
\hline AD & 8 & DC & AD (WHERE) & 8 \\
\hline S & 2 & DC & S(THERE) & 2 \\
\hline V & 4 & DC & VL3 (OTHER) & 3 \\
\hline VD & 8 & DC & VD(BIGOTHER) & 8 \\
\hline J & 4 & DC & J(CLASS) & 4 \\
\hline JD & 4 & DC & JD(LARGECLASS) & 8 \\
\hline Q & 8 & DC & QL1(LITTLE) & 1 \\
\hline QD & 4 & DC & QD(BIGLITTLE) & 8 \\
\hline R & 8 & DC & R(APSECT) & 4 \\
\hline RD & & DC & RD (BPSECT) & 8 \\
\hline
\end{tabular}

\section*{Note:}
1. Depends on whether or not an explicit length is specified in the constant.

\section*{Padding and Truncation of Values}

The nominal values specified for constants are assembled into storage. The amount of space available for the nominal value of a constant is determined:
- By the explicit length specified in the length modifier, or
- If no explicit length is specified, by the implicit length according to the type of constant defined (see Appendix B, "Summary of Constants" on page 407.

Figure 34. Alignment of Constants
\begin{tabular}{|c|c|c|c|c|}
\hline Type of constant & Implicit Boundary Alignment & \multicolumn{2}{|l|}{Examples} & Boundary Alignment \({ }^{1}\) \\
\hline B & byte & DC & B'1011' & byte \\
\hline C & byte & DC & C'Character string' & byte \\
\hline CU & byte & DC & CU'Character string' & byte \\
\hline G & byte & DC & G'<.D.B.C.S .S.T.R.I.N.G> & byte \\
\hline X & byte & DC & X'20202021202020' & byte \\
\hline \multirow[t]{2}{*}{H} & \multirow[t]{2}{*}{halfword} & DC & \(\mathrm{H}^{\prime} 25^{\prime}\) & halfword \\
\hline & & DC & HL3'25' & byte \\
\hline \multirow[t]{2}{*}{F} & \multirow[t]{2}{*}{fullword} & DC & \(F^{\prime} 225^{\prime}\) & fullword \\
\hline & & DC & FL7'225' & byte \\
\hline FD & doubleword & DC & FD'225' & doubleword \\
\hline P & byte & DC & P'2934' & byte \\
\hline \multirow[t]{2}{*}{Z} & \multirow[t]{2}{*}{byte} & DC & Z'1235' & byte \\
\hline & & DC & ZL2'1235' & byte \\
\hline \multirow[t]{2}{*}{E} & \multirow[t]{2}{*}{fullword} & DC & E'1.25' & fullword \\
\hline & & DC & EL5'1.25' & byte \\
\hline \multirow[t]{2}{*}{D} & \multirow[t]{2}{*}{doubleword} & DC & 8D'95' & doubleword \\
\hline & & DC & 8DL7'95' & byte \\
\hline L & doubleword & DC & L'2.57E65' & doubleword \\
\hline LQ & quadword & DC & LQ'0.1' & quadword \\
\hline Y & halfword & DC & Y (HERE) & halfword \\
\hline A & fullword & DC & AL1 (THERE) & byte \\
\hline AD & doubleword & DC & AD (WHERE) & doubleword \\
\hline \multirow[t]{2}{*}{S} & \multirow[t]{2}{*}{halfword} & DC & S(LABEL) & halfword \\
\hline & & DC & SL2 (LABEL) & byte \\
\hline \multirow[t]{2}{*}{V} & \multirow[t]{2}{*}{fullword} & DC & V(EXTERNAL) & fullword \\
\hline & & DC & VL3(EXTERNAL) & byte \\
\hline VD & doubleword & DC & VD(BIGOTHER) & doubleword \\
\hline J & fullword & DC & J (CLASS) & fullword \\
\hline JD & doubleword & DC & JD(LARGECLASS) & doubleword \\
\hline Q & fullword & DC & QL1 (DUMMY) & byte \\
\hline QD & doubleword & DC & QD (BIGDUMMY) & doubleword \\
\hline R & fullword & DC & R(APSECT) & fullword \\
\hline RD & doubleword & DC & RD (BPSECT) & doubleword \\
\hline
\end{tabular}
1. Depends on whether or not an explicit length is specified in the constant.

The padding and truncation rules discussed below apply to single nominal values.

\section*{Padding}

If more space is specified than is needed to accommodate the binary representation of the nominal value, the extra space is padded:
- With binary zeros on the left for the binary (B), hexadecimal (X), fixed-point \((H, F)\), packed decimal ( \(P\) ), and all address ( \(A, Y, S, V, J, Q, R\) ) constants
- With sign extension for constants that support sign extension of the nominal value ( \(\mathrm{H}, \mathrm{F}, \mathrm{Y}, \mathrm{A}\) ), as described in Figure 42 on page 149
- With EBCDIC zeros on the left (X'FO') for the zoned decimal (Z) constants
- With EBCDIC spaces on the right ( \(\mathrm{X}^{\prime} 40^{\prime}\) ) for the character (C) constants
- With EBCDIC spaces on the right ( \(\mathrm{X}^{\prime} 40\) ') for the Unicode character (CU) constant prior to translation
- With double-byte spaces on the right ( \(\mathrm{X}^{\prime} 4040\) ') for the graphic (G) constants

\section*{Notes:}
1. In floating-point constants (E,D,L), the fraction is extended to occupy the extra space available.
2. Padding is on the left for all constants except the character constant and the graphic constant.

\section*{Truncation}

If less space is available than is needed to accommodate the nominal value, the nominal value is truncated and part of the constant is lost. Truncation of the nominal value is:
- On the left for the binary (B), hexadecimal (X), fixed-point (H and F), and decimal ( P and Z )
- On the right for the character (C) constant, the Unicode character (CU) constant, and the graphic (G) constant
- On the left for absolute or relocatable address (A and Y), the external address \((\mathrm{V})\), offset \((\mathrm{Q})\), length \((\mathrm{J})\) and PSECT address ( R ) constants. The actual value stored and any possible truncation is dependent on the values inserted by the linker/binder and the length of the constant.

\section*{Notes:}
1. If significant bits are lost in the truncation of fixed-point constants, error diagnostic message ASMA072E Data item too large is issued.
2. Floating-point constants (E, D, L) are not truncated. They are rounded to fit the space available-see Figure 50 on page 162 for rounding modes.
3. The above rules for padding and truncation also apply when using the bit-length modifier (see "Subfield 5: Modifier" on page 136).
4. Double-byte data in C-type constants cannot be truncated because truncation creates incorrect double-byte data. Error ASMA208E Truncation into double-byte data is not permitted is issued if such truncation is attempted.
5. Truncation of double-byte data in CU-type and G-type constants is permitted because the length modifier restrictions (see "Subfield 5: Modifier" on page 136) ensure that incorrect double-byte data cannot be created by truncation. However, truncating bit-length constants may create incorrect double-byte data.

\section*{Subfield 1: Duplication Factor}

The syntax for coding the duplication factor is shown in the subfield format on page 127

You may omit the duplication factor. If specified, it causes the nominal value or multiple nominal values specified in a constant to be generated the number of times indicated by the factor. It is applied after the nominal value or values are
assembled into the constant. Symbols used in subfield 1 need not be previously defined. This does not apply to literals.

The duplication factor can be specified by an unsigned decimal self-defining term or by an absolute expression enclosed in parentheses.

The factor must have a positive value or be equal to zero.

\section*{Notes:}
1. A duplication factor of zero is permitted, except for literals, with the following results:
- No value is assembled.
- Alignment is forced according to the type of constant specified, if no length attribute is present (see "Alignment of Constants" on page 129).
- The length attribute of the symbol naming the constant is established according to the implicitly or explicitly specified length.

When the duplication factor is zero, the nominal value may be omitted. The alignment is forced, even if the NOALIGN option is specified.

When the duplication factor is zero for a literal, the assembler issues message ASMA067S Illegal duplication factor.
2. If duplication is specified for an address constant whose nominal value contains a location counter reference, the value of the location counter reference is incremented by the length of the constant before each duplication is done (see "Address Constants-A and Y" on page 153. If the duplication factor is zero, the value of the location counter reference is not incremented by the length of each constant that would have been generated for a non-zero duplication factor. Thus, in each of the following two statements, the first generates an ASMA072E error message for "Data item too large", but the second does not:
A DC \(0 Y(0,32768-(*-A))\)
\(B \quad D C \quad Y(0,32768-(*-B))\)
However, if duplication is specified for an address-type literal constant containing a location counter reference, the value of the location counter reference is not incremented by the length of the literal before each duplication is done. The value of the location counter reference is the location of the first byte of the literal in the literal pool, and is the same for each duplication.

The location counter value is that of the instruction in which the literal appears for A-type constants, but for S-type constants it is the location where the literal actually appears.
3. The maximum value for the duplication factor is \(2^{24}-1\), or \(X^{\prime} F F F F F F^{\prime}\) bytes. If the maximum value for the duplication factor is exceeded, the assembler issues message ASMA067S Illegal duplication factor.

\section*{Subfield 2: Type}

The syntax for coding the type is shown in the subfield format on page 127 .
You must specify the type subfield. From the type specification, the assembler determines how to interpret the constant and translate it into the correct format. The type is specified by a single-letter code as shown in Figure 35, the type extension as shown in Figure 36.

Further information about these constants is provided in the discussion of the constants themselves under "Subfield 6: Nominal Value" on page 140.

Figure 35. Type Codes for Constants
\begin{tabular}{lll}
\hline Code & Constant Type & Machine Format \\
\hline C & Character & 8-bit code for each character \\
\hline G & Graphic & 16-bit code for each character \\
\hline X & Hexadecimal & 4-bit code for each hexadecimal digit \\
\hline B & Binary & Binary format \\
\hline F & Fixed-point & Signed, fixed-point binary format; normally a fullword \\
\hline H & Fixed-point & Signed, fixed-point binary format; normally a halfword \\
\hline E & Floating-point & Short floating-point format; normally a fullword \\
\hline D & Floating-point & Long floating-point format; normally a doubleword \\
\hline L & Floating-point & Extended floating-point format; normally two doublewords \\
\hline P & Decimal & Packed decimal format \\
\hline Z & Decimal & Zoned decimal format \\
\hline A & Address & Value of address; normally a fullword \\
\hline Y & Address & Value of address; normally a halfword \\
\hline S & Address & Base register and displacement value; a halfword \\
\hline V & Address & \begin{tabular}{l} 
Space reserved for external symbol addresses; normally a \\
fullword
\end{tabular} \\
\hline J & Address & \begin{tabular}{l} 
Space reserved for length of class or DXD; normally a \\
fullword
\end{tabular} \\
\hline Q & Address & Space reserved for external dummy section offset \\
\hline R & Address & Space reserved for PSECT addresses; normally a fullword \\
\hline
\end{tabular}

The type in conjunction with an optional type extension specification indicates to the assembler:
1. How to assemble the nominal value(s) specified in subfield 6; that is, which binary representation or machine format the object code of the constant must have.
2. At what boundary the assembler aligns the constant, if no length modifier is present.
3. How much storage the constant occupies, according to the implicit length of the constant, if no explicit length modifier is present (for details, see "Padding and Truncation of Values" on page 130.

\section*{Subfield 3: Type Extension}

The syntax for coding the type extension is shown in the subfield format on page 127

You may omit the type extension subfield. If specified, the assembler, using this field in conjunction with the type subfield, determines how to interpret the constant and translate it into the correct format. The type extension is specified by a single-letter code as shown in Figure 36.

Figure 36. Type Extension Codes for Constants
\begin{tabular}{lll} 
Type & \begin{tabular}{l} 
Type \\
Extension
\end{tabular} & Description
\end{tabular}
\(\frac{\mathrm{Q}}{\mathrm{R}}\)
\begin{tabular}{lll}
\hline C & A & ASCII character constant \\
\hline & E & EBCDIC character constant \\
\hline & U & Unicode UTF-16 character constant \\
\hline E & H & New hexadecimal floating-point constant \\
\hline & B & Binary floating-point constant \\
\hline D & H & New hexadecimal floating-point constant \\
\hline & B & Binary floating-point constant \\
\hline L & H & New hexadecimal floating-point constant \\
\hline & B & Binary floating-point constant \\
\hline & Q & Hexadecimal floating-point, quadword alignment \\
\hline F & D & Doubleword fixed-point constant \\
\hline A & D & Doubleword address constant \\
\hline V & D & Doubleword address constant \\
\hline J & D & Doubleword address constant \\
\hline Q & D & Doubleword address constant \\
\hline R & D & Doubleword address constant \\
\hline
\end{tabular}

The type extension specification, in conjunction with the type subfield, indicates to the assembler:
1. How to assemble the nominal value(s) specified in subfield 6 ; that is, which binary representation or machine format the object code of the constant must have.
2. At what boundary the assembler aligns the constant, if no length modifier is present.
3. How much storage the constant occupies, according to the implicit length of the constant, if no explicit length modifier is present (for details, see "Padding and Truncation of Values" on page 130.

\section*{Subfield 4: Program type}

The syntax for coding the program type is shown in the subfield format on page 127

You may omit the program type subfield. If specified, the assembler assigns the value to the symbol naming the DC instruction, if a symbol was present. It can be specified as a decimal, character, hex or binary self-defining term and is stored as a 32-bit value. The value is not used in any way by the assembler, and may be queried by using the SYSATTRP built-in function.

The program type is specified within a P prefixed set of parenthesis -P() . For example:
```

Prog1 DC CP(7)'Perth' Program type is 7
Prog2 DC 3XP(C'APC')'FF' Program type is C'APC'

```

Symbols used in subfield 4 need not be previously defined, except in literals. For example:
```

SYM DC
FP(Rate5)'35.92'
Rate5 EQU
5

```

All expressions in program type must be evaluatable when the DC is processed.
If program type is omitted, the assembler assigns a null to the program type, and querying the value using the SYSATTRP build-in function returns a null value.

\section*{Subfield 5: Modifier}

The syntax for coding the modifier is shown in the subfield format on page 127
You may omit the modifier subfield. Modifiers describe the length in bits or bytes you want for a constant (in contrast to an implied length), and the scaling and exponent for the constant.

The three modifiers are:
- The length modifier (L), that explicitly defines the length in bytes you want for a constant. For example:
```

LENGTH DC XL10'FF'

```
- The scale modifier (S), that is only used with the fixed-point or floating-point constants (for details, see "Scale Modifier" on page 138). For example:
```

SCALE DC
FS8'35.92'

```
- The exponent modifier (E), that is only used with fixed-point or floating-point constants, and indicates the power of 10 by which the constant is to be multiplied before conversion to its internal binary format. For example:
```

EXPON DC EE3'3.414'

```

If multiple modifiers are used, they must appear in this sequence: length, scale, and exponent. For example:
```

ALL3 DC DL7S3E50'2.7182'

```

Symbols used in subfield 5 need not be previously defined, except in literals. For example:
\begin{tabular}{lll} 
SYM & DC & FS \((X)^{\prime} 35.92^{\prime}\) \\
\(X\) & EQU & 7
\end{tabular}

\section*{Length Modifier}

The length modifier indicates the number of bytes of storage into which the constant is to be assembled. It is written as \(L n\), where \(n\) is either a decimal self-defining term or an absolute expression enclosed by parentheses. It must have a positive value.

When the length modifier is specified:
- Its value determines the number of bytes of storage allocated to a constant. It therefore determines whether the nominal value of a constant must be padded or truncated to fit into the space allocated (see "Padding and Truncation of Values" on page 130.
- No boundary alignment, according to constant type, is provided (see "Alignment of Constants" on page 129.
- Its value must not exceed the maximum length allowed for the various types of constant defined.
- The length modifier must not truncate double-byte data in a C-type constant, except for bit-length modifiers.
- The length modifier must be a multiple of 2 in a G-type or CU-type constant.

When no length is specified, for character and graphic constants ( \(C\) and \(G\) ), hexadecimal constants \((X)\), binary constants \((B)\), and decimal constants ( P and Z ), the whole constant is assembled into its implicit length.

Bit-Length Modifier: The length modifier can be specified to indicate the number of bits into which a constant is to be assembled. The bit-length modifier is written as L.n where \(n\) is either a decimal self-defining term, or an absolute expression enclosed in parentheses. It must have a positive value. Such a modifier is sometimes called a "bit-length" modifier, to distinguish it from a "byte-length" modifier. You may not combine byte-length and bit-length modifiers. For example, a 12-bit field must be written L.12, not L1.4.

The value of \(n\) must lie between 1 and the number of bits (a multiple of 8) that are required to make up the maximum number of bytes allowed in the type of constant being defined. The bit-length modifier can never be used with the CU-, G-, S-, V-, R-, J- and Q-type constants, and cannot be used with the A-type or Y-type constant if the operand is simply or complexly relocatable.

When only one operand and one nominal value are specified in a DC instruction, the following rules apply:
1. The bit-length modifier allocates a field into which a constant is to be assembled. The field starts at a byte boundary and can run over one or more byte boundaries, if the bit length is greater than 8.

If the field does not end at a byte boundary and if the bit length is not a multiple of 8 , the remainder of the last byte is filled with binary zeros. For example, DC FL. 12'-1' generates X'FFF0'.
2. The nominal value of the constant is assembled into the field:
a. Starting at the high order end for the C-, E-, D-, and L-type constants
b. Starting at the low-order end for the remaining types of constants that support a bit-length modifier
3. The nominal value is padded or truncated to fit the field (see "Padding and Truncation of Values" on page 130.
Note that "padding" is not the same as "filling." In padding, the designated bit field is completed according to the rules for the constant type. Filling is always binary zeros placed at the right end of an incomplete byte.
C-type character constants are padded with EBCDIC spaces (hexadecimal X'40', and CA-type character constants are padded with ASCII spaces (hexadecimal \(\mathrm{X}^{\prime} 20^{\prime}\) ). Other constant types are padded either by sign extension or with zeros, according to the type of the constant.

The length attribute value of the symbol naming a DC instruction with a specified bit length is equal to the minimum number of integral bytes needed to contain the bit length specified for the constant. Consider the following example:

> TRUNCF DC FL.12'276'

L'TRUNCF is equal to 2. Thus, a reference to TRUNCF addresses both the two bytes that are assembled.

When more than one operand is specified in a DC instruction, or more than one nominal value in a DC operand, the above rules about bit-length modifiers also apply, except:
1. The first field allocated starts at a byte boundary, but the succeeding fields start at the next available bit. For example, BL1 DC FL. \(12^{\prime}-1,1000^{\prime}\) generates X'FFF3E8'.
2. After all the constants have been assembled into their respective fields, the bits remaining to make up the last byte are filled with zeros. For example, BL2 DC FL. 12'-1,1000,-2' generates X'FFF3E8FFE0'

If duplication is specified, filling with zeros occurs once at the end of all the fields occupied by the duplicated constants. For example, BL3 DC 3FL.12'-2' generates X'FFEFFEFFE0'.
3. The length attribute value of the symbol naming the DC instruction is equal to the number of integral bytes needed to contain the bit length specified for the first constant to be assembled. For example, the symbols BL1, BL2, and BL3 in the preceding examples each have length attribute 2.

For Double-Byte Data in C-Type Constants: If bit-lengths are specified, with a duplication factor greater than 1, and a bit-length which is not a multiple of 8 , then the double-byte data is no longer valid for devices capable of presenting DBCS characters. No error message is issued.

Storage Requirement for Constants: The total amount of storage required to assemble a DC instruction is the sum of:
1. The requirements for the individual DC operands specified in the instruction. The requirement of a DC operand is the product of:
- The sum of the lengths (implicit or explicit) of each nominal value
- The duplication factor, if specified
2. The number of bytes skipped for the boundary alignment between different operands; such skipped bytes are filled with binary zeros.

\section*{Scale Modifier}

The scale modifier specifies the amount of internal scaling that you want for:
- Binary digits for fixed-point constants (H, F)
- Hexadecimal digits for floating-point constants (E, D, L)

The scale modifier can be used only with the above types of constants. It cannot be used with EB, DB, and LB floating point constants.

The range for each type of constant is:
\[
-187 \text { to }+346
\]
\begin{tabular}{ll} 
Floating-point constant E, EH & 0 to 14 \\
Floating-point constant D, DH & 0 to 14 \\
Floating-point constant L, LH & 0 to 28
\end{tabular}

The scale modifier is written as \(S n\), where \(n\) is either a decimal self-defining term, or an absolute expression enclosed in parentheses. Both forms of the modifier's value \(n\) can be preceded by a sign; if no sign is present, a plus sign is assumed.

Scale Modifier for Fixed-Point Constants: The scale modifier for fixed-point constants specifies the power of two by which the fixed-point constant must be multiplied after its nominal value has been converted to its binary representation, but before it is assembled in its final scaled form. Scaling causes the binary point to move from its assumed fixed position at the right of the rightmost bit position.

\section*{Notes:}
1. When the scale modifier has a positive value, it indicates the number of binary positions occupied by the fractional portion of the binary number.
2. When the scale modifier has a negative value, it indicates the number of binary positions deleted from the integer portion of the binary number.
3. When low-order positions are lost because of scaling (or lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position saved.

Scale Modifier for Hexadecimal Floating-Point Constants: The scale modifier for hexadecimal floating-point constants must have a positive value. It specifies the number of hexadecimal positions that the fractional portion of the binary representation of a floating-point constant is shifted to the right. The hexadecimal point is assumed to be fixed at the left of the leftmost position in the fractional field. When scaling is specified, it causes an unnormalized hexadecimal fraction to be assembled (unnormalized means the leftmost positions of the fraction contain hexadecimal zeros). The magnitude of the constant is retained, because the exponent in the characteristic portion of the constant is adjusted upward accordingly. When non-zero hexadecimal positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost position saved.

\section*{Exponent Modifier}

The exponent modifier specifies the power of 10 by which the nominal value of a constant is to be multiplied before it is converted to its internal binary representation. It can only be used with the fixed-point ( \(H\) and \(F\) ) and floating-point ( \(\mathrm{E}, \mathrm{D}\), and L ) constants. The exponent modifier is written as En, where \(n\) can be either a decimal self-defining term, or an absolute expression enclosed in parentheses.

The decimal self-defining term or the expression can be preceded by a sign. If no sign is present, a plus sign is assumed. The range for the exponent modifier is -85 to +75 . If a type extension is used to define a floating-point constant, the exponent modifier can be in the range \(-2^{31}\) to \(2^{31}-1\). If the nominal value cannot be represented exactly, a warning message is issued.

\section*{Notes:}
1. Don't confuse the exponent modifier with the exponent that can be specified in the nominal value subfield of fixed-point and floating-point constants.
The exponent modifier affects each nominal value specified in the operand, whereas the exponent written as part of the nominal value subfield only affects the nominal value it follows. If both types of exponent are specified in a DC operand, their values are added together before the nominal value is converted to binary form. However, this sum must lie within the permissible range of -85 to +75 , unless a type extension is specified.
2. The value of the constant, after any exponents have been applied, must be contained in the implicitly or explicitly specified length of the constant to be assembled.

\section*{Subfield 6: Nominal Value}

The syntax for coding the nominal value is shown in the subfield format on page 127

You must specify the nominal value subfield unless a duplication value of zero is specified. It defines the value of the constant (or constants) described and affected by the subfields that precede it. It is this value that is assembled into the internal binary representation of the constant. Figure 37 shows the formats for specifying constants.

Figure 37. Specifying Constant Values
\begin{tabular}{llll|}
\hline \begin{tabular}{l} 
Constant \\
Type
\end{tabular} & \begin{tabular}{l} 
Single \\
Nominal Value
\end{tabular} & \begin{tabular}{l} 
Multiple \\
Nominal Value
\end{tabular} & Page No. \\
\hline C & 'value' & not allowed & 142 \\
\hline G & '<.v.a.l.u.e>' & not allowed & 145 \\
\hline B & 'value' & 'value,value,...value' & 141 \\
X & & & 147 \\
H & & & 148 \\
F & & & 148 \\
P & & & 151 \\
Z & & & 151 \\
E & & & 161 \\
D & & 167 \\
L & & & 161 \\
\hline A & & & 167 \\
Y & & & 167 \\
S & & & 153 \\
\hline V & & & 153 \\
R & & & 153 \\
\hline Q & (value, & & 153 \\
\hline J & (value) & & 159 \\
\hline
\end{tabular}

As the above list shows:
- A data constant value (any type except A, Y, S, Q, J, R and V) is enclosed by single quotation marks.
- An address constant value (type A, Y, S, V, R) or an offset constant (type Q) or a length constant (type J ) is enclosed by parentheses.
- To specify two or more values in the subfield, the values must be separated by commas, and the whole sequence of values must be enclosed by the correct delimiters; that is, single quotation marks or parentheses.
- Multiple values are not permitted for character constants.

Spaces are allowed and ignored in nominal values for the quoted constant types (BDEFHLPXZ). Spaces are significant for C and G constant types.

How nominal values are specified and interpreted by the assembler is explained in each of the subsections that follow. There is a subsection for each of the following types of constant:

Binary
Character
Graphic
Hexadecimal
Fixed-Point
Decimal
Packed Decimal
Zoned Decimal
Address
Floating-Point
Literal constants are described on page 171

\section*{Binary Constant-B}

The binary constant specifies the precise bit pattern assembled into storage. Each binary constant is assembled into the integral number of bytes (see \(\mathbf{1}\) in
Figure 38) required to contain the bits specified, unless a bit-length modifier is specified.

The following example shows the coding used to designate a binary constant. BCON has a length attribute of 1 .
\begin{tabular}{lll} 
BCON & DC & B' \(^{\prime} 11011101^{\prime}\) \\
BTRUNC & DC & BL1'100100011' \(^{\prime}\) \\
BPAD & DC & BL1'101' \(^{\prime}\) \\
BFOUR & DC & \(B^{\prime} 1111010011110100^{\prime}\)
\end{tabular}

BTRUNC assembles with the leftmost bit truncated, as follows:
00100011
BPAD assembles with five zeros as padding, as follows:
00000101

Figure 38 (Page 1 of 2). Binary Constants
\begin{tabular}{llll}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & B & \\
\hline 3. Type Extension & Not Allowed & & \\
\hline 4. Program type & Allowed & & \\
\hline
\end{tabular}

Figure 38 (Page 2 of 2). Binary Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline \multicolumn{4}{|l|}{5. Modifiers} \\
\hline Implicit length: & As needed & B DC B'10101111' & L'B = 11 \\
\hline (length modifier not present) & & C DC B'101' & \(L^{\prime} \mathrm{C}=1 \mathbf{1}\) \\
\hline Alignment: & Byte & & \\
\hline \multirow[t]{2}{*}{Range for length:} & 1 to 256 (byte length) & & \\
\hline & . 1 to 2048 (bit length) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline \multicolumn{4}{|l|}{6. Nominal value} \\
\hline Represented by: & Binary digits (0 or 1) & & \\
\hline Enclosed by: & Single quotation marks & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & With zeros at left & & \\
\hline Truncation of assembled value: & At left & & \\
\hline
\end{tabular}

\section*{Character Constant-C}

The character constant specifies character strings, such as error messages, identifiers, or other text, that the assembler converts into binary representations. If no type extension is provided, then the constant may be changed, depending on the value of the TRANSLATE option. If the type extension of "E" is provided, then the representation is also EBCDIC, but it cannot be changed by the TRANSLATE option. For information about type extension "A" see "ASCII data in character constants" on page 144 and for information about type extension "U" see "Unicode UTF-16 data from character constants" on page 144.

Any of the 256 characters from the EBCDIC character set may be designated in a character constant. Each character specified in the nominal value subfield is assembled into one byte (see \(\mathbf{1}\) in Figure 39 on page 143 .

A null nominal value is permitted if a length is specified. For example:
\[
D C \quad C L 3^{\prime \prime}
\]
is assembled as three EBCDIC spaces with object code \(X^{\prime} 404040^{\prime}\), whereas

> DC

CAL3' \({ }^{\prime}\)
is assembled as three ASCII spaces with object code \(X^{\prime} 202020^{\prime}\).
Multiple nominal values are not allowed because a comma in the nominal value is considered a valid character (see 2 in Figure 39) and is assembled into its binary
(EBCDIC) representation (see Appendix D, "Standard Character Set Code Table" on page 421). For example:
\[
D C \quad C^{\prime} A, B^{\prime}
\]
is assembled as \(A, B\) with object code \(X^{\prime} C 16 B C 2{ }^{\prime}\).
Give special consideration to representing single quotation marks and ampersands as characters. Each single quotation mark or ampersand you want as a character in the constant must be represented by a pair of single quotation marks or ampersands. Each pair of single quotation marks is assembled as one single quotation mark, and each pair of ampersands is assembled as one ampersand (see 3 in Figure 39).

Figure 39 (Page 1 of 2). Character Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & C & & \\
\hline \multirow[t]{3}{*}{3. Type Extension} & U & C DC CU'UNICODE' & \(L^{\prime} \mathrm{C}=14\) \\
\hline & A & A DC CA'ASCII' & L'C \(=5\) \\
\hline & C & E DC CE'EBCDIC' & L'E = 5 \\
\hline 4. Program type & Allowed & & \\
\hline \multicolumn{4}{|l|}{5. Modifiers} \\
\hline Implicit length: (length modifier not present) & Evaluate as an even number, if Type Extension of \(U\) is specified & C DC C'LENGTH' & \(L^{\prime} \mathrm{C}=6 \quad 1\) \\
\hline Alignment: & Byte & & \\
\hline \multirow[t]{3}{*}{Range for length:} & 1 to 256 (byte length) & & \\
\hline & Must be a multiple of 2 when the Type Extension is \(U\) & & \\
\hline & . 1 to 2048 (bit length) (Not permitted if Type Extension of \(U\) is specified.) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline \multirow[t]{4}{*}{6. \(\frac{\text { Nominal value }}{\text { Represented by: }}\)} & & & Object code \\
\hline & Characters & DC C'A''B' & X'C17DC2' 3 \\
\hline & (all 256 8-bit & DC CU'AA' & X'00410041' \\
\hline & combinations) & DC CA'AB' & X'4142' \\
\hline Enclosed by: & Single quotation marks & & \\
\hline Exponent allowed: & No (would be interpreted as character data) & & \\
\hline
\end{tabular}

Figure 39 (Page 2 of 2). Character Constants
\begin{tabular}{lllll}
\hline Subfield & Value & Example & Result \\
\hline \begin{tabular}{l} 
Number of values \\
per operand:
\end{tabular} & One & DC C'A, B' & \begin{tabular}{l} 
Object code \\
\(X^{\prime} C 16 B C 2 ' ~\)
\end{tabular} & \(\mathbf{2}\) \\
\hline Padding: & \begin{tabular}{l} 
With spaces at \\
right \(\left(X^{\prime} 40^{\prime}\right.\) \\
EBCDIC, \(X^{\prime} 20^{\prime}\)
\end{tabular} & & & \\
& ASCII) & & \\
\hline \begin{tabular}{l} 
Truncation of \\
assembled value:
\end{tabular} & At right & & \\
\hline
\end{tabular}

In the following example, the length attribute of FIELD is 12 :
```

FIELD DC C'TOTAL IS 110'

```

However, in this next example, the length attribute is 15 , and three spaces appear in storage to the right of the zero:
```

FIELD DC CL15'TOTAL IS 110'

```

In the next example, the length attribute of FIELD is 12, although 13 characters appear in the operand. The two ampersands are paired, and so count as only one byte.
FIELD DC
C'TOTAL IS \&\&10'

In the next example, a length of 4 has been specified, but there are five characters in the constant.
```

FIELD DC 3CL4'ABCDE'

```

The generated constant would be:
```

ABCDABCDABCD

```

The same constant could be specified as a literal.
\[
\text { MVC } \quad \operatorname{AREA}(12),=3 C L 4^{\prime} A B C D E '
\]

On the other hand, if the length modifier had been specified as 6 instead of 4 , the generated constant would have been:
ABCDE ABCDE ABCDE (with one trailing space)
ASCII data in character constants: For Character ASCII (CA) constants the character string is converted to ASCII, using the codepage 37 table. Any paired occurrences of ampersands and apostrophes are converted to a single occurrence of such a character prior to conversion. The assembler then maps each EBCDIC character into its ASCII equivalent. This constant is not modified by the TRANSLATE option.

Unicode UTF-16 data from character constants: For Character Unicode (CU) constants the value is converted to Unicode UTF-16 using the code page identified via the CODEPAGE assembler option. Any paired occurrences of ampersands and apostrophes are converted to a single occurrence of such a character prior to conversion. If necessary the value is padded with EBCDIC spaces on the right
( \(\mathrm{X}^{\prime} 40^{\prime}\) ). The assembler then maps each EBCDIC character into its 2-byte Unicode UTF-16 equivalent.

For example:
\begin{tabular}{llll} 
UA & DC & CU'UTF-16' & object code X' 005500540046002D00310036' \\
UB & DC & CUL4'L' & object code \(X^{\prime} 004 C 0020^{\prime}\) \\
UC & DC & CUL2'XYZ' & object code \(X^{\prime} 0058^{\prime}\)
\end{tabular}

Double-byte data in character constants: When the DBCS assembler option is specified, double-byte data may be used in a character constant. The start of double-byte data is delimited by SO, and the end by SI. All characters between SO and SI must be valid double-byte characters. No single-byte meaning is drawn from the double-byte data. Hence, special characters such as the single quotation mark and ampersand are not recognized between SO and SI. The SO and SI are included in the assembled representation of a character constant containing double-byte data.

If a duplication factor is used, \(\mathrm{SI} / \mathrm{SO}\) pairs at the duplication points are not removed. For example, the statement:
```

DBCS DC 3C'<D1>'

```
results in the assembled character string value of:
```

<D1><D1><D1>

```

Null double-byte data (SO followed immediately by SI ) is acceptable and is assembled into the constant value.

The following examples of character constants contain double-byte data:
\begin{tabular}{lll} 
DBCS0 & DC & \(C^{\prime}<>^{\prime}\) \\
DBCS1 & DC & \(C^{\prime}<. D . B \cdot C . S>{ }^{\prime}\) \\
DBCS2 & DC & \(C^{\prime} a b c<\cdot A \cdot B \cdot C>^{\prime}\) \\
DBCS3 & DC & \(C^{\prime} a b c<\cdot A \cdot B . C>d e f^{\prime}\)
\end{tabular}

The length attribute includes the SO and SI. For example, the length attribute of DBCS0 is 2, and the length attribute of DBCS2 is 11. No truncation of double-byte character strings within C-type constants is allowed, since incorrect double-byte data would be created.

It is possible to generate invalid DBCS data in these situations:
- You specify a bit-length modifier that causes truncation of the DBCS data or the shift-out and shift-in characters.
- You specify the TRANSLATE option.

\section*{Graphic Constant-G}

When the DBCS assembler option is specified, the graphic (G-type) constant is supported. This constant type allows the assembly of pure double-byte data. The graphic constant differs from a character constant containing only double-byte data in that the SO and SI delimiting the start and end of double-byte data are not present in the assembled value of the graphic constant. Because SO and SI are not assembled, if a duplication factor is used, no redundant \(\mathrm{SI} / \mathrm{SO}\) characters are created. For example, the statement:
DBCS DC \(3 G^{\prime}<D^{\prime}>^{\prime}\)
results in the assembled character string value of: D1D1D1

Examples of graphic constants are:
\begin{tabular}{|c|c|c|}
\hline DBCS1 & DC & \(\mathrm{G}^{\prime}<. A . B . C>'\) \\
\hline DBCS2 & DC & GL10 \({ }^{\text {< }}\) <.A.B.C> \\
\hline DBCS3 & DC & GL4 \({ }^{\prime}<\) A.B.C> \({ }^{\prime}\) \\
\hline
\end{tabular}

Because the length attribute does not include the SO and SI, the length attribute of DBCS1 is 6 . The length modifier of 10 for DBCS2 causes padding of 2 double-byte spaces at the right of the nominal value. The length modifier of 4 for DBCS3 causes truncation after the first 2 double-byte characters. The length attribute of a graphic constant must be a multiple of 2 .

Type Attribute of G-Type Constant: Don't confuse the G-type constant character with the type (data) attribute of a graphic constant. The type attribute of a graphic constant is \(\propto\), not \(G\). See the general discussion about data attributes on page 324 and "Type Attribute ( \(T^{\prime}\) )" on page 328 .

Figure 40. Graphic Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & DC 3G'<.A>' & Object code
\[
X^{\prime} 42 C 142 C 142 C 1 '
\] \\
\hline 2. Type & G & & \\
\hline 3. Type Extension & Not allowed & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & As needed (twice the number of DBCS characters) & GC DC G'<.A.B>' & \(L^{\prime} \mathrm{GC}=4\) \\
\hline Alignment: & Byte & & \\
\hline Range for length: & 2 to 256 , must be multiple of 2 (byte length) bit length not allowed & & \\
\hline 6. Nominal value Represented by: & DBCS characters delimited by SO and SI & \[
\begin{array}{ll}
D C & G^{\prime}<. \& . .^{\prime}> \\
D C & G^{\prime}<. A><. B>1
\end{array}
\] & \begin{tabular}{l}
Object code \\
X'4250427D' \\
\(X^{\prime} 42 \mathrm{C} 142 \mathrm{C} 2^{\prime}\)
\end{tabular} \\
\hline Enclosed by: & Single quotation marks & & \\
\hline Number of values per operand: & One & DC G \({ }^{\prime}<\). \(A, ., B>{ }^{\prime}\) & Object code
\[
X^{\prime} 42 C 1426 B 42 C 2 '
\] \\
\hline Padding: & With DBCS spaces at right ( \(\mathrm{X}^{\prime} 4040\) ') & DC GL6'<.A>' & Object code
\[
X^{\prime} 42 C 140404040^{\prime}
\] \\
\hline Truncation of assembled value: & At right & DC GL2 \({ }^{\prime}<\). \(A . B>1\) & Object code
\[
X^{\prime} 42 C 1 '
\] \\
\hline
\end{tabular}

\section*{Hexadecimal Constant-X}

Hexadecimal constants generate large bit patterns more conveniently than binary constants. Also, the hexadecimal values you specify in a source module let you compare them directly with the hexadecimal values generated for the object code and address locations printed in the program listing.

Each hexadecimal digit (see \(\mathbf{1}\) in Figure 41) specified in the nominal value subfield is assembled into four bits (their binary patterns can be found in "Self-Defining Terms" on page 34. The implicit length in bytes of a hexadecimal constant is then half the number of hexadecimal digits specified (assuming that a high-order hexadecimal zero is added to an odd number of digits). See 2 and 3 in Figure 41.

An 8-digit hexadecimal constant provides a convenient way to set the bit pattern of a full binary word. The constant in the following example sets the first and third bytes of a word with all 1 bits.
\begin{tabular}{lll} 
& DS & \(0 F\) \\
TEST & DC & \(X^{\prime} F F 00 F F 00^{\prime}\)
\end{tabular}

The DS instruction sets the location counter to a fullword boundary. (See "DS Instruction" on page 174.)

The next example uses a hexadecimal constant as a literal and inserts a byte of all 1 bits into the rightmost 8 bits of register 5 .
\[
\text { IC } \quad 5,=X^{\prime} F F^{\prime}
\]

In the following example, the digit A is dropped, because 5 hexadecimal digits are specified for a length of 2 bytes:
ALPHACON DC
3XL2'A6F4E'
Generates 6F4E 3 times

The resulting constant is 6F4E, which occupies the specified 2 bytes. It is duplicated three times, as requested by the duplication factor. If it had been specified as:
ALPHACON DC 3X'A6F4E' Generates 0A6F4E 3 times
the resulting constant would have a hexadecimal zero in the leftmost position. 0A6F4E0A6F4E0A6F4E

Figure 41 (Page 1 of 2). Hexadecimal Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & X & & \\
\hline 3. Type Extension & Not allowed & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & As needed & \[
\begin{array}{lll}
X & D C & X^{\prime} F F 00 A 2^{\prime} \\
Y & D C & X^{\prime} F 00 A 2^{\prime}
\end{array}
\] & \[
\begin{aligned}
& L^{\prime} X=3 \mathbf{2} \\
& L^{\prime} Y=3 \mathbf{2}
\end{aligned}
\] \\
\hline Alignment: & Byte & & \\
\hline
\end{tabular}

Figure 41 (Page 2 of 2). Hexadecimal Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline \multirow[t]{2}{*}{Range for length:} & 1 to 256 (byte length) & & \\
\hline & . 1 to .2048 (bit length) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline 6. Nominal value Represented by: & Hexadecimal digits ( 0 to 9 and A to F) & \[
\begin{aligned}
& D C \quad X^{\prime} 1 F^{\prime} \\
& D C \quad X^{\prime} 91 F^{\prime}
\end{aligned}
\] & \[
\begin{aligned}
& \text { Object code } \\
& \text { X'1F' }^{\prime} \\
& \text { X'091F' }^{\prime}
\end{aligned}
\] \\
\hline Enclosed by: & Single quotation marks & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & With zeros at left & & \\
\hline Truncation of assembled value: & At left & & \\
\hline
\end{tabular}

\section*{Fixed-Point Constants-F and H}

Fixed-point constants let you introduce data that is in a form suitable for the arithmetic operations of the binary fixed-point machine instructions. The constants you define can also be automatically aligned to the correct doubleword, fullword or halfword boundary for the instructions that refer to addresses on these boundaries (unless the NOALIGN option has been specified; see "General Information About Constants" on page 129. You can do algebraic operations using this type of constant because they can have positive or negative values.

A fixed-point constant is written as a decimal number, which can be followed by a decimal exponent. The format of the constant is as follows:
1. The nominal value can be a signed (see \(\mathbf{1}\) in Figure 42 on page 149 integer, fraction, or mixed number (see 2 Figure 42) followed by a signed exponent (see 3 in Figure 42). If a sign is not specified for either the number or exponent, + is assumed.
2. The exponent must lie within the permissible range (see 4 in Figure 42). If an exponent modifier is also specified, the algebraic sum (see 5 in Figure 42) of the exponent and the exponent modifier must lie within the permissible range.

Some examples of the range of values that can be assembled into fixed-point constants are given below:
\begin{tabular}{cc} 
Length & \begin{tabular}{c} 
Range of values that can \\
be assembled
\end{tabular} \\
8 & \(-2^{63}\) to \(2^{263-1}\) \\
4 & \(-2^{231}\) to \(2^{231-1}\) \\
2 & \(-2^{15}\) to \(2^{15-1}\) \\
1 & \(-2^{7}\) to \(2^{7-1}\)
\end{tabular}

The range of values depends on the implicitly or explicitly specified length (if scaling is disregarded). If the value specified for a particular constant does not lie within the allowable range for a given length, the constant is not assembled, but flagged as an error.

A fixed-point constant is assembled as follows:
1. The specified number, multiplied by any exponents, is converted to a binary number.
2. Scaling is done, if specified. If a scale modifier is not provided, the fractional portion of the number is lost.
3. The binary value is rounded, if necessary. The resulting number does not differ from the exact number specified by more than one in the least significant bit position at the right.
4. A negative number is carried in two's-complement form.
5. Duplication is applied after the constant has been assembled.

The statement below generates 3 fullwords of data. The location attribute of CONWRD is the address of the first byte of the first word, and the length attribute is 4 , the implied length for a fullword fixed-point constant. The expression CONWRD+4 could be used to address the second constant (second word) in the field.
\[
\text { CONWRD DC } 3 F^{\prime} 658474^{\prime}
\]

Figure 42 (Page 1 of 2). Fixed-Point Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & \(F\) and H & & \\
\hline 3. Type Extension & D permitted with type F & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & \begin{tabular}{l}
Doubleword: 8 bytes \\
Fullword: 4 bytes \\
Halfword: 2 bytes
\end{tabular} & & \\
\hline Alignment: (Length modifier not present) & Doubleword, fullword or halfword & & \\
\hline Range for length: & \begin{tabular}{l}
1 to 8 (byte length) \\
.1 to 64 \\
(bit length)
\end{tabular} & & \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|}
\hline \multicolumn{4}{|l|}{Figure 42 (Page 2 of 2). Fixed-Point Constants} \\
\hline Subfield & Value & Example & Result \\
\hline Range for scale: & \[
\begin{aligned}
& \text { F: }-187 \text { to }+346 \\
& \text { H: }-187 \text { to }+346
\end{aligned}
\] & & \\
\hline Range for exponent: & -85 to +754 & \[
\begin{aligned}
\text { DC } \\
\mathbf{5}
\end{aligned}
\] & value \(=2 \times 10^{2}\) \\
\hline 6. Nominal value Represented by: & Decimal digits (0 to 9) & \begin{tabular}{l}
Doubleword \\
DC FD'-200' 1 \\
Fullword \\
DC FS4'2.25' 2 \\
Halfword: \\
DC \(\mathrm{H}^{\prime}+200^{\prime}\) \\
DC HS4'.25'
\end{tabular} & \\
\hline Enclosed by: & Single quotation marks & & \\
\hline Exponent allowed: & Yes & \begin{tabular}{l}
Doubleword: \\
DC FD'2E6' \\
Fullword: \\
DC F'2E6' 3 \\
Halfword: \\
DC H'2E-6'
\end{tabular} & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & With sign bits at left & & \\
\hline Truncation of assembled value: & At left (error message issued) & & \\
\hline
\end{tabular}

In the following example, the DC statement generates a 2-byte field containing a negative constant. Scaling has been specified in order to reserve 6 bits for the fractional portion of the constant.
HALFCON DC HS6'-25.46'

In the following example, the constant (3.50) is multiplied by 10 to the power -2 before being converted to its binary format. The scale modifier reserves 12 bits for the fractional portion.

FULLCON DC HS12'3.50E-2'
The same constant could be specified as a literal:
AH 7,=HS12'3.50E-2'
The final example specifies three constants. The scale modifier requests 4 bits for the fractional portion of each constant. The 4 bits are provided whether or not the fraction exists.

THREECON DC FS4'10,25.3,100'

Remember that commas separate operands. For readability, use spaces instead, as shown in this example:
\begin{tabular}{lll} 
TWOCONS DC & \(F^{\prime} 123,445^{\prime}\) & Two constants \\
ONECON DC & F' \(^{\prime} 123456^{\prime}\) & One constant
\end{tabular}

\section*{Decimal Constants-P and \(\mathbf{Z}\)}

The decimal constants let you introduce data in a form suitable for operations on decimal data. The packed decimal constants (P-type) are used for processing by the decimal instructions. The zoned decimal constants (Z-type) are in the form (EBCDIC representation) you can use as a print image, except for the digit in the rightmost byte.

The nominal value can be a signed (plus is assumed if the number is unsigned) decimal number. A decimal point may be written anywhere in the number, or it may be omitted. The placement of a decimal point in the definition does not affect the assembly of the constant in any way, because the decimal point is not assembled into the constant; it only affects the integer and scaling attributes of the symbol that names the constant.

The specified digits are assumed to constitute an integer (see \(\mathbf{1}\) in Figure 43). You may determine correct decimal point alignment either by defining data so that the point is aligned or by selecting machine instructions that operate on the data correctly (that is, shift it for purposes of decimal point alignment).

Decimal constants are assembled as follows:

Packed Decimal Constants: Each digit is converted into its 4-bit binary coded decimal equivalent (see 2 in Figure 43). The sign indicator (see 3 in Figure 43) is assembled into the rightmost four bits of the constant.

Zoned Decimal Constants: Each digit is converted into its 8-bit EBCDIC representation (see 4 in Figure 43). The sign indicator (see 5 in Figure 43) replaces the first four bits of the low-order byte of the constant.

The range of values that can be assembled into a decimal constant is shown below:
\begin{tabular}{ll}
\begin{tabular}{l} 
Type of \\
decimal \\
constant
\end{tabular} & \begin{tabular}{l} 
Range of values that \\
can be specified
\end{tabular} \\
\hline Packed & \(1031-1\) to -1031 \\
Zoned & \(1016-1\) to -1016
\end{tabular}

For both packed and zoned decimals, a plus sign is translated into the hexadecimal digit \(C\), a minus sign into the digit \(D\). The packed decimal constants (P-type) are used for processing by the decimal instructions.

If, in a constant with an implicit length, an even number of packed decimal digits is specified, one digit is left unpaired because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits are set to zeros and the rightmost four bits contain the unpaired (first) digit.

Figure 43. Decimal Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & P and Z & & \\
\hline 3. Type Extension & Not allowed & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & As needed & \begin{tabular}{l}
Packed: \\
P DC P'+593' \\
Zoned: \\
Z DC Z'-593'
\end{tabular} & \[
\begin{aligned}
& L^{\prime} P=2 \\
& L^{\prime} Z=3
\end{aligned}
\] \\
\hline Alignment: & Byte & & \\
\hline Range for length: & 1 to 16 (byte length) & & \\
\hline
\end{tabular}
.1 to .128
(bit length)
\begin{tabular}{|c|c|c|c|}
\hline Range for scale: & \multicolumn{3}{|l|}{Not allowed} \\
\hline Range for exponent: & \multicolumn{3}{|l|}{Not allowed} \\
\hline \multicolumn{4}{|l|}{6. Nominal value} \\
\hline \multirow[t]{8}{*}{Represented by:} & \multirow[t]{8}{*}{Decimal digits (0 to 9)} & Packed: & Object code \\
\hline & & DC P'5.5' 1 & X'055C' \\
\hline & & DC P'55' 1 & X'055C' \\
\hline & & DC P'+555' 2 & X'555C' 3 \\
\hline & & DC P'-777' & X'777D' 3 \\
\hline & & & Object code \\
\hline & & Zoned: & X'F5F5D5' 5 \\
\hline & & DC Z'-555' 4 & \\
\hline
\end{tabular}
\begin{tabular}{ll}
\hline Enclosed by: & \begin{tabular}{l} 
Single quotation \\
marks
\end{tabular} \\
\hline Exponent allowed: & No \\
\hline \begin{tabular}{l} 
Number of values \\
per operand:
\end{tabular} & Multiple \\
\hline Padding: & \begin{tabular}{l} 
Packed: \\
with binary \\
zeros at left
\end{tabular} \\
& \begin{tabular}{l} 
Zoned: \\
with EBCDIC \\
zeros (X'FO') \\
at left
\end{tabular} \\
\hline Truncation of & At left \\
assembled value: &
\end{tabular}

In the following example, the DC statement specifies both packed and zoned decimal constants. The length modifier applies to each constant in the first operand (that is, to each packed decimal constant). A literal could not specify both operands.

DECIMALS DC PL8'+25.8,-3874,+2.3', Z'+80,-3.72'
The last example shows the use of a packed decimal literal.
UNPK OUTAREA, \(=\) PL2 \({ }^{\prime}+25^{\prime}\)

\section*{Address Constants}

An address constant is an absolute or relocatable expression, such as a storage address, that is translated into a constant. Address constants can be used for initializing base registers to facilitate the addressing of storage. Furthermore, they provide a means of communicating between control sections of a multisection program. However, storage addressing and control section communication also depends on the USING assembler instruction and the loading of registers. See "USING Instruction" on page 218.

The nominal value of an address constant, unlike other types of constants, is enclosed in parentheses. If two or more address constants are specified in an operand, they are separated by commas, and the whole sequence is enclosed by parentheses. There are seven types of address constants: A, Y, S, R, Q, J and V. A relocatable address constant may not be specified with bit lengths.

Complex Relocatable Expressions: A complex relocatable expression can only specify an A-or Y-type address constant. These expressions contain two or more unpaired relocatable terms, or two or more negative relocatable terms in addition to any absolute or paired relocatable terms. A complex relocatable expression might consist of external symbols and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the address constant.

The following example shows how, and why, a complex relocatable expression could be used for an A or \(Y\) address constant:

B DC \(\quad A(X-*)\) Offset from B to \(X\)
Address Constants-A and Y: The following sections describe how the different types of address constants are assembled from expressions that usually represent storage addresses, and how the constants are used for addressing within and between source modules.

In the A-type and Y-type address constants, you can specify any of the three following types of assembly-time expressions whose values the assembler then computes and assembles into object code. Use this expression computation as follows:
- Relocatable expressions for addressing
- Absolute expressions for addressing and value computation
- Complex relocatable expressions to relate addresses in different source modules

Literals, which are relocatable forms, are not allowed as operands, but length, scale and integer attribute references to literals are allowed.

Here are some examples:
\begin{tabular}{ll}
\(D C\) & \(A\left(L^{\prime}=F^{\prime} 1.23^{\prime}\right)\) \\
\(D C\) & \(A\left(I^{\prime}=F^{\prime} 3.45^{\prime}\right)\) \\
\(D C\) & \(A\left(S^{\prime}=F S 6^{\prime} 7.89\right)\)
\end{tabular}

\section*{Notes:}
1. No bit-length modifier (see 1 in Figure 44) is allowed when a relocatable or complex relocatable expression (see \(\mathbf{2}\) in Figure 44) is specified. The only explicit lengths that can be specified with relocatable or complex relocatable address constants are:
- 2 through 8 bytes for AD-type constants
- 2, 3, or 4 bytes for A-type constants
- 2 bytes for Y-type constants

The linkage editor/binder/loader you use determines which lengths are actually supported. Please see the appropriate product manual for more information.

For absolute operands, you may specify byte or bit lengths:
- byte lengths 1 through 8, or bit lengths . 1 through .128, for A-type constants
- byte lengths 1 or 2, or bit lengths . 1 through .16, for Y-type constants
2. The value of the location counter reference (*) when specified in an address constant varies from constant to constant, if any of the following, or a combination of the following, are specified:
- Multiple operands
- Multiple nominal values (see 3 in Figure 44)
- A duplication factor (see 4 in Figure 44)

The location counter is incremented with the length of the previously assembled constant.
3. When the location counter reference occurs in a literal address constant, the value of the location counter is the address of the first byte of the instruction.

Note that the behavior of location counter references in A-type address constants is different from that in S-type address constants "Address Constant-S" on page 156.

Figure 44 (Page 1 of 2). A and \(Y\) Address Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & A DC 5AL1 \((*-A)\) & 4 Object code X'0001020304' \\
\hline 2. Type & A and \(Y\) & & \\
\hline 3. Type Extension & D permitted for \(A\) type only & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & A-type: 4 bytes AD-type: 8 bytes Y-type: 2 bytes & & \\
\hline Alignment: (Length modifier not present) & A-type: fullword AD-type: doubleword Y-type: halfword & & \\
\hline
\end{tabular}

Figure 44 (Page 2 of 2). A and \(Y\) Address Constants
\begin{tabular}{|c|c|c|}
\hline Subfield & Value & Example Result \\
\hline Range for length: & \begin{tabular}{l}
A-type: \\
2 to 4 1 \\
(byte length) \\
AD-type: \\
2 to 8 \\
(byte length) \\
Y-type: \\
2 only \\
(byte length)
\end{tabular} & \\
\hline Range for scale: & Not allowed & \\
\hline Range for exponent: & Not allowed & \\
\hline 6. Nominal value Represented by: & Absolute, relocatable, or complex relocatable expressions 2 & \begin{tabular}{l}
A-type: \\
DC A(ABSOL+10) \\
Y-type: \\
DC \(Y(\) RELOC +32 ) \\
A DC \(Y(*-A, *+4) 3\) values \(=0, A+6\)
\end{tabular} \\
\hline Enclosed by: & Parentheses & \\
\hline Exponent allowed: & No & \\
\hline Number of values per operand: & Multiple & \\
\hline Padding: & If an absolute term is present, by sign extension. Otherwise, with zeros at left. & \\
\hline Truncation of assembled value: & At left & \\
\hline
\end{tabular}

Take care when using Y-type address constants and 2-byte A-type address constants for relocatable addresses, as they can only address a maximum of 65,536 bytes of storage. Using these types of address constants for relocatable addresses results in message ASMA066W being issued unless the assembler option RA2 is specified.

The A-type and Y -type address constants are processed as follows: If the nominal value is an absolute expression, it is computed to its 32 -bit value and then truncated or sign-extended on the left to fit the implicit or explicit length of the constant. If the nominal value is a relocatable or complex relocatable expression, it is not completely evaluated until linkage edit time. The relocated address values are then placed in the fields set aside for them at assembly time by the A-type and Y-type constants.

In the following examples, the field generated from the statement named ACON contains four constants, each of which occupies four bytes. The statement containing the LM instruction shows the same set of constants specified as literals (that is, address constant literals).

ACON DC A(108,LOP,END-STRT,*+4096)
LM \(4,7,=A(108\), LOP ,END-STRT,\(*+4096)\)
A location counter reference (*) appears in the fourth constant ( \(*+4096\) ). The value of the location counter is the address of the first byte of the fourth constant. When the location counter reference occurs in a literal, as in the LM instruction, the value of the location counter is the address of the first byte of the instruction.

Note: It is important to remember that expression evaluation for address constants is restricted to using 32-bit internal arithmetic. The result is then sign-extended to the length of the constant. This means that certain expressions in AD-type constants may not yield expected results, especially if the resulting value is negative.

Address Constant—S: Use the S-type address constant to assemble an explicit address in base-displacement form. You can specify the explicit address yourself or let the assembler compute it from an implicit address, using the current base register and address in its computation.

The nominal values can be specified in two ways:
1. As one absolute or relocatable expression (see 1 in Figure 45 on page 157) representing an implicit address.
2. As two absolute expressions (see 2 in Figure 45) the first of which represents the displacement and the second, enclosed in parentheses, represents the base register.

The address value represented by the expression in 1 in Figure 45, is converted by the assembler into the correct base register and displacement value. An S-type constant is assembled as a halfword and aligned on a halfword boundary. The leftmost four bits of the assembled constant represent the base register designation; the remaining 12 bits, the displacement value.

\section*{Notes:}
1. The value of the location counter (*) when specified in an S-type address constant varies from constant to constant if one or more the following is specified:
- Multiple operands
- Multiple nominal values
- A duplication factor

In each case the location counter is incremented with the length of the previously assembled constant, except when multiple S-type address constants are specified in a literal. In a literal, the same location counter value is used for each of the multiple values.
2. If a length modifier is used, only 2 bytes may be specified.
3. S-type address constants can be specified as literals. The USING instructions used to resolve them are those in effect at the place where the literal pool is assembled, and not where the literal is used.
4. The location counter value used in the literal is the value at the point where the literal is used, not where it is defined.

For example:
```

USING *,15
DC 2S(*) generates F000F002
LA 1,=2S(*) generated constants are F004F004

```

Note that this behavior is different from that in A-type address constants and Y-type address constants.

Figure 45. S Address Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & S & & \\
\hline 3. Type Extension & Not allowed & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & 2 bytes & & \\
\hline Alignment: (Length modifier not present) & Halfword & & \\
\hline Range for length: & 2 only (no bit length) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline 6. Nominal value Represented by: & Absolute or relocatable expression 1 & \[
\begin{array}{ll}
\text { DC } & \text { S(RELOC) } \\
\text { DC } & S(1024)
\end{array}
\] & \begin{tabular}{ll} 
Base & Disp \\
\(X\) & \(Y Y Y\) \\
0 & 400
\end{tabular} \\
\hline & Two absolute expressions 2 & DC \(\mathrm{S}(512(12))\) & C 200 \\
\hline Enclosed by: & Parentheses & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & Not applicable & & \\
\hline Truncation of assembled value: & Not applicable & & \\
\hline
\end{tabular}

Address Constant-V: The V-type constant reserves storage for the address of a location in a control section that is defined in another source module. Use the V-type address constant only to branch to an external address, because link-time processing may cause the branch to be indirect (for example, an assisted linkage in an overlay module). That is, the resolved address in a V-type address constant might not contain the address of the referenced symbol. In contrast, to refer to external data you should use an A-type address constant whose nominal value specifies an external symbol identified by an EXTRN instruction.

Because you specify a symbol in a V-type address constant, the assembler assumes that it is an external symbol. A value of zero is assembled into the space
reserved for the V-type constant; the correct relocated value of the address is inserted into this space by the linkage editor before your object program is loaded.

The symbol specified (see \(\mathbf{1}\) in Figure 46) in the nominal value subfield does not constitute a definition of the symbol for the source module in which the V-type address constant appears.

The symbol specified in a V-type constant must not represent external data in an overlay program.

Figure 46. V Address Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & V & & \\
\hline 3. Type Extension & D & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & V-type: 4 bytes VD-type: 8 bytes & VL4(ExtSym) & \\
\hline Alignment: (Length modifier not present) & V-type: Fullword VD-type: Doubleword & & \\
\hline Range for length: & V-type: 4 or 3 only VD-type: 3, 4 or 8 (no bit length) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline 6. Nominal value Represented by: & A single external symbol & \[
\begin{array}{ll}
\mathrm{DC} & V(\text { MODA }) \boldsymbol{1} \\
\text { DC } & \text { V(EXTADR) }
\end{array}
\] & \\
\hline Enclosed by: & Parentheses & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & None & & \\
\hline Truncation of assembled value: & Not applicable & & \\
\hline
\end{tabular}

In the following example, 12 bytes are reserved, because there are three symbols. The value of each assembled constant is zero until the program is link-edited.
VCONST DC V(SORT,MERGE,CALC)

\section*{Offset Constant-Q}

Use this constant to reserve storage for the offset into a storage area of an external dummy section, or the offset to a label in a class. The offset is entered into this space by the linker. When the offset is added to the address of an overall block of storage set aside for external dummy sections, it addresses the applicable section.

For a description of the use of the Q-type offset constant in combination with an external dummy section, see "External Dummy Sections" on page 57. See also Figure 47 for details.

In the following example, to access the external dummy section named VALUE, the value of the constant labeled \(A\) is added to the base address of the block of storage allocated for external dummy sections.
```

A DC Q(VALUE)

```

The DXD or DSECT names referenced in the Q-type offset constant need not be previously defined.

Figure 47. Q Offset Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & Q & & \\
\hline 3. Type Extension & D & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & Q-type: 4 bytes QD-type: 8 bytes & Q(DXDEXT) & \\
\hline Alignment: (Length modifier not present) & Q-type: Fullword QD-type: Quadword & & \\
\hline Range for length: & Q-type: 1 to 4 bytes QD-type: 1 to 8 bytes (no bit length) & QL2(DXDEXT) & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline 6. Nominal value Represented by: & A DXD or DSECT name (an external symbol) & \begin{tabular}{l}
DC Q(DUMMYEXT) \\
DC Q(DXDEXT)
\end{tabular} & \\
\hline Enclosed by: & Parentheses & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & None & & \\
\hline Truncation of assembled value: & Not applicable & & \\
\hline
\end{tabular}

\section*{Length Constant-J}

Use this constant to reserve storage for the length of a DXD, class or DSECT. The assembler fills the field with binary zeros, and the length is entered into this space by the linker. This constant is only available if the GOFF option is specified.

In the following example, the value at \(A\) is the length of CLASS.
A
DC
J (CLASS)

The DXD or DSECT names referenced in the J-type length constant need not be previously defined.

Figure 48. J Length Constants
\begin{tabular}{|c|c|c|}
\hline Subfield & Value Example & Result \\
\hline 1. Duplication factor & Allowed & \\
\hline 2. Type & \(J\) & \\
\hline 3. Type Extension & D & \\
\hline 4. Program type & Allowed & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & J-type: 4 bytes JD-type: 8 bytes & \\
\hline Alignment: (Length modifier not present) & J-type: Fullword JD-type: Doubleword & \\
\hline Range for length: & \begin{tabular}{l}
J-type: 2 to 4 bytes, or 8 \\
JD-type: 2 to 4 bytes, or 8 (no bit length)
\end{tabular} & \\
\hline Range for scale: & Not allowed & \\
\hline Range for exponent: & Not allowed & \\
\hline 6. Nominal value Represented by: & A single DXD, class, DC J(CLASS) or DSECT name & \\
\hline Enclosed by: & Parentheses & \\
\hline Exponent allowed: & No & \\
\hline Number of values per operand: & Multiple & \\
\hline Padding: & None. & \\
\hline Truncation of assembled value: & At left & \\
\hline
\end{tabular}

\section*{PSECT Reference-R}

The R-type constant reserves storage for the address of the PSECT of symbol1 as specified in the associated XATTR statement "XATTR Instruction (MVS and CMS)" on page 230). It is the caller's responsibility to establish the definition of the R-type address constant referencing the called routine's PSECT, and to pass that address to the called routine. This constant is only available if the GOFF option is specified.

Note: If a program is to be reentrant, R-type address constants may not appear in shared (read-only) text. They should be in the caller's PSECT, and be provided to the called routine using an appropriate convention. That is, R-type address constants referring to PSECTs should themselves reside in PSECTs. If not, there can be only a single instantiation of the PSECT work area, and the program cannot be reentrant.

Figure 49. R Address Constants
\begin{tabular}{|c|c|c|c|}
\hline Subfield & Value & Example & Result \\
\hline 1. Duplication factor & Allowed & & \\
\hline 2. Type & R & & \\
\hline 3. Type Extension & D & & \\
\hline 4. Program type & Allowed & & \\
\hline 5. Modifiers Implicit length: (length modifier not present) & R-type: 4 bytes RD-type: 8 bytes & & \\
\hline Alignment: (Length modifier not present) & R-type: Fullword RD-type: Doubleword & & \\
\hline Range for length: & R-type: 3 or 4 only RD-type: 3, 4 or 8 (no bit length) & & \\
\hline Range for scale: & Not allowed & & \\
\hline Range for exponent: & Not allowed & & \\
\hline 6. Nominal value Represented by: & An ordinary symbol & DC R(PSECT1) & \\
\hline Enclosed by: & Parentheses & & \\
\hline Exponent allowed: & No & & \\
\hline Number of values per operand: & Multiple & & \\
\hline Padding: & With zeros at left & & \\
\hline Truncation of assembled value: & Not applicable & & \\
\hline
\end{tabular}

Hexadecimal Floating-Point Constants-E, EH, D, DH, L, LH, LQ Floating-point constants let you introduce data that is in the form suitable for the operations of the floating-point feature instructions. These constants have the following advantages over fixed-point constants:
- You do not have to consider the fractional portion of a value you specify, nor worry about the position of the decimal point when algebraic operations are to be done.
- You can specify both much larger and much smaller values.
- You retain greater processing precision; that is, your values are carried in more significant figures.

The nominal value can be a signed (see 1 in Figure 51 on page 162 integer, fraction, or mixed number (see 2 Figure 51) followed by a signed exponent (see

3 in Figure 51). If a sign is not specified for either the number or exponent, a plus sign is assumed. If you specify the ' H ' type extension you can also specify a rounding mode that is used when the nominal value is converted from decimal to its hexadecimal form. The syntax for nominal values (including the binary floating-point constants) is shown in Figure 53 on page 168. The valid rounding mode values are:

1 Round by adding one in the first lost bit position
4 Unbiased round to nearest, with tie-breaking rule
5 Round towards zero (that is, truncate)
6 Round up towards the maximum positive value
7 Round down towards the minimum negative value
Figure 50. Rounding Mode Values
See 4 in Figure 51.
The exponent must lie within the permissible range. If an exponent modifier is also specified, the algebraic sum of the exponent and the exponent modifier must lie within the permissible range.

Figure 51 (Page 1 of 3). Hexadecimal Floating-Point Constants
\begin{tabular}{lll}
\hline Subfield & Value & Example \\
\hline 1. Duplication factor & Allowed & \\
\hline 2. Type & E, D, and L \\
\hline 3. Type Extension & Omitted or H or Q \\
\hline 4. Program type & Allowed \\
\hline 5. Modifiers & \\
Implicit length: & E-type: 4 bytes \\
\begin{tabular}{l} 
(length modifier \\
not present)
\end{tabular} & D-type: 8 bytes \\
\hline Alignment: & L-type: 16 bytes \\
(Length modifier & E-type: Fullword \\
not present) & D-type: Doubleword \\
& L-type: Doubleword \\
\hline
\end{tabular}

Figure 51 (Page 2 of 3). Hexadecimal Floating-Point Constants
\begin{tabular}{|c|c|c|}
\hline Subfield & Value & Example \\
\hline Range for length: & \begin{tabular}{l}
E-type: \\
1 to 8 (byte length) \\
.1 to .64 (bit length) \\
EH-type: \\
.12 to 64 (bit length) \\
D-type: \\
1 to 8 (byte length) \\
.1 to .64 (bit length) \\
DH-type: \\
.12 to 64 (bit length) \\
L-type: \\
1 to 16 (byte length) \\
.1 to 128 (bit length) \\
LH-type: \\
.12 to .128 (bit length) \\
LQ-type: \\
.12 to .128 (bit length)
\end{tabular} & \\
\hline Range for scale: & E-type: 0 to 14 D-type: 0 to 14 L-type: 0 to 28 & \\
\hline Range for exponent: & -85 to +75 & \\
\hline 6. Nominal value & & \\
\hline Represented by: & Decimal digits & \begin{tabular}{l}
E-type: \\
DC E'+525' \(\mathbf{1}\) \\
DC E'5.25' \(\mathbf{2}\) \\
D-type: \\
DC D'-525' 1 \\
DC D'+.001' 2 \\
L-type: \\
DC L'525' \\
DC L'3.414' 2
\end{tabular} \\
\hline Enclosed by: & Single quotation marks & \\
\hline Exponent allowed: & Yes & \begin{tabular}{l}
E-type: \\
DC E'1E+60' 3 \\
D-type: \\
DC D'-2.5E10' 3 \\
L-type: \\
DC L'3.712E-3' 3
\end{tabular} \\
\hline Rounding mode allowed if type extension specified: & Yes (see Figure 50 for values) & \begin{tabular}{l}
E-type: \\
DC EH'1E+60R1' 4 \\
D-type: \\
DC DH'-2.5E10R4' 4 \\
L-type: \\
DC LH'3.712E-3R5'
\end{tabular} \\
\hline
\end{tabular}

Figure 51 (Page 3 of 3). Hexadecimal Floating-Point Constants
\begin{tabular}{lll}
\hline Subfield & Value & Example \\
\hline \begin{tabular}{l} 
Number of values \\
per operand:
\end{tabular} & Multiple & \begin{tabular}{l} 
Correct fraction is \\
extended to the right and \\
rounded
\end{tabular} \\
\hline Padding: & Only if rounding mode 5; rounded otherwise. \\
\hline \begin{tabular}{l} 
Truncation of \\
assembled value:
\end{tabular} & \\
\hline
\end{tabular}

The format of the constant is shown in Figure 52 on page 165 .
The value of the constant is represented by two parts:
- An exponent portion (see 1 in Figure 52 on page 165), followed by
- A fractional portion (see \(\mathbf{2}\) in Figure 52)

A sign bit (see 3 in Figure 52) indicates whether a positive or negative number has been specified. The number specified must first be converted into a hexadecimal fraction before it can be assembled into the correct internal format. The quantity expressed is the product of the fraction (see 4 in Figure 52) and the number 16 raised to a power (see 5 in Figure 52). Figure 52 shows the external format of the three types of floating-point constants.

Here is the range of values that can be assembled into hexadecimal floating-point constants:
\begin{tabular}{ll}
\hline \begin{tabular}{l} 
Type of \\
Constant
\end{tabular} & \begin{tabular}{l} 
Range of Magnitude (M) of Values \\
(Positive and Negative)
\end{tabular} \\
\hline\(E\) & \(16-65 \leq M \leq(1-16-6) \times 1663\) \\
\hline\(D\) & \(16-65 \leq M \leq(1-16-14) \times 1663\) \\
\hline\(L\) & \(16-65 \leq M \leq(1-16-28) \times 1663\) \\
\hline\(E, D, L\) & \(5.4 \times 10-79 \leq M \leq 7.2 \times 1075\) (approximate) \\
\hline
\end{tabular}

If the value specified for a particular constant does not lie within these ranges, the assembled value then depends on these factors:
- With type extension H , overflows assemble to the largest magnitude for the specified type, underflows will denormalize the value or return zero, depending on the value and rounding mode.
- Without type extension H , certain combinations of exponents (modifier and nominal value) may produce invalid results (message ASMA071E), and if the exponent is too large it is ignored and the nominal value of the constant preceding the exponent is assembled instead.

where \(a, b, c \ldots\) are hexadecimal digits, and \(E\) is an exponent that has a positive or negative value indicated by the characteristic

Figure 52. Hexadecimal Floating-Point External Formats
Representation of Hexadecimal Floating Point: The assembler assembles a floating-point constant into its binary representation as follows: The specified number, multiplied by any exponents, is converted to the required two-part format. The value is translated into:
- A fractional portion represented by hexadecimal digits and the sign indicator. The fraction is then entered into the leftmost part of the fraction field of the constant (after rounding).
- An exponent portion represented by the excess-64 binary notation, which is then entered into the characteristic field of the constant.

The excess- 64 binary notation is obtained by adding +64 to the value of the exponent (which lies between -64 and +63 ) to yield the characteristic (which lies between 0 and 127).

\section*{Notes:}
1. The L-type floating-point constant resembles two contiguous D-type constants. The sign of the second doubleword is assumed to be the same as the sign of the first.
The characteristic for the second doubleword is equal to the characteristic for the first minus 14 (the number of hexadecimal digits in the fractional portion of the first doubleword). No indication is given if the characteristic of the second doubleword is zero.
The L-type and LH-type floating-point constants are double-word aligned. The LQ-type is quad-word aligned. A DC OLQ forces the alignment to a quad-word boundary.
2. If scaling has been specified, hexadecimal zeros are added to the left of the normalized fraction (causing it to become unnormalized), and the exponent in the characteristic field is adjusted accordingly. (For further details on scaling, see "Subfield 5: Modifier" on page 136)
3. The fraction is rounded according to the implied or explicit length of the constant. The resulting number does not differ from the exact value specified by more than one in the last place.

Note: You can control rounding by using the ' H ' type extension and specifying the rounding mode.
4. Negative fractions are carried in true representation, not in the two's-complement form.
5. Duplication is applied after the constant has been assembled.
6. An implied length of 4 bytes is assumed for a short (E) constant and 8 bytes for a long (D) constant. An implied length of 16 bytes is assumed for an extended (L) constant. The constant is aligned at the correct word (E) or doubleword (D and L ) boundary if a length is not specified. However, any length up to and including 8 bytes ( E and D ) or 16 bytes (L) can be specified by a length modifier. In this case, no boundary alignment occurs.
7. Signed zero values are correctly generated for type extensions H and B . Without a type extension, zero values of either sign are assembled with positive sign.

Any of the following statements can be used to specify 46.415 as a positive, fullword, floating-point constant; the last is a machine instruction statement with a literal operand. Note that each of the last two constants contains an exponent modifier.
\begin{tabular}{|c|c|}
\hline DC & E'46.415' \\
\hline DC & E'46415E-3' \\
\hline DC & \(\mathrm{E}^{\prime}+464.15 \mathrm{E}-1^{\prime}\) \\
\hline DC & E'+.46415E+2' \\
\hline DC & EE2'.46415' \\
\hline AE & 6, =EE2 ' \(.46415{ }^{\text {' }}\) \\
\hline
\end{tabular}

The following would generate 3 doubleword floating-point constants.

\title{
DC Instruction-Binary Floating-Point Constants
}

FLOAT DC DE+4'+46,-3.729,+473'

\section*{Binary Floating-Point Constants-EB, DB, LB}

Binary floating-point numbers may be represented in any of three formats: short, long or extended. The short format is 4 bytes with a sign of one bit, an exponent of 8 bits and a fraction of 23 bits. The long format is 8 bytes with a sign of one bit, an exponent of 11 bits and a fraction of 52 bits. The extended format is 16 bytes with a sign of one bit, an exponent of 15 bits and a fraction of 112 bits.

There are five classes of binary floating-point data, including numeric and related nonnumeric entities. Each data item consists of a sign, an exponent and a significand. The exponent is biased such that all exponents are nonnegative unsigned numbers, and the minimum biased exponent is zero. The significand consists of an explicit fraction and an implicit unit bit to the left of the binary point. The sign bit is zero for plus and one for minus values.

All finite nonzero numbers within the range permitted by a given format are normalized and have a unique representation. There are no unnormalized numbers, which might allow multiple representations for the same value, and there are no unnormalized arithmetic operations. Tiny numbers of a magnitude below the minimum normalized number in a given format are represented as denormalized numbers, because they imply a leading zero bit, but those values are also represented uniquely.

The classes are:
1. Zeros have a biased exponent of zero, a zero fraction and a sign. The implied unit bit is zero.
2. Denormalized numbers have a biased exponent of zero and a nonzero fraction. The implied unit bit is zero.

The smallest denormalized numbers have approximate magnitudes 1.4 10**-45 (short format), 4.94 10**-324 (long format) and \(6.510^{* *}-4966\) (extended format).
3. Normalized numbers have a biased exponent greater than zero but less than all ones. The implied unit bit is one and the fraction may have any value. The largest normalized numbers have approximate magnitudes 3.4 10**38 (short format), \(1.810 * * 308\) (long format), and \(1.210 * 4932\) (extended format). The smallest normalized numbers have approximate magnitudes \(1.1810^{* *}\)-38 (short format), 2.23 10**-308 (long format), and 3.4 10**-4392 (extended format).
4. An infinity is represented by a biased exponent of all ones and a zero fraction.
5. A NaN (Not-a-Number) entity is represented by a biased exponent of all ones and a nonzero fraction. NaNs are produced in place of a numeric result after an invalid operation when there is no interruption. NaNs may also be used by the program to flag special operands, such as the contents of an uninitialized storage area. There are two types of NaNs , signaling and quiet. A signaling \(\mathrm{NaN}(\mathrm{SNaN})\) is distinguished from the corresponding quiet NaN (QNaN) by the leftmost fraction bit: zero for the SNaN and one for QNaN. A special QNaN is supplied as the default result for an invalid-operation condition; it has a plus sign and a leftmost fraction bit of one, with the remaining fraction bits being set to zeros. Normally, QNaNs are just propagated during computations, so that they remain visible at the end. An SNaN operand causes an invalid operation exception.

To accommodate the definition of both hexadecimal and binary floating-point constants the syntax for coding a DC instruction is:

nominal_value (no type extension):

nominal_value (type extension \(B\) ):


\section*{nominal_value (type extension H ):}


Figure 53. DC Instruction Syntax
dup_factor
causes the constant to be generated the number of times indicated by the factor.
type
indicates that the constant is either short, long or extended floating point.
type extension
the type of conversion required to assemble the constant. Valid values are:
null Hexadecimal floating-point constant which is converted using the conversion logic of rounding mode 1 and slightly less precise algorithms

B Binary floating-point constant which is converted allowing all rounding modes

H Hexadecimal floating-point constant which is converted allowing all rounding modes
program_type
assign a programmer determined 32-bit value to the symbol naming the DC instruction, if a symbol was present.
modifier
describes the length, the scaling and the exponent of the nominal_value. The minimum length of the ' H ' hexadecimal constant is 12 bits. The minimum length in bits of the binary constant is:

9 Short floating-point constant

\section*{16 Extended floating-point constant}

This minimum length allows for the sign, exponent, the implied unit bit which is considered to be one for normalized numbers and zero for zeros and denormalized numbers.

The exponent modifier can be in the range from \(-2^{31}\) to \(2^{31}-1\) if either B or H is specified as a type extension

\section*{nominal_value}
defines the value of the constant and can include the integer, fraction or mixed number followed by an optional signed exponent and an optional explicit rounding mode.

The assembler imposes no limits on the exponent values that may be specified. The BFP architecture limits the actual values that can be represented; a warning message is issued whenever a specified value can not be represented exactly.

The rounding mode identifies the rounding required when defining a floating-point constant. The valid values are those displayed in Figure 50 on page 162

Note: As binary floating-point does not support scaling, the scale modifier is ignored and a warning message issued if the scaling modifier is specified when defining a binary floating-point constant. The H type extension causes HLASM to uses a different conversion algorithm for hexadecimal floating-point data. The results are correctly rounded for all values. Without the H type extension, some rare values could be in error by 1 unit in the last place (ulp).

\section*{Conversion to Binary Floating-Point}

For decimal to binary floating-point conversion, the assembler conforms to ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, dated August 12, 1985, with the following differences: exception status flags are not provided and traps are not supported.

Conversion of values within the represented range is correctly rounded.
Conversion of values outside the represented range is as follows. If the resultant value before rounding is larger in magnitude than MAX (the maximum allowed value) as represented in the specified length, then, depending on the rounding mode, either MAX or infinity is generated, along with a warning message. If the resultant nonzero value is less than Dmin (the minimum allowed value) as represented in the specified length, then, depending on the rounding mode, either Dmin or zero is generated, along with a warning message.

Floating-Point Special Values
For special values, the syntax of the DC statement is:

dup_factor
causes the constant to be generated the number of times indicated by the factor.
type indicates that the constant is either short, long or extended floating point.
type extension
the type of conversion required to assemble the constant.
program_type
assign a programmer determined 32-bit value to the symbol naming the DC instruction, if a symbol was present.
length_modifier
describes the length in bytes or bits into which the constant is to be assembled. For NANs and INF the minimum length in bits of the constant is:

11 Short floating-point constant
14 Long floating-point constant
18 Extended floating-point constant
This minimum length allows for the sign, exponent and two fraction bits.

\section*{nominal_value}
defines the special value to be generated.

\section*{Notes:}
1. The nominal value can be in mixed case.
2. SNAN assembles with an exponent of ones and 01 in the high order fraction bits with the remainder of the fraction containing zeros.
3. QNAN assembles with an exponent of ones and 11 in the high order fraction bits with the remainder of the fraction containing zeros.
4. NAN assembles with an exponent of one and 10 in the high order fraction bits with the remainder of the fraction containing zeros.
5. MIN assembles as a normalized minimum value, that is an exponent of one and a fraction of zeros for binary constants, and a fraction with a leading hexadecimal digit 1 followed by zeros for hexadecimal constants.
6. DMIN assembles as a denormalized minimum value with an exponent of zeros and a fraction of all zeros except for a low order bit of one.
7. INF assembles with an exponent of ones and a fraction of zeros.
8. MAX assembles with a fraction of all ones and an exponent of all ones for hexadecimal constants, and an exponent of all ones except for the low bit for binary constants.

\section*{Literal Constants}

Literal constants let you define and refer to data directly in machine instruction operands. You do not need to define a constant separately in another part of your source module. The differences between a literal, a data constant, and a self-defining term are described in "Literals" on page 40.

A literal constant is specified in the same way as the operand of a DC instruction. The general rules for the operand subfields of a DC instruction also apply to the subfield of a literal constant. Moreover, the rules that apply to the individual types of constants apply to literal constants as well.

However, literal constants differ from DC operands in the following ways:
- Literals must be preceded by an equal sign.
- Multiple operands are not allowed.
- The duplication factor must not be zero.
- Symbols used in the duplication factor or length modifier must be previously defined. Scale and Exponent modifiers do not need pre-definition.
- If an address-type literal constant specifies a duplication factor greater than one and a nominal value containing the location counter reference, the value of the location counter reference is not incremented, but remains the same for each duplication.
- The assembler groups literals together by size. If you use a literal constant, the alignment of the constant can be different than might be the case for an explicit constant. See "Literal Pool" on page 44

\section*{DROP Instruction}

The DROP instruction ends the domain of a USING instruction. This:
- Frees base registers previously assigned by the USING instruction for other programming purposes
- Ensures that the assembler uses the base register you want in a particular coding situation, for example, when two USING ranges overlap or coincide
- If a control section has not been established, DROP will initiate an unnamed (private) control section

sequence_symbol
is a sequence symbol.
base_register
is an absolute expression whose value represents one of the general registers 0 through 15. The expression in base_register indicates a general register, previously specified in the operand of an ordinary USING statement, that is no longer to be used for base addressing.
label
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

The ordinary symbol denoted by label must be a symbol previously used in the name field of a labeled USING statement or a labeled dependent USING statement.

If neither base_register nor label is specified in the operand of a DROP instruction, all active base registers assigned by ordinary, labeled, and labeled dependent USING instructions are dropped.

After a DROP instruction:
- The assembler does not use the register or registers specified in the DROP instruction as base registers. A register made unavailable as a base register by a DROP instruction can be reassigned as a base register by a subsequent USING instruction.
- The label or labels specified in the DROP instruction are no longer available as symbol qualifiers. A label made unavailable as a symbol qualifier by a DROP instruction can be reassigned as a symbol qualifier by a subsequent labeled USING instruction.

The following statements, for example, stop the assembler using registers 7 and 11 as base registers, and the label FIRST as a symbol qualifier:
```

DROP
DROP
7,11
FIRST

```

Labeled USING: You cannot end the domain of a labeled USING instruction by coding a DROP instruction that specifies the same registers as were specified in the labeled USING instruction. If you want to end the domain of a labeled USING instruction, you must code a DROP instruction with an operand that specifies the label of the labeled USING instruction.

Dependent USING: To end the domain of a dependent USING instruction, you must end the domain of the corresponding ordinary USING instruction. In the following example, the DROP instruction prevents the assembler from using register 12 as a base register. The DROP instruction causes the assembler to end the domain of the ordinary USING instruction and the domains of the two dependent USING instructions. The storage areas represented by INREC and OUTREC are both within the range of the ordinary USING instruction (register 12).
\begin{tabular}{ll} 
USING & \(*, 12\) \\
USING & RECMAP, INREC \\
USING & RECMAP, OUTREC \\
\(\cdot\) & \\
- & \\
DROP & 12 \\
\(\cdot\) & \\
- & \\
DS & CL156 \\
DS &
\end{tabular}

To end the domain of a labeled dependent USING instruction, you can code a DROP instruction with the USING label in the operand. The following example shows this:
\begin{tabular}{lll} 
& USING & \(*, 12\) \\
PRIOR & USING & RECMAP, INREC \\
POST & USING & RECMAP,OUTREC \\
& - & \\
& DROP & \\
& - & \\
& PRIOR, POST \\
INREC & DS & \\
OUTREC & DS & CL156 \\
& & CL156
\end{tabular}

In the above example, the DROP instruction makes the labels PRIOR and POST unavailable as symbol qualifiers.

When a labeled dependent USING domain is dropped, none of any subordinate USING domains are dropped. In the following example the labeled dependent USING BLBL1 is not dropped, even though it is dependent on the USING ALBL2 that is dropped:

\section*{DS Instruction}
\begin{tabular}{|c|c|c|}
\hline & USING & DSECTA, 14 \\
\hline \multirow[t]{3}{*}{ALBL1} & USING & DSECTA, 14 \\
\hline & USING & DSECTB,ALBL1.A \\
\hline & - & \\
\hline ALBL2 & USING & DSECTA,ALBL1.A \\
\hline & & \\
\hline \multirow[t]{3}{*}{BLBL1} & USING & DSECTA,ALBL2. \(\mathrm{A}+4\) \\
\hline & - & \\
\hline & DROP & ALBL2 \\
\hline & - & \\
\hline DSECTA & DSECT & \\
\hline A & DS & A \\
\hline DSECTB & DSECT & \\
\hline B & DS & A \\
\hline
\end{tabular}

A DROP instruction is not needed:
- If the base address is being changed by a new ordinary USING instruction, and the same base register is assigned. However, the new base address must be loaded into the base register by an appropriate sequence of instructions.
- If the base address is being changed by a new labeled USING instruction or a new labeled dependent USING instruction, and the same USING label is assigned; however, the correct base address must be loaded into the base register specified in the USING instruction by an appropriate sequence of instructions.
- At the end of a source module

\section*{DS Instruction}

The DS instruction:
- Reserves areas of storage
- Provides labels for these areas
- Uses these areas by referring to the symbols defined as labels
- If a control section has not previously been established, DS will initiate an unnamed (private) control section


\section*{symbol}
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If symbol denotes an ordinary symbol, the ordinary symbol represents the address of the first byte of the storage area reserved. If several operands are
specified, the first storage area defined is addressable by the ordinary symbol. The other storage areas can be reached by relative addressing.
operand
is an operand of six subfields. The first five subfields describe the attributes of the symbol. The sixth subfield provides the nominal values that determine the implicit lengths; however no constants are generated.

A DS operand has this format:
\(\square\)

The format of the DS operand is identical to that of the DC operand; exactly the same subfields are used and are written in exactly the same sequence as they are in the DC operand. For more information about the subfields of the DC instruction, see "DC Instruction" on page 126

Unlike the DC instruction, the DS instruction causes no data to be assembled. Therefore, you do not have to specify the nominal value (sixth subfield) of a DS instruction operand. The DS instruction is the best way of symbolically defining storage for work areas, input/output buffers, etc.

Although the formats are identical, there are two differences in the specification of subfields. They are:
- The nominal value subfield is optional in a DS operand, but it is mandatory in a DC operand. If a nominal value is specified in a DS operand, it must be valid.
- The maximum length that can be specified for the character ( C ) and hexadecimal (X) type areas is 65,535 bytes rather than 256 bytes for the same DC operands. The maximum length for the graphic ( \(G\) ) type is 65,534 bytes.

If symbol denotes an ordinary symbol, the ordinary symbol, as with the DC instruction:
- Has an address value of the first byte of the area reserved, after any boundary alignment is done
- Has a length attribute value, depending on the implicit or explicit length of the type of area reserved

If the DS instruction is specified with more than one operand or more than one nominal value in the operand, the label addresses the area reserved for the field that corresponds to the first nominal value of the first operand. The length attribute value is equal to the length explicitly specified or implicit in the first operand.

Bytes Skipped for Alignment: Unlike the DC instruction, bytes skipped for alignment are not set to zero. Also, nothing is assembled into the storage area reserved by a DS instruction. No assumption should be made as to the contents of the skipped bytes or the reserved area.

The size of a storage area that can be reserved by a DS instruction is limited only by the size of virtual storage or by the maximum value of the location counter, whichever is smaller.

\section*{How to Use the DS Instruction}

Use the DS instruction to:
- Reserve storage
- Force alignment of the location counter so that the data that follows is on a particular storage boundary
- Name fields in a storage area.

To Reserve Storage: If you want to take advantage of automatic boundary alignment (if the ALIGN option is specified) and implicit length calculation, you should not supply a length modifier in your operand specifications. Instead, specify a type subfield that corresponds to the type of area you need for your instructions.

Using a length modifier can give you the advantage of explicitly specifying the length attribute value assigned to the label naming the area reserved. However, your areas are not aligned automatically according to their type. If you omit the nominal value in the operand, you should use a length modifier for the binary \((B)\), character (C), graphic (G), hexadecimal (X), and decimal ( \(P\) and \(Z\) ) type areas; otherwise, their labels are given a length attribute value of 1 (2 for \(G\) and \(C U\) type).

When you need to reserve large areas, you can use a duplication factor. However, in this case, you can only refer to the first area by its label. You can also use the character \((C)\) and hexadecimal \((X)\) field types to specify large areas using the length modifier. Duplication has no effect on implicit length.

Although the nominal value is optional for a DS instruction, you can put it to good use by letting the assembler compute the length for areas of the \(B, C, G, X\), and decimal ( P or Z ) type areas. You achieve this by specifying the general format of the nominal value that is placed in the area at execution time.

If a nominal value and no length modifier are specified for a Unicode character string, the length of the storage reserved is derived by multiplying by two the number of characters specified in the nominal value (after pairing).

To Force Alignment: Use the DS instruction to align the instruction or data that follows, on a specific boundary. You can align the location counter to a doubleword, a fullword, or a halfword boundary by using the correct constant type (for example, D, F, or H) and a duplication factor of zero. No space is reserved for such an instruction, yet the data that follows is aligned on the correct boundary. For example, the following statements set the location counter to the next doubleword boundary and reserve storage space for a 128-byte field (whose first byte is on a doubleword boundary).
\begin{tabular}{lll} 
& DS & OD \\
AREA & DS & CL128
\end{tabular}

Alignment is forced whether or not the ALIGN assembler option is set.
To Name Fields within an Area: Using a duplication factor of zero in a DS instruction also provides a label for an area of storage without actually reserving the area. Use DS or DC instructions to reserve storage for, and assign labels to, fields
within the area. These fields can then be addressed symbolically. (Another way of accomplishing this is described in "DSECT Instruction" on page 178) The whole area is addressable by its label. In addition, the symbolic label has the length attribute value of the whole area. Within the area, each field is addressable by its label.

For example, assume that 80-character records are to be read into an area for processing and that each record has the following format:
\begin{tabular}{ll} 
Positions 5-10 & Payroll Number \\
Positions 11-30 & Employee Name \\
Positions 31-36 & Date \\
Positions 47-54 & Gross Wages \\
Positions 55-62 & Withholding Tax
\end{tabular}

The following example shows how DS instructions might be used to assign a name to the record area, then define the fields of the area and allocate storage for them. The first statement names the whole area by defining the symbol RDAREA; this statement gives RDAREA a length attribute of 80 bytes, but does not reserve any storage. Similarly, the fifth statement names a 6-byte area by defining the symbol DATE; the three subsequent statements actually define the fields of DATE and allocate storage for them. The second, ninth, and last statements are used for spacing purposes and, therefore, are not named.
\begin{tabular}{lll} 
RDAREA & DS & OCL80 \\
& DS & CL4 \\
PAYNO & DS & CL6 \\
NAME & DS & CL20 \\
DATE & DS & 0CL6 \\
DAY & DS & CL2 \\
MONTH & DS & CL2 \\
YEAR & DS & CL2 \\
& DS & CL10 \\
GROSS & DS & CL8 \\
FEDTAX & DS & CL8 \\
& DS & CL18
\end{tabular}

Additional examples of DS statements are shown below:
\begin{tabular}{llll} 
ONE & DS & CL80 & One 80-byte field, length attribute of 80 \\
TWO & DS & 80 C & \begin{tabular}{l} 
80 1-byte fields, length attribute of 1
\end{tabular} \\
THREE & DS & 6 F & 6 fullwords, length attribute of 4
\end{tabular}

To define four 10-byte fields and one 100-byte field, the respective DS statements might be as follows:
\begin{tabular}{lll} 
FIELD & DS & 4CL10 \\
AREA & DS & CL100
\end{tabular}

Although FIELD might have been specified as one 40-byte field, the preceding definition has the advantage of providing FIELD with a length attribute of 10. This would be pertinent when using FIELD as an SS machine instruction operand.

\section*{DSECT Instruction}

The DSECT instruction identifies the beginning or continuation of a dummy control section. One or more dummy sections can be defined in a source module.


\section*{symbol}
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The DSECT instruction can be used anywhere in a source module after the ICTL instruction.

If symbol denotes an ordinary symbol, the ordinary symbol identifies the dummy section. If several DSECT instructions within a source module have the same symbol in the name field, the first occurrence initiates the dummy section and the rest indicate the continuation of the dummy section. The ordinary symbol denoted by symbol represents the address of the first byte in the dummy section, and has a length attribute value of 1.

If symbol is not specified, or if name is a sequence symbol, the DSECT instruction initiates or indicates the continuation of the unnamed control section.

The location counter for a dummy section is always set to an initial value of 0 . However, when an interrupted dummy control section is continued using the DSECT instruction, the location counter last specified in that control section is continued.

The source statements that follow a DSECT instruction belong to the dummy section identified by that DSECT instruction.

\section*{Notes:}
1. The assembler language statements that appear in a dummy section are not assembled into object code.
2. When establishing the addressability of a dummy section, the symbol in the name field of the DSECT instruction, or any symbol defined in the dummy section can be specified in a USING instruction.
3. A symbol defined in a dummy section can be specified in an address constant only if the symbol is paired with another symbol from the same dummy section, and if the symbols have opposite signs.

To effect references to the storage area defined by a dummy section, do the following:
- Provide either:
- An ordinary or labeled USING statement that specifies both a general register that the assembler can use as a base register for the dummy section, and a value from the dummy section that the assembler may assume the register contains, or
- A dependent or labeled dependent USING statement that specifies a supporting base address (for which there is a corresponding ordinary USING statement) that lets the assembler determine a base register and displacement for the dummy section, and a value from the dummy section that the assembler may assume is the same as the supporting base address
- Ensure that the base register is loaded with either:
- The actual address of the storage area if an ordinary USING statement or a labeled USING statement was specified, or
- The base address specified in the corresponding ordinary USING statement if a dependent or labeled dependent USING statement was specified.

The values assigned to symbols defined in a dummy section are relative to the initial statement of the section. Thus, all machine instructions that refer to names defined in the dummy section refer, at execution time, to storage locations relative to the address loaded into the register.

Figure 54 shows an example of how to code the DSECT instruction. The sample code is referred to as "Assembly-2."

Assume that two independent assemblies (Assembly-1 and Assembly-2) have been loaded and are to be run as a single overall program. Assembly-1 is a routine that
1. Places a record in an area of storage
2. Places the address of the storage area in general register 3
3. Branches to Assembly-2 to process the record

The storage area from Assembly-1 is identified in Assembly-2 by the dummy control section (DSECT) named INAREA. Parts of the storage area that you want to work with are named INCODE, OUTPUTA, and OUTPUTB. The statement USING INAREA, 3 assigns general register 3 as the base register for the INAREA DSECT. General register 3 contains the address of the storage area. Because the symbols in the DSECT are defined relative to the beginning of the DSECT, the address values they represent are, at the time of program execution, the actual storage locations of the storage area that general register 3 addresses.
\begin{tabular}{lll}
\hline ASEMBLY2 & CSECT & \\
& USING & \(*, 15\) \\
& USING & INAREA,3 \\
& CLI & INCODE, ' 'A' \\
& BE & ATYPE \\
& MVC & OUTPUTA, DATA_B \\
& MVC & OUTPUTB, DATA_A \\
& B & FINISH \\
ATYPE & DS & 0H \\
& MVC & OUTPUTA, DATA_A \\
& MVC & OUTPUTB, DATA_B \\
FINISH & BR & 14 \\
DATA-A & DC & CL8'ADATA' \\
DATA-B & DC & CL8'BDATA' \\
INAREA & DSECT & \\
INCODE & DS & CL1 \\
OUTPUTA & DS & CL8 \\
OUTPUTB & DS & CL8 \\
& END & \\
& & \\
\hline
\end{tabular}

Figure 54. Sample Code Using the DSECT Instruction (Assembly-2)

\section*{DXD Instruction}

The DXD instruction identifies and defines an external dummy section.

symbol
is an external symbol which is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
duplication_factor
is the duplication factor subfield equivalent to the duplication factor subfield of the DS instruction.
type
is the type subfield equivalent to the type subfield of the DS instruction.
type_extension
is the type extension subfield equivalent to the type extension subfield of the DS instruction.
modifiers
is the modifiers subfield equivalent to the modifiers subfield of the DS instruction.

\section*{EJECT Instruction}

\section*{nominal_value}
is the nominal-value subfield equivalent to the nominal-value subfield of the DS instruction. The nominal value is optional. If specified, it is not generated.

The DXD instruction can be used anywhere in a source module, after the ICTL instruction.

In order to reference the storage defined by the external dummy section, the ordinary symbol denoted by symbol must appear in the operand of a Q-type constant. This symbol represents the address of the first byte of the external dummy section defined, and has a length attribute value of 1 .

The subfields in the operand field (duplication factor, type, type extension, modifier, and nominal value) are specified in the same way as in a DS instruction. The assembler computes the amount of storage and the alignment required for an external dummy section from the area specified in the operand field. For more information about how to specify the subfields, see "DS Instruction" on page 174.

For example:
\begin{tabular}{llll} 
A & DXD & CL20 & 20 bytes, byte alignment \\
B & DXD & \(3 F, X L 4\) & 20 bytes, fullword alignment \\
C & DXD & LQ & 16 bytes, quadword alignment
\end{tabular}

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

\section*{Notes:}
1. The DSECT instruction also defines an external dummy section, but only if the symbol in the name field appears in a Q-type offset constant in the same source module. Otherwise, a DSECT instruction defines a dummy section.
2. If two or more external dummy sections for different source modules have the same name, the linker uses the most restrictive alignment, and the largest section to compute the total length.

\section*{EJECT Instruction}

The EJECT instruction stops the printing of the assembler listing on the current page, and continues the printing on the next page.

sequence_symbol
is a sequence symbol.
The EJECT instruction causes the next line of the assembler listing to be printed at the top of a new page. If the line before the EJECT statement appears at the bottom of a page, the EJECT statement has no effect.

An EJECT instruction immediately following another EJECT instruction is ignored. A TITLE instruction immediately following an EJECT instruction causes the title to
change but no additional page eject is performed. (The TITLE instruction normally forces a page eject.)

The EJECT instruction statement itself is not printed in the listing.

\section*{END Instruction}

Use the END instruction to end the assembly of a program. You can also supply an address in the operand field to which control can be transferred after the program is loaded. The END instruction must always be the last statement in the source program.

sequence_symbol
is a sequence symbol.

\section*{expression}
specifies the point to which control can be transferred when loading of the object program completes. If the GOFF option is in effect this parameter is ignored. This point is usually the address of the first executable instruction in the program, as shown in the following sequence:
NAME
CSECT
AREA DS 50F
BEGIN BALR USING

2,0
.
-
END BEGIN
If specified, expression may be generated by substitution into variable symbols.
- It must be a simply relocatable expression representing an address in the source module delimited by the END instruction, or
- If it contains an external symbol, the external symbol must be the only term in the expression, or the remaining terms in the expression must reduce to zero.
- It must not be a literal.

\section*{language}
a marker for use by language translators that produce assembly code. The operand has three sub-operands. The values in this operand are copied into the END record in the object deck.
The syntax of this operand is
(char10,char4, char5)
where all three sub-operands, and the commas and parentheses are required.
char10 is a one to ten character code. It is intended to be a language translator identifier. char4 must be exactly four characters long. It is intended to be a version and release code. char5 must be exactly five characters long, and

\section*{ENTRY Instruction}
should be a date in the format "YYDDD." It is intended to be the compile date. For example:

END ENTRYPT, (MYCOMPILER,0101,00273)

\section*{Notes:}
1. If the END instruction is omitted, one is generated by the assembler, and message ASMA140W END record missing is issued.
2. Refer to the note on page 341 about lookahead processing, and the effect it has on generated END statements.
3. If the END statement is not the last statement in the input stream, and the BATCH option (see HLASM Programmer's Guide has been specified, the assembler will initiate assembly of a new source module when the current assembly is completed.

\section*{ENTRY Instruction}

The ENTRY instruction identifies symbols defined in one source module so that they can be referred to in another source module. These symbols are entry symbols.

sequence_symbol
is a sequence symbol.
entry_point
is a relocatable symbol that:
- Is a valid symbol
- Is defined in an executable control section
- Is not defined in a dummy control section, a common control section, or an external control section

Up to 65535 individual control sections, external symbols, and external dummy sections can be defined in a source module. However, the practical maximum number depends on the amount of table storage available to the program that links the object module.

The assembler lists each entry symbol of a source module in an external symbol dictionary, along with entries for external symbols, common control sections, parts, and external control sections.

A symbol used as the name entry of a START or CSECT instruction is also automatically considered an entry symbol, and does not have to be identified by an ENTRY instruction.

The length attribute value of entry symbols is the same as the length attribute value of the symbol at its point of definition.

\section*{EQU Instruction}

The EQU instruction assigns absolute or relocatable values to symbols. Use it to:
- Assign single absolute values to symbols.
- Assign the values of previously defined symbols or expressions to new symbols, thus letting you use different mnemonics for different purposes.
- Compute expressions whose values are unknown at coding time or difficult to calculate. The value of the expressions is then assigned to a symbol.
- Assign length and type attributes to symbols, either implicitly or explicitly.
- Assign program type and assembler type values to symbols.

EQU also assigns attributes. It takes the value, relocation, and length attributes of the operand and assigns them to the name field symbol, and sets the integer and scale attributes to zero. The type attributes of an absolute expression is always ' \(U\) ', and its length attribute is always 1 (unless the second and third operands are specified.

When there is a symbol naming a complex relocatable expression, or a complex relocatable expression is eventually "reduced" to an absolute or simply relocatable expression, the first symbol is used for attribute assignment.

The program type is always null, and the assembler type is always null, except when the appropriate operand is specified.


Note:
\({ }^{1}\) Use commas as placeholders when there is an expression following

\section*{symbol}
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
expression_1
represents a value and attributes that the assembler assigns to the symbol in the name field. expression_1 may have any value allowed for an assembly expression: absolute (including negative), relocatable, or complexly relocatable. The assembler carries this value as a signed 4-byte (32-bit) number; all four bytes are printed in the program listings opposite the symbol. Implicitly, the relocation and length attributes are also assigned for certain types of expressions.

Any symbols used in expression_1 need not be previously defined. However, if any symbol is not previously defined, the value of expression_1 is not assigned to the symbol in the name field until assembly time and therefore may not be
used during conditional assembly (see "Using Conditional Assembly Values" on page 187.
If expression_1 is a complexly relocatable expression, the whole expression, rather than its value, is assigned to the symbol. During the evaluation of any expression that includes a complexly relocatable symbol, that symbol is replaced by its own defining expression. Consider the following example, in which A1 and A2 are defined in one control section, and B1 and B2 in another:
\begin{tabular}{lll}
\(X\) & EQU & \(A 1+B 1\) \\
\(Y\) & EQU & \(X-A 2-B 2\)
\end{tabular}

The first EQU statement assigns a complexly relocatable expression (A1+B1) to X . During the evaluation of the expression in the second EQU statement, X is replaced by its defining relocatable expression ( \(\mathrm{A} 1+\mathrm{B} 1\) ), and the assembler evaluates the resulting expression (A1+B1-A2-B2) and assigns an absolute value to \(Y\), because the relocatable terms in the expression are paired. Note that the expression may not contain literals.

\section*{expression_2}
represents a value that the assembler assigns as a length attribute value to the symbol in the name field. It is optional, but, if specified, must be an absolute value in the range 0 to 65,535 . This value overrides the normal length attribute value implicitly assigned from expression_1.
All symbols appearing in expression_2 must have been previously defined, and all expressions in expression_2 must be evaluatable when the EQU statement is processed. For example, the second operand in the statements defining the symbol \(X\) cannot be evaluated when the last statement has been processed, because the value of the symbol \(X\) is unknown until the symbol \(A\) has been defined.
\begin{tabular}{lllll} 
Z & DS & XL(L'A) & Z & DS XL(A) \\
Y & DS & XL7 & Y & DS XL7 \\
X & EQU & Z,*-Z & X & EQU Z,*-Z \\
A & DS & XL5 & A & EQU 5
\end{tabular}

If expression_2 is omitted, the assembler assigns a length attribute value to the symbol in the name field according to the length attribute value of the leftmost (or only) term of expression_1, as follows:
1. If the leftmost term of expression_1 is a location counter reference (*), a self-defining term, or a symbol length attribute value reference, the length attribute is 1 . This also applies if the leftmost term is a symbol that is equated to any of these values.
2. If the leftmost term of expression_1 is a symbol that is used in the name field of a DC or DS instruction, the length attribute value is equal to the implicit or explicit length of the first (or only) constant specified in the DC or DS operand field.
3. If the leftmost term is a symbol that is used in the name field of a machine instruction, the length attribute value is equal to the length of the assembled instruction.
4. Symbols that name assembler instructions, except the DC, DS, CCW, CCW0, and CCW1 instructions, have a length attribute value of 1. Symbols that name a CCW, CCW0, or CCW1 instruction have a length attribute value of 8 .
5. The length attribute value described in cases 2,3 , and 4 above is the assembly-time value of the attribute.

For more information about the length attribute value, see "Symbol Length Attribute Reference" on page 38 .

For example:
\begin{tabular}{llll}
\(X\) & DS & CL80 & \(X\) has length attribute 80 \\
\(Y\) & EQU & \(X, 40\) & \(Y\) has length attribute 40
\end{tabular}
expression_3
represents a value that the assembler assigns as a type attribute value to the symbol in the name field. It is optional, but, if specified, it must be an absolute value in the range 0 to 255 .
All symbols appearing in expression_3 must have been previously defined, and all expressions in expression_3 must be evaluatable when the EQU statement is processed.
If expression_3 is omitted, the assembler assigns a type attribute value of \(U\) to the symbol, which means the symbol in the name field has an undefined (or unknown or unassigned) type attribute. See the general discussion about data attributes on page 324, and "Type Attribute ( \(T^{\prime}\) )" on page 328
For example:
\begin{tabular}{llll} 
A & DS & D & A has type attribute D \\
B & EQU & A, ,C'X' & B has type attribute X
\end{tabular}
expression_4
represents a value that the assembler assigns as a program type value to the symbol in the name field. It is optional. It can be specified as a decimal, character, hex or binary self-defining term and is stored as a 4-byte (32-bit) number; all four bytes are printed in the program listings opposite the symbol. The value is not used in any way by the assembler, and may be queried by using the SYSATTRP built-in function.

All symbols appearing in expression_4 must have been previously defined, and all expressions in expression_4 must be evaluatable when the EQU statement is processed.

If expression_4 is omitted, the assembler assigns a null to the program type, and querying the value using the SYSATTRP built-in function returns a null value.
expression_5
represents a character value from 1-4 bytes in length, that the assembler assigns as an assembler type value to the symbol in the name field. It is optional. It is stored as a 4-byte string; all four bytes are printed in the program listings opposite the symbol. The value is used by the assembler when type-checking has been activated, and may be queried by using the SYSATTRA built-in function.

Valid values for this expression are:
\begin{tabular}{ll} 
AR & Register - Access \\
CR & Register - Control \\
CR32 & Register - Control 32-bit \\
CR64 & Register - Control 64-bit \\
FPR & Register - Floating-Point \\
GR & Register - General \\
GR32 & Register - General 32-bit \\
GR64 & Register - General 64-bit
\end{tabular}

If expression_5 is omitted, the assembler assigns a null value to the assembler type, and querying the value using the SYSATTRA build-in function returns a null value.

The EQU instruction can be used anywhere in a source module after the ICTL instruction. Note, however, that the EQU instruction will initiate an unnamed control section (private code) if it is specified before the first control section.

\section*{Using Conditional Assembly Values}

The following rules describe when you can use the value, length attribute value, or type attribute value of an equated symbol in conditional assembly statements:
- If you want to use the value of the symbol in conditional assembly statements, then:
- The EQU statement that defines the symbol must be processed by the assembler before the conditional assembly statement that refers to the symbol.
- The symbol in the name field of the EQU statement must be an ordinary symbol.
- Expression_1 must be an absolute expression, and must contain only self-defining terms or previously defined symbols.
- If only expression_1 is specified, the assembler assigns a value of 1 to the length attribute, and a value of \(U\) to the type attribute of the symbol during conditional assembly. You can use these values in conditional assembly statements, although references to the length attribute may be flagged.
If you specify expression_2 or expression_3 and you want to use the explicit attribute value during conditional assembly processing, then:
- The symbol in the name field must be an ordinary symbol.
- The expression must contain only self-defining terms.

\section*{EXITCTL Instruction}

The EXITCTL instruction sets or modifies the contents of the four signed fullword exit-control parameters that the assembler maintains for each type of exit.

sequence_symbol
is a sequence symbol.
exit_type
identifies the type of exit to which this EXITCTL instruction applies. Exit_type must have one of the following values:

SOURCE Sets the exit-control parameters for the user-supplied exit module specified in the INEXIT suboption of the EXIT assembler option.
LIBRARY Sets the exit-control parameters for the user-supplied exit module specified in the LIBEXIT suboption of the EXIT assembler option.
LISTING Sets the exit-control parameters for the user-supplied exit module specified in the PRTEXIT suboption of the EXIT assembler option.
PUNCH Sets the exit-control parameters for the user-supplied exit module specified in the OBJEXIT suboption of the EXIT assembler option when it is called to process the object module records generated when the DECK assembler option is specified.
OBJECT (MVS and CMS)
Sets the exit-control parameters for the user-supplied exit module specified in the OBJEXIT suboption of the EXIT assembler option when it is called to process the object module records generated when the OBJECT or GOFF assembler option is specified.
ADATA Sets the exit-control parameters for the user-supplied exit module specified in the ADEXIT suboption of the EXIT assembler option.
TERM Sets the exit-control parameters for the user-supplied exit module specified in the TRMEXIT suboption of the EXIT assembler option.
control_value
is the value to which the corresponding exit-control parameter should be set. For each exit type, the assembler maintains four exit-control parameters known as EXITCTL_1, EXITCTL_2, EXITCTL_3, and EXITCTL_4. Therefore, up to four values may be specified. Which exit-control parameter is set is determined by the position of the value in the operand of the instruction. You must code a comma in the operand for each omitted value. If specified, control_value must be either:
- A decimal self-defining term with a value in the range \(-2^{31}\) to \(+2^{31}-1\).
- An expression in the form \(* \pm n\), 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 control_value is omitted, the corresponding exit-control parameter retains its current value.

The following example shows how to set the exit-control parameters EXITCTL_1 and EXITCTL_3 for the LISTING exit without affecting the contents of the other exit-control parameters:
EXITCTL LISTING,256,,*+128

See the HLASM Programmer's Guide for information about how EXITCTL values are passed to each type of exit.

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

\section*{EXTRN Instruction}

\section*{EXTRN Instruction}

The EXTRN instruction identifies symbols referred to in a source module but defined in another source module. These symbols are external symbols.

sequence_symbol
is a sequence symbol.

\section*{external_symbol}
is a relocatable symbol that:
- Is a valid symbol
- Is not used as the name entry of a source statement in the source module in which it is defined

Up to 65535 individual control sections, external symbols, and external dummy sections can be defined in a source module. However, the practical maximum number depends on the amount of table storage available during link-editing.

The assembler lists each external symbol identified in a source module in the external symbol dictionary, along with entries for entry symbols, common control sections, parts, and external control sections.

External symbols have a length attribute of 1. See also "WXTRN Instruction" on page 229

\section*{ICTL Instruction}

The ICTL instruction changes the begin, end, and continue columns that establish the coding format of the assembler language source statements.


\section*{begin}
specifies the begin column of the source statement. It must be a decimal self-defining term within the range of 1 to 40 , inclusive.
end
specifies the end column of the source statement. When end is specified it must be a decimal self-defining term within the range of 41 to 80 , inclusive. It must be not less than begin +5 , and must be greater than continue. If end is not specified, it is assumed to be 71 .
continue
specifies the continue column of the source statement. When specified, continue must be a decimal self-defining term within the range of 2 to 40 , and it
must be greater than begin. If continue is not specified, or if column 80 is specified as the end column, the assembler assumes that continuation lines are not allowed.

\section*{Default}

1,71,16
Use the ICTL instruction only once, at the very beginning of a source program. If no ICTL statement is used in the source program, the assembler assumes that 1 , 71 , and 16 are the begin, end, and continue columns, respectively.

With the ICTL instruction, you can, for example, increase the number of columns to be used for the identification or sequence checking of your source statements. By changing the begin column, you can even create a field before the begin column to contain identification or sequence numbers. For example, the following instruction designates the begin column as 9 and the end column as 80 . Since the end column is specified as 80 , no continuation records are recognized.
\[
\text { ICTL } \quad 9,80
\]

COPY Instruction: The ICTL instruction does not affect the format of statements brought in by a COPY instruction or generated from a library macro definition. The assembler processes these statements according to the standard begin, end, and continue columns described in "Field Boundaries" on page 14.

\section*{ISEQ Instruction}

The ISEQ instruction forces the assembler to check if the statements in a source module are in sequential order. In the ISEQ instruction, you specify the columns between which the assembler is to check for sequence numbers.

sequence_symbol
is a sequence symbol.
left
specifies the first column of the field to be sequence-checked. If specified, left must be a decimal self-defining term in the range 1 to 80 , inclusive.

\section*{right}
specifies the rightmost column of the field to be sequence checked. If specified, right must be a decimal self-defining term in the range 1 to 80 , inclusive, and must be greater than or equal to left.

If left and right are omitted, sequence checking is ended. Sequence checking can be restarted with another ISEQ statement. An ISEQ statement that is used to end sequence checking is itself sequence-checked.

The assembler begins sequence checking with the first statement line following the ISEQ instruction. The assembler also checks continuation lines.

\section*{LOCTR Instruction}

Sequence numbers on adjacent statements or lines are compared according to the 8 -bit internal EBCDIC collating sequence. When the sequence number on one line is not greater than the sequence number on the preceding line, a sequence error is flagged, and a warning message is issued, but the assembly is not ended.

If the sequence field in the preceding line is spaces, the assembler uses the last preceding line with a non-space sequence field to make its comparison.

The assembler checks only those statements that are specified in the coding of a source module. This includes any COPY instruction statement or macro instruction. The assembler does not check:
- Statements inserted by a COPY instruction
- Statements generated from model statements inside macro definitions or from model statements in open code (statement generation is discussed in detail in Chapter 7, "How to Specify Macro Definitions" on page 243
- Statements in library macro definitions

\section*{LOCTR Instruction}

The LOCTR instruction specifies multiple location counters within a control section. The assembler assigns consecutive addresses to the segments of code using one location counter before it assigns addresses to segments of coding using the next location counter.
- - symbol——OCTR
symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

By using the LOCTR instruction, you can code your control section in a logical order. For example, you can code work areas and data constants within the section of code, using them without having to branch around them:

\section*{LOCTR Instruction}


LOCTRs are ordered by their definition order. So in the previous example, the ordering is A, B and C. When there are statements in LOCTR groups, the code is generated using currently active USINGs and then moved to the final location.

\section*{Notes:}
1. The first location counter of a section, class or part is defined by the name of the START, CSECT, DSECT, RSECT, CATTR or COM instruction defining the section.
2. The LOCTR instruction defines a location counter.
3. The LOCTR continues a previously defined location counter. A location counter remains in use until it is interrupted by a LOCTR, CSECT, DSECT, or COM instruction.
4. A LOCTR instruction with the same name as a control section continues the first location counter of that section. However, an unnamed LOCTR cannot be used to continue an unnamed (private code) control section.
5. A LOCTR instruction with the same name as a LOCTR instruction in a previous control section causes that control section to be continued using the location counter specified, even though the LOCTR instruction may follow the definition (or resumption) of a different section.
6. To continue a location counter in an unnamed section, a named location counter must first be specified for the section by a LOCTR in the unnamed section.

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

The length attribute of a LOCTR name is 1 .
LOCTR instructions do not force alignment; code assembled under a location counter other than the first location counter of a control section is assembled starting at the next available byte after the previous segment.

A LOCTR name may be referenced as an ordinary symbol. If the LOCTR name does not match a section name, its value is the location counter value assigned to its first appearance, and it may have arbitrary alignment and other attributes. If the

LOCTR name is also a control section name, the value assigned is that of the origin of the control section. So a LOCTR with the same name as the CSECT will resume the first location counter within the CSECT. A CSECT instruction will resume the last location counter used.

Figure 55. LOCTR behavior with NOGOFF option
\begin{tabular}{l|l}
\hline \begin{tabular}{l} 
LOCTR \\
name
\end{tabular} & Effect \\
\hline Section & Resumes assembling with the first location counter of that section \\
\hline Other & \begin{tabular}{l} 
- If the LOCTR name was previously declared, resumes assembling with \\
- If the LOCation counter of that LOCTR group \\
new LOCTR group of statements to be assembled following the most \\
recently processed section or LOCTR group
\end{tabular} \\
\hline
\end{tabular}

Figure 56. LOCTR behavior with GOFF option
\begin{tabular}{l|l}
\hline \begin{tabular}{l} 
LOCTR \\
name
\end{tabular} & Effect \\
\hline Section & \begin{tabular}{l} 
Resumes assembling with the first location counter of the element in the \\
B_TEXT class of that section
\end{tabular} \\
\hline Class & Not allowed \\
\hline Part & Resumes assembling with the first location counter of the part \\
\hline Other & \begin{tabular}{l} 
- If the LOCTR name was previously declared, resumes assembling with \\
- If location counter of that LOCTR group \\
statements in a new LOCTR group to be assembled following the most \\
recently processed class, part, or LOCTR group.
\end{tabular} \\
\hline
\end{tabular}

\section*{LTORG Instruction}

Use the LTORG instruction so that the assembler can collect and assemble literals into a literal pool. A literal pool contains the literals you specify in a source module either after the preceding LTORG instruction, or after the beginning of the source module.

If a control section has not been established, LTORG will initiate an unnamed (private) control section.
\(\square\)
symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

\section*{LTORG Instruction}

If symbol is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the literal pool. This symbol is aligned on a boundary specified by the SECTALGN option, and has a length attribute of 1 . If bytes are skipped after the end of a literal pool to achieve alignment for the next instruction, constant, or area, the bytes are not filled with zeros. If the literal pool includes any items that require quadword alignment and the SECTALGN value defaults to 8, then the assemble of the literal causes the issue of an ASMA500W message.

The assembler ignores the borders between control sections when it collects literals into pools. Therefore, you must be careful to include the literal pools in the control sections to which they belong (for details, see "Addressing Considerations" on page 195.

The creation of a literal pool gives the following advantages:
- Automatic organization of the literal data into sections that are correctly aligned and arranged so that minimal space is wasted in the literal pool.
- Assembling of duplicate data into the same area.
- Because all literals are cross-referenced, you can find the literal constant in the pool into which it has been assembled.

\section*{Literal Pool}

A literal pool is created under the following conditions:
- Immediately after a LTORG instruction.
- If no LTORG instruction is specified, and no LOCTRs are used in the first control section, a literal pool generated after the END statement is created at the end of the first control section, and appears in the listing after the END statement.
- If no LTORG instruction is specified, and LOCTRs are used in the first control section, a literal pool generated after the END statement is created at the end of the most recent LOCTR segment of the first section, and appears in the listing after the END statement.
- To force the literal pool to the end of the control section when using LOCTRs, you must resume the last LOCTR of the CSECT before the LTORG statement (or before the END statement if no LTORG statement is specified).

Each literal pool has five segments into which the literals are stored (a) in the order that the literals are specified, and (b) according to their assembled lengths, which, for each literal, is the total explicit or implied length, as described below.
- The first segment contains all literal constants whose assembled lengths are a multiple of 16.
- The second segment contains those whose assembled lengths are a multiple of 8 , but not of 16 .
- The third segment contains those whose assembled lengths are a multiple of 4 , but not a multiple of 8 .
- The fourth segment contains those whose assembled lengths are even, but not a multiple of 4 .
- The fifth segment contains all the remaining literal constants whose assembled lengths are odd.

Since each literal pool is aligned on a SECTALGN alignment, this guarantees that all literals in the second segment are doubleword aligned; in the third segment, fullword aligned; and, in the fourth, halfword aligned. The minimum value of SECALGN is doubleword, so quadword alignment is not guaranteed. No space is wasted except, possibly, at the origin of the pool, and in aligning to the start of the statement following the literal pool.

Literals from the following statements are in the pool, in the segments indicated by the parenthesized numbers:
\begin{tabular}{|c|c|c|c|}
\hline FIRST & START & 0 & \\
\hline & MVC & T0, =3F'9' & (3) \\
\hline & AD & 2,=D'7' & (2) \\
\hline & IC & 2,=XL1'8' & (5) \\
\hline & MVC & MTH, =CL3'JAN \({ }^{\prime}\) & (5) \\
\hline & LM & 4,5,=2F'1, \({ }^{\prime}\) & (2) \\
\hline & AH & 5, = \({ }^{\prime} 33^{\prime}\) & (4) \\
\hline & L & 2, =A (ADDR) & (3) \\
\hline & MVC & FIVES, =XL16'05' & (1) \\
\hline
\end{tabular}

\section*{Addressing Considerations}

If you specify literals in source modules with multiple control sections, you should:
- Write a LTORG instruction at the end of each control section, so that all the literals specified in the section are assembled into the one literal pool for that section. If a control section is divided and interspersed among other control sections, you should write a LTORG instruction at the end of each segment of the interspersed control section.
- When establishing the addressability of each control section, make sure (a) that all of the literal pool for that section is also addressable, by including it within a USING range, and (b) that the literal specifications are within the corresponding USING domain. The USING range and domain are described in UUSING Instruction" on page 218 .

All the literals specified after the last LTORG instruction, or, if no LTORG instruction is specified, all the literals in a source module are assembled into a literal pool at the end of the first control section. You must then make this literal pool addressable, along with the addresses in the first control section. This literal pool is printed in the program listing after the END instruction.

\section*{Duplicate Literals}

If you specify duplicate literals within the part of the source module that is controlled by a LTORG instruction, only one literal constant is assembled into the pertinent literal pool. This also applies to literals assembled into the literal pool at the end of the first or only control section of a source module that contains no LTORG instructions.

Literals are duplicates only if their specifications are identical, not if the object code assembled happens to be identical.

\section*{MNOTE Instruction}

When two literals specifying identical A-type, Y-type or S-type address constants contain a reference to the value of the location counter (*), both literals are assembled into the literal pool. This is because the value of the location counter may be different in the two literals. Even if the location counter value is the same for both, they are still both assembled into the literal pool.

The following examples show how the assembler stores pairs of literals, if the placement of each pair is controlled by the same LTORG statement.
```

=X'F0' Both are
=C'0' stored
=XL3'0' Both are
=HL3'0' stored
=A(*+4) Both are
=A(*+4) stored
=X'FFFF' Identical,
=X'FFFF' only one copy is stored

```

\section*{MNOTE Instruction}

The MNOTE instruction generates your own error messages or displays intermediate values of variable symbols computed during conditional assembly.

The MNOTE instruction can be used inside macro definitions or in open code, and its operation code can be created by substitution. The MNOTE instruction causes the generation of a message that is given a statement number in the printed listing.

sequence_symbol
is a sequence symbol.

\section*{severity}
is a severity code. The severity operand may be any absolute expression allowed in the operand field of a SETA instruction. The term must have a value in the range 0 through 255 . The severity code is used to determine the return code issued by the assembler when it returns control to the operating system. The severity may also change the value of the system variable symbols \&SYSM_HSEV and \&SYSM_SEV (see "\&SYSM_HSEV System Variable Symbol" on page 279 and "\&SYSM_SEV System Variable Symbol" on page 280.
message
is the message text. It may be any combination of characters enclosed in single quotation marks.
The rules that apply to this character string are as follows:
- Variable symbols are allowed. The single quotation marks that enclose the message can be generated from variable symbols.
- Two ampersands and two single quotation marks are needed to generate an ampersand or a single quotation mark, respectively. If variable symbols have ampersands or single quotation marks as values, the values must be coded as two ampersands or two single quotation marks.
- If the number of characters in the character string plus the rest of the MNOTE operand exceeds 1024 bytes the assembler issues diagnostic message
Note: The maximum length of the second operand is three less than the maximum supported length of SETC character string.
ASMA062E Illegal operand format
The * notation can be used as a continuation.
Here is an example taken from a CICS® macro:
```

MNOTE 8,'FIELD IS DEFINED OUTSIDE OF THE SIZE OPERAND'
MNOTE *,'PARAMETERS SPECIFIED IN THE DFHMDI MACRO,'
MNOTE *,'MACRO REQUEST IS IGNORED.'

```

A further advantage of this approach is that only one severity 8 error is seen instead of three.
- Double-byte data is permissible in the operand field when the DBCS assembler option is specified. The double-byte data must be valid.
- The DBCS ampersand and apostrophe are not recognized as delimiters.
- A double-byte character that contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.

Remarks: Any remarks for the MNOTE instruction statement must be separated by one or more spaces from the single quotation mark that ends the message.

If severity is provided, or severity is omitted but the comma separating it from message is present, the message is treated as an error message; otherwise the message is treated as comments. The rules for specifying the contents of severity are:
- The severity code can be specified as any arithmetic expression allowed in the operand field of a SETA instruction. The expression must have a value in the range 0 through 255.

\section*{Example:}

MNOTE 2,'ERROR IN SYNTAX'
The generated result is:
2,ERROR IN SYNTAX
- If the severity code is omitted, but the comma separating it from the message is present, the assembler assigns a default value of 1 as the severity code.

\section*{Example:}
MNOTE ,'ERROR, SEV 1'

The generated result is:
, ERROR, SEV 1
- An asterisk in the severity code subfield causes the message and the asterisk to be generated as a comment statement.

\section*{Example:}
```

MNOTE *,'NO ERROR'

```

The generated result is:
*,NO ERROR
- If the severity code subfield is omitted, including the comma separating it from the message, the assembler generates the message as a comment statement.

\section*{Example:}

MNOTE 'NO ERROR'
The generated result is:
NO ERROR

\section*{Notes:}
1. An MNOTE instruction causes a message to be printed, if the current PRINT option is ON, even if the PRINT NOGEN option is specified.
2. The statement number of the message generated from an MNOTE instruction with a severity code is listed among any other error messages for the current source module. However, the message is printed only if the severity code specified is greater than or equal to the severity code \(n n n\) specified in the FLAG(nnn) assembler option.
3. The statement number of the comments generated from an MNOTE instruction without a severity code is not listed among other error messages.

\section*{OPSYN Instruction}

The OPSYN instruction defines or deletes symbolic operation codes.
The OPSYN instruction has two formats. The first format defines a new operation code to represent an existing operation code, or to redefine an existing operation code for:
- Machine and extended mnemonic branch instructions
- Assembler instructions, including conditional assembly instructions
- Macro instructions

\section*{Define Operation Code}


The second format deletes an existing operation code for:
- Machine and extended mnemonic branch instructions
- Assembler instructions, including conditional assembly instructions
- Macro instructions

\section*{Delete Operation Code}
-- operation_code_1—OPSYN
symbol
is one of the following:
- An ordinary symbol that is not the same as an existing operation code
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol and is not the same as an existing operation code
operation_code_1
is one of the following:
- An operation code described in this chapter, or in Chapter 4, "Machine Instruction Statements" on page 78 or Chapter 9, "How to Write Conditional Assembly Instructions" on page 318
- The operation code defined by a previous OPSYN instruction
- The name of a previously defined macro.
operation_code_2
is one of the following:
- An operation code described in this chapter, or in Chapter 4, "Machine Instruction Statements" on page 78 or Chapter 9, "How to Write Conditional Assembly Instructions" on page 318
- The operation code defined by a previous OPSYN instruction

In the first format, the OPSYN instruction assigns the properties of the operation code denoted by operation_code_2 to the ordinary symbol denoted by symbol or the operation code denoted by operation_code_1.

In the second format, the OPSYN instruction causes the operation code specified in operation_code_1 to lose its properties as an operation code.

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

The symbol in the name field can represent a valid operation code. It loses its current properties as if it had been defined in an OPSYN instruction with a space-filled operand field. In the following example, L and LR both possess the properties of the LR machine instruction operation code:
L OPSYN LR

When the same symbol appears in the name field of two OPSYN instructions, the latest definition takes precedence. In the example below, STORE now represents the STH machine operation:
\begin{tabular}{lll} 
STORE & OPSYN & ST \\
STORE & OPSYN & STH
\end{tabular}

Note: OPSYN is not processed during lookahead mode (see"Lookahead" on page 340. Therefore it cannot be used during lookahead to replace an opcode that must be processed during lookahead, such as COPY. For example, assuming

AFTER is defined in COPYBOOK, the following code gives an ASMA042E error (Length attribute of symbol is unavailable):
```

AIF (L'AFTER LT 2).BEYOND
OPCOPY OPSYN COPY OPSYN not processed during look ahead
OPCOPY COPYBOOK OPCOPY fails
.BEYOND ANOP ,

```

\section*{Redefining Conditional Assembly Instructions}

A redefinition of a conditional assembly instruction only comes into effect in macro definitions occurring after the OPSYN instruction. The original definition is always used when a macro instruction calls a macro that was defined and edited before the OPSYN instruction.

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


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 instructions are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC. In subsequent calls of the macro MAC, 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 instructions. 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.

\section*{ORG Instruction}

The ORG instruction alters the setting of the location counter and thus controls the structure of the current control section. This redefines portions of a control section.

If a control section has not been previously established, ORG will initiate an unnamed (private) control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If symbol denotes an ordinary symbol, the ordinary symbol is defined with the value that the location counter had before the ORG statement is processed.

\section*{expression}
is a relocatable expression, the value of which is used to set the location counter. If expression is omitted, the location counter is set to the next available location for the current control section.

\section*{boundary}
is an absolute expression that must be a power of 2 with a range from 8 (doubleword) to 4096 (page). If boundary exceeds the SECTALGN value, message ASMA500E is issued.
offset
Any absolute expression
If boundary or offset are provided, then the resultant location counter is calculated by rounding the expression up to the next higher boundary and then adding the offset value.

In general, symbols used in expression need not have been previously defined. However, the relocatable component of expression (that is, the unpaired relocatable term) must have been previously defined in the same control section in which the ORG statement appears, or be equated to a previously defined value.

A length attribute reference to the name of a ORG instruction is always invalid. Message ASMSO42E is issued, and a default value of 1 is assigned.

An ORG statement cannot be used to specify a location below the beginning of the control section in which it appears. For example, the following statement is not correct if it appears less than 500 bytes from the beginning of the current control section.
\[
\text { ORG } \quad *-500
\]

This is because the expression specified is negative, and sets the location counter to a value larger than the assembler can process. The location counter wraps around (the location counter is discussed in detail in "Location Counter" on page 36).

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

An ORG statement cannot be used to change sections or LOCTR segments. For example:
\begin{tabular}{lll} 
AA & CSECT & \\
\(X\) & \(D S\) & \(D\) \\
\(Y\) & \(D S\) & \(F\) \\
BB & CSECT & \\
& ORG & \(Y\)
\end{tabular}
is invalid, because the section containing the ORG statement \((\mathrm{BB})\) is not the same as the section in AA in which the ORG operand expression \(Y\) is defined.

With the ORG statement, you can give two instructions the same location counter values. In such a case, the second instruction does not always eliminate the effects of the first instruction. Consider the following example:
\begin{tabular}{lll} 
ADDR & DC & A(ADDR) \\
& ORG & \(*-4\) \\
B & DC & \(C^{\prime} B E T A^{\prime}\)
\end{tabular}

In this example, the value of \(B\) ('BETA') is destroyed by the relocation of ADDR during linkage editing.

The following example shows some examples of ORG using the boundary and offset operands:
```

origin csect
ds 235x Define 235 bytes
org origin,,3 Move location counter back to start + 3
org *,8 Align on 8 byte boundary
org *,8,-2 Align to 8 byte boundary -2 bytes
translate dc cl256' '
Define aligned translate table
org translate+c'a'
dc c'ABCDEFGHI'
org translate+c'j'
dc c'JKLMNOPQR'
org translate+c's'
dc c'STUVWXYZ'
org translate+c'A'
dc c'ABCDEFGHI'
org translate+c'J'
dc c'JKLMNOPQR'
org translate+c'S'
dc c'STUVWXYZ'
org ,
end

```

Using Figure 57 on page 203 as an example, to build a translate table (for example, to convert EBCDIC character code into some other internal code):
1. Define the table (see 1 in Figure 57) as being filled with zeros.
2. Use the ORG instruction to alter the location counter so that its counter value indicates a specific location (see 2 in Figure 57) within the table.
3. Redefine the data (see \(\mathbf{3}\) in Figure 57) to be assembled into that location.
4. After repeating the first three steps (see 4 in Figure 57) until your translate table is complete, use an ORG instruction with a null operand field to alter the location counter. The counter value then indicates the next available location (see 5 in Figure 57) in the current control section (after the end of the translate table).

Both the assembled object code for the whole table filled with zeros, and the object code for the portions of the table you redefined, are printed in the program listings. However, the data defined later is loaded over the previously defined zeros and becomes part of your object program, instead of the zeros.

That is, the ORG instruction can cause the location counter to be set to any part of a control section, even the middle of an instruction, into which you can assemble data. It can also cause the location counter to be set to the next available location so that your program can be assembled sequentially.


Figure 57. Building a Translate Table

\section*{POP Instruction}

The POP instruction restores the PRINT, USING or ACONTROL status saved by the most recent PUSH instruction.


Note:
\({ }^{1}\) Each keyword from this group may be selected only once.

\section*{sequence_symbol}
is a sequence symbol.

\section*{PRINT}
instructs the assembler to restore the PRINT status to the status saved by the most recent PUSH instruction.

\section*{USING}
instructs the assembler to restore the USING status to the status saved by the most recent PUSH instruction.

\section*{ACONTROL}
instructs the assembler to restore the ACONTROL status to the status saved by the most recent PUSH instruction.

\section*{NOPRINT}
instructs the assembler to suppress the printing of the POP statement in which it is specified.

The POP instruction causes the status of the current PRINT, USING or ACONTROL instruction to be overridden by the PRINT, USING or ACONTROL status saved by the last PUSH instruction. For example:
\begin{tabular}{lll} 
& PRINT & GEN \\
DCMAC & X,27 & Printed macro generated code \\
\\
DC & \(X^{\prime} 27^{\prime}\) & Call macro to generate DC
\end{tabular}

\section*{PRINT Instruction}

The PRINT instruction controls the amount of detail printed in the listing of programs.

sequence_symbol
is a sequence symbol.
operand
is an operand from one of the groups of operands described below. If a null operand is supplied, it is accepted by the assembler with no effect on the other operands specified. The operands are listed in hierarchic order. The effect, if any, of one operand on other operands is also described.


\section*{ON}
instructs the assembler to print, or resume printing, the source and object section of the assembler listing.

OFF
instructs the assembler to stop printing the source and object section of the assembler listing. A subsequent PRINT ON instruction resumes printing.
When this operand is specified the printing actions requested by the GEN, DATA, MCALL, and MSOURCE operands do not apply.
\(\square\)

\section*{GEN}
instructs the assembler to print all statements generated by the processing of a macro. This operand does not apply if PRINT OFF has been specified.

\section*{NOGEN}
instructs the assembler not to print statements generated by conditional assembly or the processing of a macro. This applies to all levels of macro nesting; no generated code is displayed while PRINT NOGEN is in effect. If this operand is specified, the DATA operand does not apply to constants that are generated during macro processing. Also, if this operand is specified, the MSOURCE operand does not apply. 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.

The MNOTE instruction always causes a message to be printed.
\(\square\)

\section*{NODATA}
instructs the assembler to print only the first 8 bytes of the object code of constants. This operand does not apply if PRINT OFF has been specified. If PRINT NOGEN has been specified, this operand does not apply to constants generated during macro processing.

\section*{DATA}
instructs the assembler to print the object code of all constants in full. This operand does not apply if PRINT OFF has been specified. If PRINT NOGEN has been specified, this operand does not apply to constants generated during macro processing.


NOMCALL
instructs the assembler to suppress the printing of nested macro call instructions.

\section*{MCALL}
instructs the assembler to print nested macro call instructions, including the name of the macro definition to be processed and the operands and values passed to the macro definition. The assembler only prints the operands and comments up to the size of its internal processing buffer. If this size is exceeded the macro call instruction is truncated, and the characters ... MORE are added to the end of the printed macro call. This does not affect the processing of the macro call.

This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.


\section*{MSOURCE}
instructs the assembler to print the source statements generated during macro processing, as well as the assembled addresses and generated object code of the statements. This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.

\section*{NOMSOURCE}
instructs the assembler to suppress the printing of source statements generated during macro processing, without suppressing the printing of the assembled addresses and generated object code of the statements. This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.


\section*{UHEAD}
instructs 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. This operand does not apply if PRINT OFF has been specified.

\section*{NOUHEAD}
instructs the assembler not to print a summary of active USINGs.


\section*{NOPRINT}
instructs the assembler to suppress the printing of the PRINT statement in which it is specified. The NOPRINT operand may only be specified in conjunction with one or more other operands.

The PRINT instruction can be specified any number of times in a source module, but only those operands actually specified in the instruction change the current print status.

PRINT options can be generated by macro processing during conditional assembly. However, at assembly time, all options are in force until the assembler encounters a new and opposite option in a PRINT instruction.

The PUSH and POP instructions, described in "PUSH Instruction" on page 209 and "POP Instruction" on page 204, also influence the PRINT options by saving and restoring the PRINT status.

You can override the effect of the operands of the PRINT instruction by using the PCONTROL assembler option. For more information about this option, see the HLASM Programmer's Guide.

Unless the NOPRINT operand is specified, or the assembler listing is suppressed by the NOLIST assembler option, the PRINT instruction itself is printed.

\section*{Process Statement}

The process statement is described under "**PROCESS Statement" on page 102.

\section*{PUNCH Instruction}

The PUNCH instruction creates a record containing a source or other statement, or an object record, to be written to the object file.

sequence_symbol
is a sequence symbol.

\section*{string}
is a character string of up to 80 characters, enclosed in single quotation marks. All 256 characters in the EBCDIC character set are allowed in the character string. Variable symbols are also allowed.

Double-byte data is permissible in the operand field when the DBCS assembler option is specified. However, the following rules apply to double-byte data:
- The DBCS ampersand and the single quotation mark are not recognized as delimiters.
- A double-byte character that contains the value of an EBCDIC ampersand or a single quotation mark in either byte is not recognized as a delimiter when enclosed by SO and SI.

The position of each character specified in the PUNCH statement corresponds to a column in the record to be punched. However, the following rules apply to ampersands and single quotation marks:
- A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.
- A pair of ampersands is punched as one ampersand.
- A pair of single quotation marks is punched as one single quotation mark.
- An unpaired single quotation mark followed by one or more spaces simply ends the string of characters punched. If a non-space character follows an unpaired single quotation mark, an error message is issued and nothing is punched.

Only the characters punched, including spaces, count toward the maximum of 80 allowed.

The PUNCH instruction causes the data in its operand to be punched into a record. One PUNCH instruction produces one record, but as many PUNCH instructions as necessary can be used.

You can code PUNCH statements in:
- A source module to produce control statements for the linker. The linker uses these control statements to process the object module.
- Macro definitions to produce, for example, source statements in other computer languages or for other processing phases.

The assembler writes the record produced by a PUNCH statement when it writes the object deck. The ordering of this record in the object deck is determined by the order in which the PUNCH statement is processed by the assembler. The record appears after any object deck records produced by previous statements, and before any other object deck records produced by subsequent statements.

The PUNCH instruction statement can appear anywhere in a source module. If a PUNCH instruction occurs before the first control section, the resultant record punched precedes all other records in the object deck.

The record punched as a result of a PUNCH instruction is not a logical part of the object deck, even though it can be physically interspersed in the object deck.

\section*{Notes:}
1. The identification and sequence number field generated as part of other object deck records is not generated for the record punched by the PUNCH instruction.
2. If the NODECK and NOOBJECT assembler options are specified, no records are punched for the PUNCH instruction.
3. Do not use the PUNCH instruction if the GOFF option is specified, as the resulting file may be unusable.

\section*{PUSH Instruction}

The PUSH instruction saves the current PRINT, USING or ACONTROL status in push-down storage on a last-in, first-out basis. You restore this PRINT, USING or ACONTROL status later, also on a last-in, first-out basis, by using a POP instruction.


Note:
\({ }^{1}\) Each keyword from this group may be selected only once.

\section*{sequence_symbol}
is a sequence symbol.

\section*{PRINT}
instructs the assembler to save the PRINT status in a push-down stack.

\section*{USING}
instructs the assembler to save the USING status in a push-down stack.

\section*{ACONTROL}
instructs the assembler to save the ACONTROL status in a push-down stack.

\section*{NOPRINT}
instructs the assembler to suppress the printing of the PUSH statement in which it is specified.

The PUSH instruction only causes the status of the current PRINT, USING or ACONTROL instructions to be saved. The PUSH instruction does not:
- Change the status of the current PRINT or ACONTROL instructions
- Imply a DROP instruction, or change the status of the current USING instructions

\section*{REPRO Instruction}

The REPRO instruction causes the data specified in the statement that follows to be punched into records, to be written to the object file.


\section*{sequence_symbol}
is a sequence symbol.

The REPRO instruction can appear anywhere in a source module. One REPRO instruction produces one punched record, but as many REPRO instructions as necessary can be used. Records are created as the object file is being created, so records may be interspersed among object code. The punched records are part of the object file, but are not intended to contain normal object code or symbols.

The statement to be reproduced can contain any of the 256 characters in the EBCDIC character set, including spaces, ampersands, and single quotation marks. Unlike the PUNCH instruction, the REPRO instruction does not allow values to be substituted into variable symbols before the record is punched.

\section*{Notes:}
1. The identification and sequence numbers generated as part of other object deck records is not generated for records punched by the REPRO instruction.
2. If the NODECK and NOOBJECT assembler options are specified, no records are punched for the REPRO instruction, or for the object deck of the assembly.
3. Since the text of the line following a REPRO statement is not validated or changed in any way, it can contain double-byte data, but this data is not validated.
4. Do not use the REPRO instruction if the GOFF option is specified, as the resulting file may be unusable.

\section*{RMODE Instruction}

The RMODE instruction specifies the residence mode to be associated with control sections in the object deck.

name
is the name field that associates the residence mode with a control section. If there is a symbol in the name field, it must also appear in the name field of a START, CSECT, RSECT, or COM instruction in this assembly. If the name field is space-filled, there must be an unnamed control section in this assembly. If the name field contains a sequence symbol (see "Symbols" on page 29 for details), it is treated as a blank name field.

24 specifies that a residence mode of 24 is to be associated with the control section; that is, the control section must be resident below 16 megabytes.

31 specifies that a residence mode of either 24 or 31 is to be associated with the control section; that is, the control section can be resident above or below 16 megabytes.

64 Specifies that a residence mode of 64 is to be associated with the control section (see "64-bit Addressing Mode" on page 101].

ANY
is understood to mean RMODE 31.
Any field of this instruction may be generated by a macro, or by substitution in open code.

\section*{Notes:}
1. RMODE can be specified anywhere in the assembly. It does not initiate an unnamed control section.
2. An assembly can have multiple RMODE instructions; however, two RMODE instructions cannot have the same name field.
3. The valid and invalid combinations of AMODE and RMODE are shown in the following table. Note that combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64-bit Addressing Mode" on page 101.

Figure 58. AMODE/RMODE Combinations
\begin{tabular}{lccc}
\hline & RMODE 24 & RMODE 31 & RMODE 64 \\
\hline AMODE 24 & OK & invalid & invalid \\
\hline AMODE 31 & OK & OK & invalid \\
\hline AMODE ANYIANY31 & OK & OK & invalid \\
\hline AMODE 64 & OK & OK & OK \\
\hline
\end{tabular}
4. AMODE or RMODE cannot be specified for an unnamed common control section.
5. The defaults used when zero or one MODE is specified are shown in the following table. Note that combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64-bit Addressing Mode" on page 101

Figure 59. AMODE/RMODE Defaults
\begin{tabular}{ll}
\hline Specified & Default \\
\hline Neither & AMODE 24,RMODE 24 \\
\hline AMODE 24 & RMODE 24 \\
\hline AMODE 31 & RMODE 24 \\
\hline AMODE ANYIANY31 & RMODE 24 \\
\hline RMODE 24 & AMODE 24 \\
\hline RMODE 31 (was ANY) & AMODE 31 \\
\hline AMODE 64 & RMODE 31 \\
\hline RMODE 64 & AMODE 64 \\
\hline
\end{tabular}

\section*{RSECT Instruction}

The RSECT instruction initiates a read-only executable control section or indicates the continuation of a read-only executable control section.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

When an executable control section is initiated by the RSECT instruction, the assembler automatically checks the control section for possible coding violations of program reenterability, regardless of the setting of the RENT assembler option. As the assembler cannot check program logic, the checking is not exhaustive.
Non-reentrant code is diagnosed by a warning message.
The RSECT instruction can be used anywhere in a source module after the ICTL instruction. If it is used to initiate the first executable control section, it must not be preceded by any instruction that affects the location counter and thereby causes the first control section to be initiated.

If symbol denotes an ordinary symbol, the ordinary symbol identifies the control section. If several RSECT instructions within a source module have the same symbol in the name field, the first occurrence initiates the control section and the
rest indicate the continuation of the control section. The ordinary symbol denoted by symbol represents the address of the first byte in the control section, and has a length attribute value of 1 .

If symbol is not specified, or if name is a sequence symbol, the RSECT instruction initiates or indicates the continuation of the unnamed control section.

See "CSECT Instruction" on page 123 for a discussion on the interaction between RSECT and the GOFF assembler option.

The beginning of a control section is aligned on a boundary determined by the SECTALGN option. However, when an interrupted control section is continued using the RSECT instruction, the location counter last specified in that control section is continued.

The source statements following a RSECT instruction that either initiate or indicate the continuation of a control section are assembled into the object code of the control section identified by that RSECT instruction.

\section*{Notes:}
1. The assembler indicates that a control section is read-only by setting the read-only attribute in the object module.
2. The end of a control section or portion of a control section is marked by (a) any instruction that defines a new or continued control section, or (b) the END instruction.

\section*{SPACE Instruction}

The SPACE instruction inserts one or more blank lines in the listing of a source module. This separates sections of code on the listing page.

sequence_symbol
is a sequence symbol.
number_of_lines
is an absolute expression that specifies the number of lines to be left blank. You may use any absolute expression to specify number_of_lines. If number_of_lines is omitted, one line is left blank. If number_of_lines has a value greater than the number of lines remaining on the listing page, the instruction has the same effect as an EJECT statement.

The SPACE statement itself is not printed in the listing unless a variable symbol is specified as a point of substitution in the statement, in which case the statement is printed before substitution occurs.

\section*{START Instruction}

The START instruction can be used to initiate the first or only control section of a source module, and optionally to set an initial location counter value.

symbol
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol
expression
is an absolute expression, the value of which the assembler uses to set the location counter to an initial value for the source module.

Any symbols referenced in expression must have been previously defined.
The START instruction must be the first instruction of the first executable control section of a source module. It must not be preceded by any instruction that affects the location counter, and thereby causes the first control section to be initiated.

Use the START instruction to initiate the first or only control section of a source module, because it:
- Determines exactly where the first control section is to begin, thus avoiding the accidental initiation of the first control section by some other instruction.
- Gives a symbolic name to the first control section, which can then be distinguished from the other control sections listed in the external symbol dictionary.
- Specifies the initial setting of the location counter for the first or only control section.

If symbol denotes an ordinary symbol, the ordinary symbol identifies the first control section. It must be used in the name field of any CSECT instruction that indicates the continuation of the first control section. The ordinary symbol denoted by symbol represents the address of the first byte in the control section, and has a length attribute value of 1 .

If symbol is not specified, or if name is a sequence symbol, the START instruction initiates an unnamed control section.

The assembler uses the value expression in the operand field, if specified, to set the location counter to an initial value for the source module. All control sections are aligned on the boundary specified by the SECTALGN option. Therefore, if the value specified in expression is not divisible by the SECTALGN value, the assembler sets the initial value of the location counter to the next higher required boundary. If expression is omitted, the assembler sets the initial value to 0 .

The source statements that follow the START instruction are assembled into the first control section. If a CSECT instruction indicates the continuation of the first control section, the source statements that follow this CSECT instruction are also assembled into the first control section.

Any instruction that defines a new or continued control section marks the end of the preceding control section. The END instruction marks the end of the control section in effect.

\section*{TITLE Instruction}

The TITLE instruction:
- Provides headings for each page of the source and object section of the assembler listing. If the first statement in your source program is an ICTL instruction or a *PROCESS statement then the title is not printed on the first page of the Source and Object section, because each of these instructions must precede all other instructions.
- Identifies the assembly output records of your object modules. You can specify up to 8 identification characters that the assembler includes as a deck ID in all object records, beginning at byte 73 . If the deck ID is less than 8 characters, the assembler puts sequence numbers in the remaining bytes up to byte 80 .

name
You can specify name only once in the source module. It is one of the following:
- A string of printable characters
- A variable symbol that has been assigned a string of printable characters
- A combination of the above
- A sequence symbol

Except when the name is a sequence symbol, the assembler uses the first 8 characters you specify, and discards the remaining characters without warning.

\section*{title_string}
is a string of 1 to 100 characters enclosed in single quotation marks
If two or more TITLE instructions are together, the title provided by the last instruction is printed as the heading.

\section*{Deck ID in Object Records}

When you specify the name, and it is not a sequence symbol, it has a special significance. The assembler uses the name value to generate the deck ID in object records. The deck ID is placed in the object records starting at byte 73. It is not generated for records produced by the PUNCH and REPRO instructions. The name value does not need to be on the first TITLE instruction.

The name value is not defined as a symbol, so it can be used in the name entry of any other statement in the same source module, provided it is a valid ordinary symbol.

GOFF Assembler Option (MVS and CMS): When you specify the GOFF assembler option the deck ID is not generated.

\section*{Printing the Heading}

The character string denoted by title_string is printed as a heading at the top of each page of the source and object section of the assembler listing. The heading is printed beginning on the page in the listing that follows the page on which the TITLE instruction is specified. A new heading is printed each time a new TITLE instruction occurs in the source module. If the TITLE instruction is the first instruction in the source module the heading is printed on the first page of the listing.

When a TITLE instruction immediately follows an EJECT instruction, the assembler changes the title but does not perform an additional page-eject.

\section*{Printing the TITLE Statement}

The TITLE statement is printed in the listing when you specify a variable symbol in the name, or in the title_string, in which case the statement is printed before substitution occurs.

\section*{Sample Program Using the TITLE Instruction}

The following example shows three TITLE instructions:
```

PGM1 TITLE 'The First Heading'
PGM1 CSECT
USING PGM1,12 Assign the base register
TITLE 'The Next Heading'
LR 12,15 Load the base address
\&VARSYM SETC 'Value from Variable Symbol'
TITLE 'The \&VARSYM'
BR 14 Return
END

```

After the program is assembled, the characters PGM1 are placed in bytes 73 to 76 of all object records, and the heading appears at the top of each page in the listing as shown in Figure 60 on page 217 The TITLE instruction at statement 7 is printed because it contains a variable symbol.


Figure 60. Sample Program Using TITLE Instruction

\section*{Page Ejects}

Each inline TITLE statement causes the listing to be advanced to a new page before the heading is printed unless it is preceded immediately by one of the following:
- A CEJECT instruction
- An EJECT instruction
- A SPACE instruction that positions the current print line at the start of a new page
- A TITLE instruction

If the TITLE statement appears in a macro or contains a variable symbol and PRINT NOGEN is specified, the listing is not advanced to a new page.

\section*{Valid Characters}

Any printable character specified appears in the heading, including spaces. Double-byte data can be used when the DBCS assembler option is specified. The double-byte data must be valid. Variable symbols are allowed. However, the following rules apply to ampersands and single quotation marks:
- The DBCS ampersand and single quotation mark are not recognized as delimiters.
- A double-byte character that contains the value of an EBCDIC ampersand or single quotation mark in either byte is not recognized as a delimiter when enclosed by SO and SI.
- A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.
- A pair of ampersands is printed as one ampersand.
- A pair of single quotation marks is printed as one single quotation mark.
- An unpaired single quotation mark followed by one or more spaces simply ends the string of characters printed. If a non-space character follows an unpaired single quotation mark, the assembler issues an error message and prints no heading.

\section*{USING Instruction}

Only the characters printed in the heading count toward the maximum of 100 characters allowed. If the count of characters to be printed exceeds 100, the heading that is printed is truncated and error diagnostic message
ASMA062E Illegal operand format
is issued.

\section*{USING Instruction}

The USING instruction specifies a base address and range and assigns one or more base registers. If you also load the base register with the base address, you have established addressability in a control section. If a control section has not been established, USING will initiate an unnamed (private) control section.

To use the USING instruction correctly, you should know:
- Which locations in a control section are made addressable by the USING instruction
- Where in a source module you can use implicit addresses in instruction operands to refer to these addressable locations

Base Address: The term base address is used throughout this manual to mean the location counter value within a control section from which the assembler can compute displacements to locations, or addresses, within the control section. Don't confuse this with the storage address of a control section when it is loaded into storage at execution time.

The USING instruction has three formats:
- The first format specifies a base address, an optional range, and one or more base registers. This format of the USING instruction is called an ordinary USING instruction, and is described under "Ordinary USING Instruction" on page 220.
- The second format specifies a base address, an optional range, one or more base registers, and a USING label which may be used as a symbol qualifier. This format of the USING instruction is called a labeled USING instruction, and is described under "Labeled USING Instruction" on page 223
- The third format specifies a base address, an optional range, and a relocatable expression instead of one or more base registers. This format of a USING instruction is called a dependent USING instruction, and is described under "Dependent USING Instruction" on page 226 If a USING label is also specified, this format of the USING instruction is called a labeled dependent USING instruction.
Note: The assembler identifies and warns about statements where the implied alignment of an operand does not match the requirements of the instruction. However, if the base for a USING is not aligned on the required boundary, the assembler cannot diagnose a problem. For example:
\begin{tabular}{|c|c|c|c|}
\hline \multirow[t]{2}{*}{DS1} & \multicolumn{2}{|l|}{DSECT} & \\
\hline & DS & H & \\
\hline REGPAIR & DS & 2ADL8 & Halfword alignment \\
\hline DS2 & DSECT & & \\
\hline \multicolumn{2}{|l|}{\multirow[t]{2}{*}{\[
\begin{aligned}
& \text { REGPAIR_ALIGN } \\
& \text { CSECT }
\end{aligned}
\]}} & DS 2ADL8 & Doubleword alignment \\
\hline & & & \\
\hline \multicolumn{3}{|c|}{...} & \\
\hline & USING & DS1,R1 & Ordinary USING \\
\hline & USING & DS2,REGPAIR & Dependent USING \\
\hline & STPQ & R0, REGPAIR & REGPAIR is not a quadword \\
\hline & STPQ & R0,REGPAIR_ALIGN & But REGPAIR_ALIGN is \\
\hline
\end{tabular}

The first STPQ instruction is diagnosed as an alignment error. The second STPQ instruction is not, even though the same storage location is implied by the code.

You must take care to ensure base addresses match the alignment requirements of storage mapped by a USING. For a description of the alignment requirements of instructions, see the relevant Principles of Operation.

\section*{How to Use the USING Instruction}

Specify the USING instruction so that:
- All the required implicit addresses in each control section lie within a USING range.
- All the references for these addresses lie within the corresponding USING domain.

You could, therefore, place all ordinary USING instructions at the beginning of the control section and specify a base address in each USING instruction that lies at the beginning of each control section.

For Executable Control Sections: To establish the addressability of an executable control section defined by a START or CSECT instruction, specify a base address and assign a base register in the USING instruction. At execution time, the base register must be loaded with the correct base address.

If a control section requires addressability to more than 4096 bytes, you must assign more than one base register, or make implicit references using only instructions supporting 20-bit displacements ("long displacements"). This establishes the addressability of the entire control section with one USING instruction.

For Reference Control Sections: A dummy section is a reference control section defined by the DSECT instructions. To establish the addressability of a dummy section, specify the address of the first byte of the dummy section as the base address, so that all its addresses lie within the pertinent USING range. The address you load into the base register must be the address of the storage area being described by the dummy section. However, if all references to fields withing the DSECT are made with instructions supporting long displacements, the base address need not be the first byte of the dummy section.

When you refer to symbolic addresses in the dummy section, the assembler computes displacements accordingly. However, at execution time, the assembled addresses refer to the location of real data in the storage area.

\section*{Base Registers for Absolute Addresses}

Absolute addresses used in a source module must also be made addressable. Absolute addresses require a base register other than the base register assigned to relocatable addresses (as described above).

However, the assembler does not need a USING instruction to convert absolute implicit addresses in the range 0 through 4095 to their explicit form. The assembler uses register 0 as a base register. Displacements are computed from the base address 0 , because the assembler assumes that a base or index of 0 implies that a zero quantity is to be used in forming the address, regardless of the contents of register 0 . The USING domain for this automatic base register assignment is the entire source module.

If a register is specified with base address zero, the assembler will use it in preference to the default use of register zero. For example:

USING 3,0
LA 7,5
generates the instruction \(X^{\prime} 41703005^{\prime}\); in the absence of the USING statement, the generated instruction would be X'41700005'.

For absolute implicit addresses greater than 4095 and in the absence of long-displacement instructions, a USING instruction must be specified according to the following:
- With a base address representing an absolute expression
- With a base register that has not been assigned by a USING instruction in which a relocatable base address is specified

This base register must be loaded with the base address specified.

\section*{Ordinary USING Instruction}

The ordinary USING instruction format specifies a base address and one or more base registers.

sequence_symbol
is a sequence symbol.
base
specifies a base address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and \(2^{31}-1\).
end
specifies the end address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and \(2^{31}-1\). The end address may exceed the (base address + default range) without error. The
end address must be greater than the base and must have the same relocatability attributes.

The resolvable range of a USING with an 'end' operand is
base,MIN(4095, end-1)
Thus USING base, reg is equivalent to USING (base, base+4096), reg.
base_register
is an absolute expression whose value represents general registers 0 through 15.

The default range is 4096 per base register.
The assembler assumes that the base register denoted by the first base_register operand contains the base address base at execution time. If present, the subsequent base_register operands represent registers that the assembler assumes contain the address values base+4096, base+8192, and so forth.

For example:
\[
\text { USING } \quad \text { BASE,9,10,11 }
\]
has the logical equivalent of:
\begin{tabular}{ll} 
USING & BASE,9 \\
USING & BASE \(+4096,10\) \\
USING & BASE \(+8192,11\)
\end{tabular}

In another example, the following statement:
\[
\text { USING } \quad *, 12,13
\]
tells the assembler to assume that the current value of the location counter is in general register 12 at execution time, and that the current value of the location counter, incremented by 4096, is in general register 13 at execution time.

Computing Displacement: If you change the value in a base register being used, and want the assembler to compute displacements from this value, you must tell the assembler the new value by means of another USING statement. In the following sequence, the assembler first assumes that the value of ALPHA is in register 9. The second statement then causes the assembler to assume that ALPHA +1000 is the value in register 9 .
\begin{tabular}{ll} 
USING & ALPHA,9 \\
\(\cdot\) & \\
USING & ALPHA \(+1000,9\)
\end{tabular}

Using General Register Zero: You can refer to the first 4096 bytes of storage using general register 0 , subject to the following conditions:
- The value of operand base must be either absolute or relocatable zero.
- Register 0 must be specified as the first base_register operand.

The assembler assumes that register 0 contains zero. Therefore, regardless of the value of operand base, it calculates displacements as if operand base were absolute or relocatable zero. The assembler also assumes that subsequent registers specified in the same USING statement contain 4096, 8192, etc.

If register 0 is used as a base register, the referenced control section (or dummy section) is not relocatable, despite the fact that operand base may be relocatable. The control section can be made relocatable by:
- Replacing register 0 in the USING statement
- Loading the new register with a relocatable value
- Reassembling the program

\section*{Range of an Ordinary USING Instruction}

The range of an ordinary USING instruction (called the ordinary USING range, or simply the USING range) is the 4096 bytes beginning at the base address specified in the USING instruction, or the range as specified by the range end, whichever is the lesser. For long-displacement instructions, the range is the addresses between (base_address-524288) and (base_address+524287). Addresses that lie within the USING range can be converted from their implicit to their explicit base-displacement form using the designated base registers; those outside the USING range cannot be converted.

The USING range does not depend upon the position of the USING instruction in the source module; rather, it depends upon the location of the base address specified in the USING instruction.

The USING range is the range of addresses in a control section that is associated with the base register specified in the USING instruction. If the USING instruction assigns more than one base register, the composite USING range is the union of the USING ranges that would apply if the base registers were specified in separate USING instructions.

Note that USING ranges need not be contiguous. For example, you could specify
```

USING X,4
USING X+6000,5

```
and implicit addresses with values between \(\mathrm{X}+4096\) and \(\mathrm{X}+5999\) would not be addressable by instructions with unsigned 12 -bit displacements.

Two USING ranges coincide when the same base address is specified in two different USING instructions, even though the base registers used are different. When two USING ranges coincide, the assembler uses the higher-numbered register for assembling the addresses within the common USING range. In effect, the domain of the USING instruction that specifies the lower-numbered register is ended by the other USING instruction. If the domain of the USING instruction that specifies the higher-number register is subsequently terminated, the domain of the other USING instruction is resumed.

Two USING ranges overlap when the base address of one USING instruction lies within the range of another USING instruction. You can use the WARN suboption of the USING assembler option to find out if you have any overlapping USING ranges. When an overlap occurs the assembler issues a diagnostic message. However, the assembler does allow an overlap of one byte in USING ranges so that you don't receive a diagnostic message if you code the following statements:
\begin{tabular}{lll} 
PSTART & CSECT & \\
& LR & R12,R15 \\
& LA & R11,4095(,R12) \\
& USING & PSTART,R12 \\
& USING & PSTART+4095,R11
\end{tabular}

In the above example, the second USING instruction begins the base address of the second base register (R11) in the 4096th byte of the first base register (R12) USING range. If you don't want the USING ranges to overlap, you can code the following statements:
```

PSTART CSECT
LR R12,R15
LA R11,4095(,R12)
LA R11,1(,R11)
USING PSTART,R12
USING PSTART+4096,R11

```

When two ranges overlap, the assembler computes displacements from the base address that gives the smallest non-negative displacement; or if no non-negative displacement can be found, for long-displacement instructions, the base register giving the smallest negative displacement; it uses the corresponding base register when it assembles the addresses within the range overlap. This applies only to implicit addresses that appear after the second USING instruction.

LOCTR does not affect the USING domain.

\section*{Domain of an Ordinary USING Instruction}

The domain of an ordinary USING instruction (called the ordinary USING domain, or simply the USING domain) begins where the USING instruction appears in a source module. It continues until the end of a source module, except when:
- A subsequent DROP instruction specifies the same base register or registers assigned by a preceding USING instruction.
- A subsequent USING instruction specifies the same register or registers assigned by a preceding USING instruction.
The assembler converts implicit address references into their explicit form when the following conditions are met:
- The address reference appears in the domain of a USING instruction.
- The addresses referred to lie within the range of some USING instruction.

The assembler does not convert implicit address references that are outside the USING domain. The USING domain depends on the position of the USING instruction in the source module after conditional assembly, if any, has been done.

\section*{Labeled USING Instruction}

The labeled USING instruction specifies a base address, one or more base registers, and a USING label which can be used as a symbol qualifier.

label
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
base
specifies a base address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and \(2^{31}-1\).
end
specifies the end address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and \(2^{31}-1\). The end address may exceed the (base address + default range) without error. The end address must be greater than the base and must have the same relocatability attributes.
base_register
is an absolute expression whose value represents general registers 0 through 15.

The default range is 4096 per base register.
The essential difference between a labeled USING instruction and an ordinary USING instruction is the label placed on the USING statement. To indicate to the assembler that the USING established with the label is to provide resolution of base and displacement for a symbol, the label must be used to qualify the symbol. Qualifying a symbol consists of preceding the symbol with the label on the USING followed by a period. The only symbols resolved by the labeled USING are those symbols qualified with the label. This label cannot be used for any other purpose in the program, except possibly as a label on other USING instructions.

The following examples show how labeled USINGs are used:
```

PRIOR USING IHADCB,R10
NEXT USING IHADCB,R2
mVC PRIOR.DCBLRECL,NEXT.DCBLRECL

```

The same code without labeled USINGs could be written like this:
USING IHADCB,R10
MVC DCBLRECL,DCBLRECL-IHADCB(R2)
In the following example, a new element, NEW, is inserted into a doubly-linked list between two existing elements LEFT and RIGHT, where the links are stored as pointers LPTR and RPTR:
\begin{tabular}{llll} 
LEFT & USING & ELEMENT,R3 & \\
RIGHT & USING & ELEMENT,R6 & \\
NEW & USING & ELEMENT,R1 & \\
& - & & \\
& MVC & NEW.RPTR,LEFT.RPTR & Move previous Right pointer \\
& MVC & NEW.LPTR,RIGHT.LPTR & Move previous Left pointer \\
& ST & R1,LEFT.RPTR & Chain new element from Left \\
& ST & R1,RIGHT.LPTR & Chain new element from Right \\
& - & & \\
ELEMENT & DSECT & & Link to left element \\
LPTR & DS & A & Link to right element \\
RPTR & DS & A &
\end{tabular}

\section*{Range of a Labeled USING Instruction}

The range of a labeled USING instruction (called the labeled USING range) is the 4096 bytes beginning at the base address specified in the labeled USING instruction, or the range as specified by the range end, whichever is the lesser. Addresses that lie within the labeled USING range can be converted from their implicit form (qualified symbols) to their explicit form; those outside the USING range cannot be converted.

Like the ordinary USING range, the labeled USING range is the range of addresses in a control section that is associated with the base register specified in the labeled USING instruction. If the labeled USING instruction assigns more than one base register, the composite labeled USING range is the product of the number of registers specified in the labeled USING instruction and 4096 bytes. The composite labeled USING range begins at the base address specified in the labeled USING instruction. Unlike the ordinary USING range, however, you cannot specify separate labeled USING instructions to establish the same labeled USING range. For example,
IN USING BASE,10,11
specifies a range of 8192 bytes beginning at BASE, but
\(\begin{array}{ll}\text { IN } & \text { USING BASE, } 10 \\ \text { IN } & \text { USING BASE }+4096,11\end{array}\)
specifies a single labeled USING range of 4096 bytes beginning at BASE+4096.
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, High Level Assembler does not treat the USING ranges as coinciding. When you specify an unqualified symbol in an assembler instruction, the base register specified in the ordinary USING is used by the assembler to resolve the address into base-displacement form. An example of coexistent ordinary USINGs and labeled USINGs is given below:
\begin{tabular}{lll} 
& USING & IHADCB,R10 \\
SAMPLE & USING & IHADCB,R2 \\
& MVC & DCBLRECL,SAMPLE.DCBLRECL
\end{tabular}

In this MVC instruction, the (unqualified) first operand is resolved with the ordinary USING, and the (qualified) second operand is resolved with the labeled USING.

\section*{Domain of a Labeled USING Instruction}

The domain of a labeled USING instruction (called the labeled USING domain) begins where the USING instruction appears in a source module. It continues to the end of the source module, except when:
- A subsequent DROP instruction specifies the label used in the preceding labeled USING instruction.
- A subsequent USING instruction specifies the same label used in the preceding labeled USING instruction. The second specification of the label causes the assembler to end the domain of the prior USING with the same label.

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 eligible to be used as symbol 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 assembler converts implicit address references into their explicit form using the base register or registers specified in a labeled USING instruction when the following conditions are met:
- The address reference appears in the domain of the labeled USING instruction.
- The address reference takes the form of a qualified symbol and the qualifier is the label of the labeled USING instruction.
- The address lies within the range of the labeled USING instruction.

\section*{Dependent USING Instruction}

The dependent USING instruction format specifies a base address and a relocatable expression instead of one or more base registers. If a USING label is also specified, this format USING instruction is called a labeled dependent USING instruction.

label
is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

\section*{sequence_symbol}
is a sequence symbol.
base
specifies a base address, which must be a relocatable expression. The value of the expression must lie between 0 and \(2^{31}-1\).
address
is a simply relocatable expression that represents an implicit address within the range of an active USING instruction. The range of an active USING is considered to be that which is valid for generating 12-bit or 20-bit displacements.
end
specifies the end address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and \(2^{31}-1\). The end address may exceed the (base address + default range) without error. The end address must be greater than the base and must have the same relocatability attributes.

The implicit address denoted by address specifies the address where base is to be based, and is known as the supporting base address. As address is a relocatable expression, it distinguishes a dependent USING from an ordinary USING. The assembler converts the implicit address denoted by address into its explicit base-displacement form. It then assigns the base register from this explicit address as the base register for base. The assembler assumes that the base register contains the base address base minus the displacement determined in the explicit address. The assembler also assumes that address is appropriately aligned for the code based on base. Warnings are not issued for potential alignment problems in the dependent USING address.

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

The following example shows the use of an unlabeled dependent USING:
EXAMPLE CSECT
USING EXAMPLE,R10,R11 Ordinary USING
-
USING IHADCB,DCBUT2 Unlabeled dependent USING
LH R0,DCBBLKSI Uses R10 or R11 for BASE

DCBUT2 DCB DDNAME=SYSUT2,...

The following example shows the use of two labeled dependent USINGs:
\begin{tabular}{llll} 
EXAMPLE & CSECT & & \\
& USING & EXAMPLE,R10,R11 & Ordinary USING \\
& \(\cdot\) & & \\
DCB1 & USING & IHADCB,DCBUT1 & Labeled dependent USING \\
DCB2 & USING & IHADCB,DCBUT2 & Labeled dependent USING \\
& MVC & DCB2.DCBBLKSI,DCB1.DCBBLKSI & Uses R10 or R11 for BASE \\
& \(\cdot\) & & \\
DCBUT1 & DCB & DDNAME=SYSUT1,... & \\
DCBUT2 & DCB & DDNAME=SYSUT2,... &
\end{tabular}

\section*{Range of a Dependent USING Instruction}

The range of a dependent USING instruction (called the dependent USING range) is either the range as specified by the range end, or the range of the corresponding USING minus the offset of address within that range, whichever is the lesser. If the corresponding labeled or ordinary USING assigns more than one base register, the maximum dependent USING range is the composite USING range of the labeled or ordinary 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.

\section*{Domain of a Dependent USING Instruction}

The domain of a dependent USING instruction (called the dependent USING domain) begins where the dependent USING 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.
- You end the domain of a labeled dependent USING by specifying the label of the labeled dependent USING in the operand of a subsequent labeled USING instruction.

When a labeled dependent USING domain is dropped, none of any subordinate USING domains are dropped. In the following example the labeled dependent USING BLBL1 is not dropped, even though it appears to be dependent on the USING ALBL2 that is being dropped:
\begin{tabular}{lll} 
ALBL1 & \begin{tabular}{l} 
USING \\
\\
\\
\\
\\
\\
\(\cdot\) \\
USING
\end{tabular} & \begin{tabular}{l} 
DSECTA, 14 \\
DSECTB,ALBL1.A
\end{tabular} \\
ALBL2 & USING & \\
BLBL1 & USING & DSECTA,ALBL1.A \\
& - & \\
& DROP & DSECTA,ALBL2.A \\
DSECTA & DSECT & ALBL2 \\
A & DS & \\
DSECTB & DSECT & A \\
B & DS & \\
& & A
\end{tabular}

A dependent USING is not dependent on another dependent USING. It is dependent on the ordinary or labeled USING that is finally used to resolve the address. For example, the USING at BLBL1 is dependent on the ALBL1 USING.

Remember that all dependent USINGs must eventually be based on an ordinary or labeled USING that provides the base register used for base-displacement resolutions.

\section*{WXTRN Instruction}

The WXTRN statement identifies "weak external" symbols referred to in a source module but defined in another source module. The WXTRN instruction differs from the EXTRN instruction (see "EXTRN Instruction" on page 189) as follows:
- The EXTRN instruction causes the linker to automatically search libraries (if automatic library call is in effect) to find the module that contains the external symbols that you identify in its operand field. If the module is found, linkage addresses are resolved; the module is then linked to your module, which contains the EXTRN instruction.
- The WXTRN instruction suppresses automatic search of libraries. The linker only resolves the linkage addresses if the external symbols that you identify in the WXTRN operand field are defined:
- In a module that is linked and loaded along with the object module assembled from your source module, or
- In a module brought in from a library because of the presence of an EXTRN instruction in another module linked and loaded with yours.

sequence_symbol
is a sequence symbol.

\section*{XATTR Instruction}

\section*{external_symbol}
is a relocatable symbol that is not:
- Used as the name entry of a source statement in the source module in which it is defined

The external symbols identified by a WXTRN instruction have the same properties as the external symbols identified by the EXTRN instruction. However, the type code assigned to these external symbols differs.

V-Type Address Constant: If a symbol, specified in a V-type address constant, is also identified by a WXTRN instruction, it is assigned the same ESD type code as the symbol in the WXTRN instruction, and is treated by the linkage editor as a weak external symbol.

If an external symbol is identified by both an EXTRN and WXTRN instruction in the same source module, the first declaration takes precedence, and subsequent declarations are flagged with diagnostic messages.

\section*{XATTR Instruction (MVS and CMS)}

The XATTR instruction enables attributes to be assigned to an external symbol. This instruction is only valid when you specify the GOFF assembler option.


\section*{symbol}
is a symbol which has been declared implicitly or explicitly as an external symbol. Further, if the PSECT attribute is specified, must be a RSECT, CSECT or START name or an ENTRY name (where the entry is in one of the preceding types of section)

\section*{attribute}
is one or more attributes from the group of attributes described below. The assembler sets the appropriate attribute flags in the GOFF External Symbol Directory record.

Notes:
1. If more than one value is specified for a given attribute, no diagnostic is issued and only the last value is used.
2. All attributes of an external symbol must be specified in a single XATTR statement (which may be continued).

\section*{ATTRIBUTES}
--ATTRIBUTES(label)

\section*{ATTRIBUTES(label), abbreviation ATTR(label)}
is a symbol (internal or external) known in the declaring program. It names the location of the extended attribute information to be associated with symbol.

Instructs the assembler to place the ESDID and offset of the label in the GOFF External Symbol Dictionary record.

\section*{LINKAGE}
\(\leadsto-\operatorname{LINKAGE}\left(\square_{\text {XPLINK_ }}^{\mathrm{OS}}\right) \longrightarrow\) -

\section*{LINKAGE(OS), abbreviation LINK(OS)}

Instructs the assembler to set the "Linkage Type" attribute to standard OS linkage.

\section*{LINKAGE(XPLINK), abbreviation LINK(XPLINK)}

Instructs the assembler to set the "Linkage Type" attribute to indicate "Extra Performance Linkage."

PSECT
\(»\)-PSECT (name) \(\qquad\)

\section*{PSECT (name)}

Identifies the private read-write "section" or PSECT associated with name by its being an internal or external symbol belonging to an element in the class to which the PSECT belongs. The name is one of:
- an ENTRY name, where the entry is in the same section (CSECT or RSECT) as name, but in a different class. For reentrant code, the PSECT is normally a non-shared class, so a separate CATTR statement is needed to declare that class and its attributes.
- an internal label within the PSECT.


\section*{REFERENCE(DIRECT), abbreviation REF(DIRECT)}

Instructs the assembler to reset (clear) the "Indirect Reference" attribute.

\section*{REFERENCE(INDIRECT), abbreviation REF(INDIRECT)}

Instructs the assembler to assign the "Indirect Reference" attribute.

\section*{REFERENCE(CODE), abbreviation REF(CODE)}

Instructs the assembler to set the Executable attribute.

\section*{REFERENCE(DATA), abbreviation REF(DATA)}

Instructs the assembler to set the Not Executable attribute.

\section*{SCOPE}


\section*{SCOPE(SECTION), abbreviation SCOPE(S)}

Instructs the assembler to set the binding scope to "Section."

\section*{SCOPE(MODULE), abbreviation SCOPE(M)}

Instructs the assembler to set the binding scope to "Module."

\section*{SCOPE(LIBRARY), abbreviation SCOPE(L)}

Instructs the assembler to set the binding scope to "Library."

\section*{Association of Code and Data Areas (MVS and CMS)}

To provide support for application program reentrancy and dynamic binding, the assembler provides a way to associate read-only code and read-write data areas. This is done by defining and accessing "associated data areas" called PSECTs. A PSECT (Private or Prototype Control Section) when instantiated becomes the non-shared working storage for an invocation of a shared reentrant program.

In the Program Object model, a PSECT is an element within the same section as the element containing the shared code to which it belongs. The two classes defining these elements will have attributes appropriate to their respective uses.

Typically, V-type and R-type address constants are used to provide code and data-area addressability for a reentrant program using PSECTs.

Figure 61 on page 233 shows an example of two sections \(A\) and \(B\), each with a PSECT. When the program object \(A B\) containing \(A\) and \(B\) is instantiated, a single copy of the reentrant CODE class is loaded into read-only storage, and a copy of the PSECT class belonging to \(A B\) is loaded into read-write storage. The invoker of A provides the address of A's PSECT so that A can address its own read-write data. A later instantiation of \(A B\) would load only a new copy of the PSECT class.

Figure 61. Program Object with PSECTs, Example 1
When a program in the CODE class of section A calls a program in the CODE class of section B , a linkage convention might require loading the entry address of B into general register 15 and the address of B's PSECT into general register 0. For example:
\begin{tabular}{lll}
\(L\) & \(15,=V(B)\) & B's entry point address \\
\(L\) & \(0,=R(B)\) & \(B^{\prime} s\) PSECT address (from A's PSECT) \\
BASR & 14,15 & Linkage to \(B\)
\end{tabular}

Further information about linkage conventions for referencing Dynamic Link Libraries (DLLs) under the Language Environment can be found in z/OS V1R6.0 Language Environment Programming Guide (SA22-7561).

\section*{XATTR Instruction}

\section*{Part 3. Macro Language}
Chapter 6. Introduction to Macro Language ..... 238
Using Macros ..... 238
Macro Definition ..... 238
Model Statements ..... 239
Processing Statements ..... 240
Comment Statements ..... 240
Macro Instruction ..... 241
Source and Library Macro Definitions ..... 241
Macro Library ..... 242
System Macro Instructions ..... 242
Conditional Assembly Language ..... 242
Chapter 7. How to Specify Macro Definitions ..... 243
Where to Define a Macro in a Source Module ..... 243
Format of a Macro Definition ..... 244
Macro Definition Header and Trailer ..... 244
MACRO Statement ..... 244
MEND Statement ..... 245
Macro Instruction Prototype ..... 245
Body of a Macro Definition ..... 247
Model Statements ..... 247
Variable Symbols as Points of Substitution ..... 248
Listing of Generated Fields ..... 248
Rules for Concatenation ..... 249
Rules for Model Statement Fields ..... 251
Symbolic Parameters ..... 253
Positional Parameters ..... 254
Keyword Parameters ..... 255
Combining Positional and Keyword Parameters ..... 255
Subscripted Symbolic Parameters ..... 255
Processing Statements ..... 256
Conditional Assembly Instructions ..... 256
Inner Macro Instructions ..... 256
Other Conditional Assembly Instructions ..... 256
AEJECT Instruction ..... 257
AINSERT Instruction ..... 257
AREAD Instruction ..... 257
ASPACE Instruction ..... 259
COPY Instruction ..... 260
MEXIT Instruction ..... 260
Comment Statements ..... 261
Ordinary Comment Statements ..... 261
Internal Macro Comment Statements ..... 261
System Variable Symbols ..... 262
Scope and Variability of System Variable Symbols ..... 262
\&SYSADATA_DSN System Variable Symbol ..... 263
\&SYSADATA_MEMBER System Variable Symbol ..... 264
\&SYSADATA_VOLUME System Variable Symbol ..... 265
\&SYSASM System Variable Symbol ..... 265
\&SYSCLOCK System Variable Symbol ..... 266

\section*{Part 3. Macro Language}
\&SYSDATC System Variable Symbol ..... 266
\&SYSDATE System Variable Symbol ..... 267
\&SYSECT System Variable Symbol ..... 267
\&SYSIN_DSN System Variable Symbol ..... 269
\&SYSIN_MEMBER System Variable Symbol ..... 270
\&SYSIN_VOLUME System Variable Symbol ..... 271
\&SYSJOB System Variable Symbol ..... 272
\&SYSLIB_DSN System Variable Symbol ..... 272
\&SYSLIB_MEMBER System Variable Symbol ..... 273
\&SYSLIB_VOLUME System Variable Symbol ..... 273
\&SYSLIN_DSN System Variable Symbol ..... 274
\&SYSLIN_MEMBER System Variable Symbol ..... 275
\&SYSLIN_VOLUME System Variable Symbol ..... 275
\&SYSLIST System Variable Symbol ..... 276
\&SYSLOC System Variable Symbol ..... 278
\&SYSMAC System Variable Symbol ..... 279
\&SYSM_HSEV System Variable Symbol ..... 279
\&SYSM_SEV System Variable Symbol ..... 280
\&SYSNDX System Variable Symbol ..... 281
\&SYSNEST System Variable Symbol ..... 283
\&SYSOPT_DBCS System Variable Symbol ..... 284
\&SYSOPT_OPTABLE System Variable Symbol ..... 285
\&SYSOPT_RENT System Variable Symbol ..... 285
\&SYSOPT_XOBJECT System Variable Symbol ..... 285
\&SYSPARM System Variable Symbol ..... 286
\&SYSPRINT_DSN System Variable Symbol ..... 286
\&SYSPRINT_MEMBER System Variable Symbol ..... 288
\&SYSPRINT_VOLUME System Variable Symbol ..... 288
\&SYSPUNCH_DSN System Variable Symbol ..... 289
\&SYSPUNCH_MEMBER System Variable Symbol ..... 290
\&SYSPUNCH_VOLUME System Variable Symbol ..... 290
\&SYSSEQF System Variable Symbol ..... 291
\&SYSSTEP System Variable Symbol ..... 292
\&SYSSTMT System Variable Symbol ..... 292
\&SYSSTYP System Variable Symbol ..... 292
\&SYSTEM_ID System Variable Symbol ..... 293
\&SYSTERM_DSN System Variable Symbol ..... 294
\&SYSTERM_MEMBER System Variable Symbol ..... 295
\&SYSTERM_VOLUME System Variable Symbol ..... 295
\&SYSTIME System Variable Symbol ..... 296
\&SYSVER System Variable Symbol ..... 296
Chapter 8. How to Write Macro Instructions ..... 297
Macro Instruction Format ..... 297
Alternative Formats for a Macro Instruction ..... 298
Name Entry ..... 299
Operation Entry ..... 299
Operand Entry ..... 300
Sublists in Operands ..... 304
Values in Operands ..... 307
Omitted Operands ..... 307
Unquoted Operands ..... 308
Special Characters ..... 308
Nesting Macro Instruction Definitions ..... 311
Inner and Outer Macro Instructions ..... 312
Levels of Macro Call Nesting ..... 312
General Rules and Restrictions ..... 313
Passing Values through Nesting Levels ..... 314
System Variable Symbols in Nested Macros ..... 316
Chapter 9. How to Write Conditional Assembly Instructions ..... 318
SET Symbols ..... 319
Subscripted SET Symbols ..... 319
Scope of SET Symbols ..... 319
Scope of Symbolic Parameters ..... 319
SET Symbol Specifications ..... 320
Subscripted SET Symbol Specification ..... 322
Created SET Symbols ..... 323
Data Attributes ..... 324
Attributes of Symbols and Expressions ..... 326
Type Attribute ( \(\mathrm{T}^{\prime}\) ) ..... 328
Length Attribute (L') ..... 332
Scale Attribute (S') ..... 333
Integer Attribute (I') ..... 334
Count Attribute ( \(\mathrm{K}^{\prime}\) ) ..... 335
Number Attribute ( \(\mathrm{N}^{\prime}\) ) ..... 336
Defined Attribute (D') ..... 337
Operation Code Attribute ( \(\mathrm{O}^{\prime}\) ) ..... 337
Sequence Symbols ..... 339
Lookahead ..... 340
Open Code ..... 342
Conditional Assembly Instructions ..... 343
Declaring SET Symbols ..... 343
GBLA, GBLB, and GBLC Instructions ..... 344
LCLA, LCLB, and LCLC Instructions ..... 345
Assigning Values to SET Symbols ..... 347
SETA Instruction ..... 347
SETB Instruction ..... 362
SETC Instruction ..... 369
Extended SET Statements ..... 387
SETAF Instruction ..... 388
SETCF Instruction ..... 389
Branching ..... 390
AIF Instruction ..... 390
AGO Instruction ..... 392
ACTR Instruction ..... 394
ANOP Instruction ..... 395
Chapter 10. MHELP Instruction ..... 397

\section*{Chapter 6. Introduction to Macro Language}

This chapter introduces the basic macro concept: what you can use the macro facility for, how you can prepare your own macro definitions, and how you call these macro definitions for processing by the assembler.

Macro language is an extension of assembler language. It provides a convenient way to generate a sequence of assembler language statements many times in one or more programs. A macro definition is written only once; thereafter, a single statement, a macro instruction statement, is written each time you want to generate the sequence of statements. This simplifies the coding of programs, reduces the chance of programming errors, and ensures that standard sequences of statements are used to accomplish the functions you want.

In addition, conditional assembly lets you code statements that may or may not be assembled, depending upon conditions evaluated at conditional assembly time. These conditions are usually tests of values which may be defined, set, changed, and tested during assembly. Conditional assembly statements can be used within macro definitions or in open code.

\section*{Using Macros}

The main use of macros is to insert assembler language statements into a source program.

You call a named sequence of statements (the macro definition) by using a macro instruction, or macro call. The assembler replaces the macro call by the statements from the macro definition and inserts them into the source module at the point of call. The process of inserting the text of the macro definition is called macro generation or macro expansion. Macro generation occurs during conditional assembly.

The expanded stream of code then becomes the input for processing at assembly time; that is, the time at which the assembler translates the machine instructions into object code.

\section*{Macro Definition}

A macro definition is a named sequence of statements you can call with a macro instruction. When it is called, the assembler processes and usually generates assembler language statements from the definition into the source module. The statements generated can be:
- Copied directly from the definition
- Modified by parameter values and other values in variable symbols before generation
- Manipulated by internal macro processing to change the sequence in which they are generated

You can define your own macro definitions in which any combination of these three processes can occur. Some macro definitions, like some of those used for system
generation, do not generate assembler language statements, but do only internal processing.

A macro definition provides the assembler with:
- The name of the macro
- The parameters used in the macro
- The sequence of statements the assembler generates when the macro instruction appears in the source program.

Every macro definition consists of a macro definition header statement (MACRO), a macro instruction prototype statement, one or more assembler language statements, and a macro definition trailer statement (MEND), as shown in Figure 62.


Figure 62. Parts of a Macro Definition
- The macro definition header and trailer statements (MACRO and MEND) indicate to the assembler the beginning and end of a macro definition (see 1 in Figure 62).
- The macro instruction prototype statement names the macro (see 2 in Figure 62), and declares its parameters (see 3 in Figure 62). In the operand field of the macro instruction, you can assign values (see 4 in Figure 62) to the parameters declared for the called macro definition.
- The body of a macro definition (see 5 in Figure 62 ) contains the statements that are generated when you call the macro. These statements are called model statements; they are usually interspersed with conditional assembly statements or other processing statements.

\section*{Model Statements}

You can write machine instruction statements and assembler instruction statements as model statements. During macro generation, the assembler copies them exactly as they are written. You can also use variable symbols as points of substitution in a model statement. The assembler enters values in place of these points of substitution each time the macro is called.

The three types of variable symbols in the assembler language are:
- Symbolic parameters, declared in the prototype statement
- System variable symbols
- SET symbols, which are part of the conditional assembly language

The assembler processes the generated statements, with or without value substitution, at assembly time.

\section*{Processing Statements}

Processing statements are processed during conditional assembly, when macros are expanded, but they are not themselves generated for further processing at assembly time. The processing statements are:
- AEJECT instructions
- AREAD instructions
- ASPACE instructions
- Conditional assembly instructions
- Inner macro calls
- MEXIT instructions
- MNOTE instructions

The AEJECT and ASPACE instructions let you control the listing of your macro definition. Use the AEJECT instruction to stop printing the listing on the current page and continue printing on the next. Use the ASPACE instruction to insert blank lines in the listing. The AEJECT instruction is described on page 257 The ASPACE instruction is described on page 259

The AREAD instruction assigns a character string value, of a statement that is placed immediately after a macro instruction, to a SETC symbol. The AREAD instruction is described on page 257

Conditional assembly instructions, inner macro calls, and macro processing instructions are described in detail in the following chapters.

The MNOTE instruction generates an error message with an error condition code attached, or generates comments in which you can display the results of a conditional assembly computation. The MNOTE instruction is described on page 196

The MEND statement delimits the contents of a macro definition, and also provides an exit from the definition. The MEND instruction is described on page 245

The MEXIT instruction tells the assembler to stop processing a macro definition, and provides an exit from the macro definition at a point before the MEND statement. The MEXIT instruction is described on page 260

\section*{Comment Statements}

One type of comment statement describes conditional assembly operations and is not generated. The other type describes assembly-time operations and is, therefore, generated. For a description of the two types of comment statements, see "Comment Statements" on page 261.

\section*{Macro Instruction}

A macro instruction is a source program statement that you code to tell the assembler to process a particular macro definition. The assembler generates a sequence of assembler language statements for each occurrence of the same macro instruction. The generated statements are then processed as any other assembler language statement.

The macro instruction provides the assembler with:
- The name of the macro definition to be processed.
- The information or values to be passed to the macro definition. The assembler uses the information either in processing the macro definition or for substituting values into a model statement in the definition.

The output from a macro definition, called by a macro instruction, can be:
- A sequence of statements generated from the model statements of the macro for further processing at assembly time.
- Values assigned to global SET symbols. These values can be used in other macro definitions and in open code.

You can call a macro definition by specifying a macro instruction anywhere in a source module. You can also call a macro definition from within another macro definition. This type of call is an inner macro call; it is said to be nested in the macro definition.

\section*{Source and Library Macro Definitions}

You can include a macro definition in a source module. This type of definition is called a source macro definition, or, sometimes, an in-line macro definition.

You can also insert a macro definition into a system or user library by using the applicable utility program. This type of definition is called a library macro definition. The IBM-supplied macro definitions are examples of library macro definitions.

You can call a source macro definition only from the source module in which it is included. You can call a library macro definition from any source module if the library containing the macro definition is available to the assembler.

Syntax errors in processing statements are handled differently for source macro definitions and library macro definitions. In source macro definitions, error messages are listed following the statements in error. In library macros, however, error messages cannot be associated with the statement in error, because the statements in library macro definitions are not included in the assembly listing. Therefore, the error messages are listed directly following the first call of that macro.

Because of the difficulty of finding syntax errors in library macros, a macro definition should be run and "debugged" as a source macro before it is placed in a macro library. Alternatively, you can use the LIBMAC assembler option to have the assembler automatically include the source statements of the library macro in your source module. For more information about the LIBMAC option, see the HLASM Programmer's Guide.

\section*{Macro Library}

The same macro definition may be made available to more than one source program by placing the macro definition in the macro library. The macro library is a collection of macro definitions that can be used by all the assembler language programs in an installation. When a macro definition has been placed in the macro library, it can be called by coding its corresponding macro instruction in a source program. Macro definitions must be in a macro library with a member name that is the same as the macro name. The procedure for placing macro definitions in the macro library is described in the applicable utilities manual.

The DOS/VSE assembler requires library macro definitions to be placed in the macro library in a special edited format. High Level Assembler does not require this. Library macro definitions must be placed in the macro library in source statement format. If you wish to use edited macros in VSE you can provide a LIBRARY exit to read the edited macros and convert them into source statement format. A library exit is supplied with VSE and is described in VSE/ESA Guide to System Functions.

\section*{System Macro Instructions}

The macro instructions that correspond to macro definitions prepared by IBM are called system macro instructions. System macro instructions are described in the applicable operating system manuals that describe macro instructions for supervisor services and data management.

\section*{Conditional Assembly Language}

The conditional assembly language is a programming language with most of the features that characterize a programming language. For example, it provides:
- Variables
- Data attributes
- Expression computation
- Assignment instructions
- Labels for branching
- Branching instructions
- Substring operators that select characters from a string

Use the conditional assembly language in a macro definition to receive input from a calling macro instruction. You can produce output from the conditional assembly language by using the MNOTE instruction.

Use the functions of the conditional assembly language to select statements for generation, to determine their order of generation, and to do computations that affect the content of the generated statements.

The conditional assembly language is described in Chapter 9, "How to Write Conditional Assembly Instructions" on page 318.

\section*{Chapter 7. How to Specify Macro Definitions}

A macro definition is a set of statements that defines the name, the format, and the conditions for generating a sequence of assembler language statements. The macro definition can then be called by a macro instruction to process the statements. See page 241 for a description of the macro instruction. To define a macro you must:
- Give it a name
- Declare any parameters to be used
- Write the statements it contains
- Establish its boundaries with a macro definition header statement (MACRO) and a macro definition trailer statement (MEND)

Except for conditional assembly instructions, this chapter describes all the statements that can be used to specify macro definitions. Conditional assembly instructions are described in Chapter 9, "How to Write Conditional Assembly Instructions" on page 318

\section*{Where to Define a Macro in a Source Module}

Macro definitions can appear anywhere in a source module. They remain in effect for the rest of your source module, or until another macro definition defining a macro with the same operation code is encountered, or until an OPSYN statement deletes its definition. Thus, you can redefine a macro at any point in your program. The new definition is used for all subsequent calls to the macro in the program.

This type of macro definition is called a source macro definition, or, sometimes, an in-line macro definition. A macro definition can also reside in a system library; this type of macro is called a library macro definition. Either type can be called from the source module by the applicable macro instruction.

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

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

\section*{Example:}
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & \multirow{5}{*}{\[
\begin{aligned}
& \& A, \& C= \\
& \left(' \& C^{\prime} E Q \quad '\right) \cdot A
\end{aligned}
\]} & Macro header for outer macro \\
\hline & OUTER & & Macro prototype \\
\hline & AIF & & \\
\hline & MACRO & & Macro header for inner macro \\
\hline & INNER & & Macro prototype \\
\hline & - & & \\
\hline \multirow{4}{*}{.A} & MEND & & Macro trailer for inner macro \\
\hline & ANOP & & \\
\hline & & & \\
\hline & MEND & & Macro trailer for outer macro \\
\hline
\end{tabular}

\section*{MACRO and MEND Statements}

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

Open Code: Open code is that part of a source module that lies outside of any source macro definition. At coding time, it is important to distinguish between source statements that lie in open code, and those that lie inside macro definitions.

\section*{Format of a Macro Definition}

The general format of a macro definition is shown in Figure 63. The four parts are described in detail in the following sections.


Figure 63. Format of a Macro Definition

\section*{Macro Definition Header and Trailer}

You must establish the boundaries of a macro definition by coding:
- A macro definition header statement as the first statement of the macro definition (a MACRO statement)
- A macro definition trailer statement as the last statement of the macro definition (a MEND statement)

The instructions used to define the boundaries of a macro instruction are described in the following sections.

\section*{MACRO Statement}

Use the MACRO statement to indicate the beginning of a macro definition. It must be the first non-comment statement in every macro definition. Library macro definitions may have ordinary or internal macro comments before the MACRO statement.


The MACRO statement must not have a name entry or an operand entry.

\section*{MEND Statement}

Use the MEND statement to indicate the end of a macro definition. It also provides an exit when it is processed during macro expansion. It can appear only once within a macro definition and must be the last statement in every macro definition.

sequence_symbol
is a sequence symbol.
See "MEXIT Instruction" on page 260 for details on exiting from a macro before the MEND statement.

\section*{Macro Instruction Prototype}

The macro instruction prototype statement (hereafter called the prototype statement) specifies the mnemonic operation code and the format of all macro instructions that you use to call the macro definition.

The prototype statement must be the second non-comment statement in every macro definition. Both ordinary comment statements and internal comment statements are allowed between the macro definition header and the macro prototype. Such comment statements are listed only with the macro definition.

name_field
is a variable symbol.
You can write a name field parameter, similar to the symbolic parameter, as the name entry of a macro prototype statement. You can then assign a value to this parameter from the name entry in the calling macro instruction.
If this parameter also appears in the body of a macro, it is given the value assigned to the parameter in the name field of the corresponding macro instruction.

\section*{operation_field}
is an ordinary symbol.
The symbol in the operation field of the prototype statement establishes the name by which a macro definition must be called. This name becomes the operation code required in any macro instruction that calls the macro.
Any operation code can be specified in the prototype operation field. If the entry is the same as an assembler or a machine operation code, the new definition overrides the previous use of the symbol. The same is true if the specified operation code has been defined earlier in the program as a macro, or is the operation code of a library macro.

Macros that are defined inline may use any ordinary symbol, up to 63 characters in length, for the operation field. However, operating system rules may prevent some of these macros from being stored as member names.

The assembler requires that the library member name and macro name be the same; otherwise error diagnostic message ASMA126S Library macro name incorrect is issued.
symbolic_parameter
The symbolic parameters are used in the macro definition to represent the operands of the corresponding macro instruction. A description of symbolic parameters appears under "Symbolic Parameters" on page 253 .

The operand field in a prototype statement lets you specify positional or keyword parameters. These parameters represent the values you can pass from the calling macro instruction to the statements within the body of a macro definition.

The operand field of the macro prototype statement must contain 0 to 32000 symbolic parameters separated by commas. They can be positional parameters or keyword parameters, or both.

If no parameters are specified in the operand field and if the absence of the operand entry is indicated by a comma preceded and followed by one or more spaces, remarks are allowed.

The following is an example of a prototype statement:
\&NAME MOVE \&TO,\&FROM

\section*{Alternative Formats for the Prototype Statement}

The prototype statement can be specified in one of the following three ways:
- The normal way, with all the symbolic parameters preceding any remarks
- An alternative way, allowing remarks for each parameter
- A combination of the first two ways

The continuation rules for macro instructions are different from those for machine or assembler instruction statements. This difference is important for those who write macros that override a machine/assembler mnemonic.

The following examples show the normal statement format (\&NAME1), the alternative statement format (\&NAME2), and a combination of both statement formats (\&NAME3):
\begin{tabular}{lll} 
& \begin{tabular}{l} 
Opera- \\
tion
\end{tabular} Operand & Comment
\end{tabular}\(\quad\) Cont.

\section*{Notes:}
1. Any number of continuation lines is allowed. However, each continuation line must be indicated by a non-space character in the column after the end column on the preceding line.
2. For each continuation line, the operand field entries (symbolic parameters) must begin in the continue column; otherwise, the whole line and any lines that follow are considered to contain remarks.
No error diagnostic message is issued to indicate that operands are treated as remarks in this situation. However, the FLAG(CONT) assembler option can be specified so that the assembler issues warning messages if it suspects an error in a continuation line.
3. The standard value for the continue column is 16 and the standard value for the end column is 71 .
4. A comma is required after each parameter except the last. If you code excess commas between parameters, they are considered null positional parameters. No error diagnostic message is issued.
5. One or more spaces is required between the operand and the remarks.
6. If the DBCS assembler option is specified, the continuation features outlined in "Continuation of double-byte data" on page 16 apply to continuation in the macro language. Extended continuation may be useful if a macro keyword parameter contains double-byte data.

\section*{Body of a Macro Definition}

The body of a macro definition contains the sequence of statements that constitutes the working part of a macro. You can specify:
- Model statements to be generated
- Processing statements that, for example, can alter the content and sequence of the statements generated or issue error messages
- Comment statements, some that are generated and others that are not
- Conditional assembly instructions to compute results to be displayed in the message created by the MNOTE instruction, without causing any assembler language statements to be generated

The statements in the body of a macro definition must appear between the macro prototype statement and the MEND statement of the definition. The body of a macro definition can be empty, that is, contain no statements.

Nesting Macros: You can include macro definitions in the body of a macro definition.

\section*{Model Statements}

Model statements are statements from which assembler language statements are generated during conditional assembly. They let you determine the form of the statements to be generated. By specifying variable symbols as points of substitution in a model statement, you can vary the contents of the statements
generated from that model statement. You can also substitute values into model statements in open code.

A model statement consists of one or more fields, separated by one or more spaces, in columns 1 to 71 . The fields are called the name, operation, operand, and remarks fields.

Each field or subfield can consist of:
- An ordinary character string composed of alphanumeric and special characters
- A variable symbol as a point of substitution, except in remarks fields and comment statements
- Any combination of ordinary character strings and variable symbols to form a concatenated string

The statements generated from model statements during conditional assembly must be valid machine or assembler instructions, but must not be conditional assembly instructions. They must follow the coding rules described in "Rules for Model Statement Fields" on page 251 or they are flagged as errors at assembly time.

\section*{Examples:}
\begin{tabular}{lll} 
LABEL & L & 3,AREA \\
LABEL2 & L & \(3,20(4,5)\) \\
\&LABEL & L & \(3, \& A R E A\) \\
FIELD\&A & L & 3, AREA\&C
\end{tabular}

\section*{Variable Symbols as Points of Substitution}

Values can be substituted for variable symbols that appear in the name, operation, and operand fields of model statements; thus, variable symbols represent points of substitution. The three main types of variable symbol are:
- Symbolic parameters (positional or keyword)
- System variable symbols (see "System Variable Symbols" on page 262)
- SET symbols (global-scope or local-scope SETA, SETB, or SETC symbols)

\section*{Examples:}
\&PARAM (3)
\&SYSLIST \((1,3)\)
\&SYSLIST(2)
\&SETA(10)
\&SETC(15)
Symbols That Can Be Subscripted: Symbolic parameters, SET symbols, and the system variable symbols \&SYSLIST and \&SYSMAC, can all be subscripted. All remaining system variable symbols contain only one value.

\section*{Listing of Generated Fields}

The different fields in a macro-generated statement or a statement generated in open code appear in the listing in the same column as they are coded in the model statement, with the following exceptions:
- If the substituted value in the name or operation field is too large for the space available, the next field is moved to the right with one space separating the fields.
- If the substituted value in the operand field causes the remarks field to be displaced, the remarks field is written on the next line, starting in the column where it is coded in the model statement.
- If the substituted value in the operation field of a macro-generated statement contains leading spaces, the spaces are ignored.
- If the substituted value in the operation field of a model statement in open code contains leading spaces, the spaces are used to move the field to the right.
- If the substituted value in the operand field contains leading spaces, the spaces are used to move the field to the right.
- If the substituted value contains trailing spaces, the spaces are ignored.

Listing of Generated Fields Containing Double-Byte Data: If the DBCS assembler option is specified, then the following differences apply:
- Any continuation indicators present in the model statement are discarded.
- Double-byte data that must be split at a continuation point is always readable on a device capable of presenting DBCS characters-SI and SO are inserted at the break point, and the break-point always occurs between double-byte characters.
- The continuation indicator is extended to the left, if necessary, to fill space that cannot be filled with double-byte data because of alignment and delimiter considerations. The maximum number of columns filled is 3 .
- If continuation is required and the character to the left of the continuation indicator is \(X\), then + is used as the continuation indicator so as to clearly distinguish the position of the end column. This applies to any generated field, regardless of its contents, to prevent ambiguity.
- Redundant SI/SO pairs may be present in a field after substitution. If they occur at a continuation point, the assembler does not distinguish them from SI and SO inserted in the listing by the assembler to preserve readability. Refer to the generated object code to resolve this ambiguity. For more information, see Figure 64 on page 250

\section*{Rules for Concatenation}

If a symbolic parameter in a model statement is immediately preceded or followed by other characters or another symbolic parameter, the characters that correspond to the symbolic parameter are combined in the generated statement with the other characters, or with the characters that correspond to the other symbolic parameter. This process is called concatenation.

When variable symbols are concatenated to ordinary character strings, the following rules apply to the use of the concatenation character (a period). The concatenation character is mandatory when:
1 An alphanumeric character follows a variable symbol.
2 A left parenthesis that does not enclose a subscript follows a variable symbol.
3-4 A period (.) is to be generated. Two periods must be specified in the concatenated string following a variable symbol.

The concatenation character is not required when:

5 An ordinary character string precedes a variable symbol.
6 A special character, except a left parenthesis or a period, is to follow a variable symbol.
7 A variable symbol follows another variable symbol.
8 A variable symbol is used with a subscript. The concatenation character must not be used between a variable symbol and its subscript; otherwise, the characters are considered a concatenated string and not a subscripted variable symbol.

Figure 64, in which the numbers correspond to the numbers in the above list, gives the rules for concatenating variable symbols to ordinary character strings.

Figure 64. Rules for Concatenation
\begin{tabular}{|c|c|c|c|}
\hline \multirow[b]{2}{*}{Concatenated String} & \multicolumn{2}{|r|}{Values to be Substituted} & \multirow[b]{2}{*}{Generated Result} \\
\hline & Variable Symbol & Value & \\
\hline \[
\begin{array}{ll}
\hline \text { \&FIELD.A } \\
\text { \&FIELDA }
\end{array}
\] & \&FIELD \&FIELDA & \[
\begin{aligned}
& \text { AREA } \\
& \text { SUM }
\end{aligned}
\] & \[
\begin{aligned}
& \text { AREAA } \\
& \text { SUM }
\end{aligned}
\] \\
\hline  & \[
\begin{aligned}
& \text { \&DISP } \\
& \text { \&BASE }
\end{aligned}
\] & \[
\begin{aligned}
& 100 \\
& 10
\end{aligned}
\] & 100(10) \\
\hline \begin{tabular}{l}
 \\
DC D'\&INT\&FRACT' 4
7
\end{tabular} & \[
\begin{aligned}
& \text { \&INT } \\
& \text { \&FRACT }
\end{aligned}
\] & \[
\begin{aligned}
& 99 \\
& 88
\end{aligned}
\] &  \\
\hline FIELD\&A 5 & \& A & A & FIELDA \\
\hline  & \[
\begin{aligned}
& \& A \\
& \& B
\end{aligned}
\] & \[
\begin{aligned}
& \text { A } \\
& \text { B }
\end{aligned}
\] & \(A+B * 3-D\) \\
\hline  & \[
\begin{aligned}
& \text { \&SUBSCR } \\
& \text { \&SYM(10) }
\end{aligned}
\] & \[
\begin{aligned}
& 10 \\
& \text { ENTRY }
\end{aligned}
\] & ENTRY \\
\hline
\end{tabular}

\section*{Notes:}
1. The concatenation character is not generated.

Concatenation of Fields Containing Double-Byte Data: If the DBCS assembler option is specified, then the following additional rules apply:
- Because ampersand is not recognized in double-byte data, variable symbols must not be present in double-byte data.
- The concatenation character is mandatory when double-byte data is to follow a variable symbol.
- The assembler checks for redundant SI and SO at concatenation points. If the byte to the left of the join is SI and the byte to the right of the join is SO, then the \(\mathrm{SI} / \mathrm{SO}\) pair is considered redundant and is removed.

Note: The rules for redundant SI and SO are different for variable substition and listing display, which are described at "Listing of Generated Fields Containing Double-Byte Data" on page 249

The following example shows these rules:
```

\&SYMBOL SETC '<DcDd>'
DBCS DC C'<DaDb>\&SYMBOL.<.\&.S.Y.M.B.O.L>'

```

The SI/SO pairs between double-byte characters Db and Dc, and Dd and .\&, are removed. The variable symbol \&SYMBOL is recognized between the double-byte strings but not in the double-byte strings. The result after concatenation is:
DBCS DC C'<DaDbDcDd.\&.S.Y.M.B.O.L>'

\section*{Rules for Model Statement Fields}

The fields that can be specified in model statements are the same fields that can be specified in an ordinary assembler language statement. They are the name, operation, operand, and remarks fields. You can also specify a continuation-indicator field, an identification-sequence field, and, in source macro definitions, a field before the begin column if the correct ICTL instruction has been specified. Character strings in the last three fields (in the standard format only, columns 72 through 80 ) are generated exactly as they appear in the model statement, and no values are substituted for variable symbols.

Model statements must have an entry in the operation field, and, in most cases, an entry in the operand field in order to generate valid assembler language instructions.

\section*{Name Field}

The entries allowed in the name field of a model statement, before generation, are:
- Space
- An ordinary symbol
- A sequence symbol
- A variable symbol
- Any combination of variable symbols, or system variable symbols such as \&SYSNDX, and other character strings concatenated together

The generated result must be spaces (if valid) or a character string that represents a valid assembler or machine instruction name field. Double-byte data is not valid in an assembler or machine instruction name field and must not be generated.

Variable symbols must not be used to generate comment statement indicators (* or .*).

\section*{Notes:}
1. You can not reference an ordinary symbol defined in the name field of a model statement until the macro definition containing the model statement has been called, and the model statement has been generated.
2. Restrictions on the name entry of assembler language instructions are further specified where each individual assembler language instruction is described in this manual.

\section*{Operation Field}

The entries allowed in the operation field of a model statement, before generation, are given in the following list:
- An ordinary symbol that represents the operation code for:
- Any machine instruction
- A macro instruction
- MNOTE instruction
- A variable symbol
- A combination of variable strings concatenated together
- All assembler instructions, except ICTL and conditional assembly instructions

The following rules apply to the operation field of a model statement:
- Operation code ICTL is not allowed inside a macro definition.
- The MACRO and MEND statements are not allowed in model statements; they are used only for delimiting macro definitions.
- If the REPRO operation code is specified in a model statement, no substitution is done for the variable symbols in the statement line following the REPRO statement.
- Variable symbols can be used alone or as part of a concatenated string to generate operation codes for:
- Any machine instruction
- Any assembler instruction, except COPY, ICTL, ISEQ, REPRO, and MEXIT

The generated operation code must not be an operation code for the following (or their OPSYN equivalents):
- A conditional assembly instruction
- The following assembler instructions: COPY, ICTL, ISEQ, MACRO, MEND, MEXIT, and REPRO
- Double-byte data is not valid in the operation field.

\section*{Operand Field}

The entries allowed in the operand field of a model statement, before generation, are:
- Spaces (if valid)
- An ordinary symbol
- A character string, combining alphanumeric and special characters (but not variable symbols)
- A variable symbol
- A combination of variable symbols and other character strings concatenated together
- If the DBCS assembler option is specified, character strings may contain double-byte data, provided the character strings are enclosed by apostrophes.

The allowable results of generation are spaces (if valid) and a character string that represents a valid assembler, machine instruction, or macro instruction operand field.

Variable Symbols: Variable symbols must not be used in the operand field of a ICTL, or ISEQ instruction. A variable symbol must not be used in the operand field of a COPY instruction that is inside a macro definition.

\section*{Remarks Field}

The remarks field of a model statement can contain any combination of characters. No substitution is done for variable symbols appearing in the remarks field.

Using spaces: One or more spaces must be used in a model statement to separate the name, operation, operand, and remarks fields from each other. Spaces cannot be generated between fields in order to create a complete assembler language statement. The exception to this rule is that a combined operand-remarks field can be generated with one or more spaces to separate the two fields. Note, however, that if the generated operand field is part of a macro instruction, the entire string (including spaces) is passed as an operand.

\section*{Symbolic Parameters}

Symbolic parameters let you receive values into the body of a macro definition from the calling macro instruction. You declare these parameters in the macro prototype statement. They can serve as points of substitution in the body of the macro definition and are replaced by the values assigned to them by the calling macro instruction.

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

Symbolic parameters must be valid variable symbols. A symbolic parameter consists of an ampersand followed by an alphabetic character and from 0 to 61 alphanumeric characters.

The following are valid symbolic parameters:
\begin{tabular}{ll} 
\&READER & \&LOOP2 \\
\&A23456 & \(\& N\) \\
\& 4452 & \(\& \$ 4\)
\end{tabular}

The following are not valid symbolic parameters:
```

CARDAREA first character is not an ampersand
\&256B first character after ampersand is not alphabetic
\&BCD%34 contains a special character other than initial ampersand
\&IN AREA contains a special character [space] other than initial ampersand

```

\section*{Positional Parameters}

Symbolic parameters have a local scope; that is, the name and value they are assigned only applies to the macro definition in which they have been declared.

The value of the parameter remains constant throughout the processing of the containing macro definition during each call of that definition.

\section*{Notes:}
1. Symbolic parameters must not have multiple definitions or be identical to any other variable symbols within the given local scope. This applies to the system variable symbols described in "System Variable Symbols" on page 262 and to local-scope and global-scope SET symbols described in "SET Symbols" on page 319 .
2. Symbolic parameters should not begin with \&SYS because these characters are used for system variable symbols provided with High Level Assembler.

The two kinds of symbolic parameters are:
- Positional parameters
- Keyword parameters

Each positional or keyword parameter used in the body of a macro definition must be declared in the prototype statement.

The following is an example of a macro definition with symbolic parameters.
\begin{tabular}{llll} 
& MACRO & Header \\
\&NAME & MOVE & \&TO,\&FROM & Prototype \\
\&NAME & ST & 2,SAVE & Model \\
& L & \(2, \& F R O M\) & Model \\
& ST & \(2, \& T 0\) & Model \\
& L & 2, SAVE & Model \\
& MEND & & Trailer
\end{tabular}

In the following macro instruction that calls the above macro, the characters HERE, FIELDA, and FIELDB of the MOVE macro instruction correspond to the symbolic parameters \&NAME, \&TO, and \&FROM, respectively, of the MOVE prototype statement.
```

HERE MOVE FIELDA,FIELDB

```

If the preceding macro instruction were used in a source program, the following assembler language statements would be generated:
```

HERE ST 2,SAVE
L 2,FIELDB
ST 2,FIELDA
L 2,SAVE

```

\section*{Positional Parameters}

You should use a positional parameter in a macro definition if you want to change the value of the parameter each time you call the macro definition. This is because it is easier to supply the value for a positional parameter than for a keyword parameter. You only have to write the value you want the corresponding argument to have in the correct position in the operand of the calling macro instruction. However, if you need a large number of parameters, you should use keyword parameters. The keywords make it easier to keep track of the individual values you
must specify at each call by reminding you which parameters are being given values.

See "Positional Operands" on page 300 for details of how to write macro definitions with positional parameters.

\section*{Keyword Parameters}

You should use a keyword parameter in a macro definition for a value that changes infrequently, or if you have a large number of parameters. The keyword, repeated in the operand, reminds you which parameter is being given a value and for which purpose the parameter is being used. By specifying a standard default value to be assigned to the keyword parameter, you can omit the corresponding keyword argument operand in the calling macro instruction. You can specify the corresponding keyword operands in any order in the calling macro instruction.

See "Keyword Operands" on page 301 for details of how to write macro definitions with keyword parameters.

\section*{Combining Positional and Keyword Parameters}

By using positional and keyword parameters in a prototype statement, you combine the benefits of both. You can use positional parameters in a macro definition for passing values that change frequently, and keyword parameters for passing values that do not change often.

Positional and keyword parameters can be mixed freely in the macro prototype statement.

See "Combining Positional and Keyword Operands" on page 303 for details of how to write macro definitions using combined positional and keyword parameters.

\section*{Subscripted Symbolic Parameters}

Subscripted symbolic parameters must be coded in the format:
\&PARAM(subscript)
where \&PARAM is a variable symbol and subscript is an arithmetic expression. The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (arithmetic expressions are discussed in "SETA Instruction" on page 347. The arithmetic expression can contain subscripted variable symbols. Subscripts can be nested to any level provided that the total length of an individual operand does not exceed 1024 characters.

The value of the subscript must be greater than or equal to one. The subscript indicates the position of the entry in the sublist that is specified as the value of the subscripted parameter (sublists as values in macro instruction operands are fully described in "Sublists in Operands" on page 304.

\section*{Processing Statements}

\section*{Conditional Assembly Instructions}

Conditional assembly instructions let you determine at conditional assembly time the content of the generated statements and the sequence in which they are generated. The instructions and their functions are listed below:
\begin{tabular}{ll}
\hline Conditional Assembly & Operation Done \\
\hline GBLA, GBLB, GBLC & Declaration of variable symbols (global-scope and local-scope \\
LCLA, LCLB, LCLC & SET symbols) and setting of default initial values \\
\hline SETA, SETB, SETC & Assignment of values to variable symbols (SET symbols) \\
\hline SETAF, SETCF & \begin{tabular}{l} 
External function assignment of values to variable symbols \\
(SET symbols)
\end{tabular} \\
\hline ACTR & Setting loop counter \\
\hline AGO & Unconditional branch \\
\hline AIF & Conditional branch (based on logical test) \\
\hline ANOP & Pass control to next sequential instruction (no operation) \\
\hline
\end{tabular}

Conditional assembly instructions can be used both inside macro definitions and in open code. They are described in Chapter 9, "How to Write Conditional Assembly Instructions."

\section*{Inner Macro Instructions}

Macro instructions can be nested inside macro definitions, allowing you to call other macros from within your own definition.

\section*{Other Conditional Assembly Instructions}

Several additional instructions can help you write your macro definitions. The instructions and their functions are listed below:
\begin{tabular}{ll}
\hline Inner Macro Instruction & Operation Done \\
\hline AEJECT & Skip to next page \\
\hline AINSERT & Insert statement into input stream \\
\hline AREAD & \begin{tabular}{l} 
Assign an arbitrary character string to a variable symbol \\
(SETC symbol)
\end{tabular} \\
\hline ASPACE & Insert one or more blank lines in listing \\
\hline COPY & \begin{tabular}{l} 
Copy the source statements from a source language \\
library member.
\end{tabular} \\
\hline MEXIT & Exit from the macro definition \\
\hline
\end{tabular}

\section*{AEJECT Instruction}

\section*{AEJECT Instruction}

Use the AEJECT instruction to stop the printing of the assembler listing of your macro definition on the current page, and continue the printing on the next page.

sequence_symbol
is a sequence symbol.
The AEJECT instruction causes the next line of the assembly listing of your macro definition to be printed at the top of a new page. If the line before the AEJECT statement appears at the bottom of a page, the AEJECT statement has no effect. An AEJECT instruction immediately following another AEJECT instruction causes a blank page in the listing of the macro definition.

\section*{Notes:}
1. The AEJECT instruction can only be used inside a macro definition.
2. The AEJECT instruction itself is not printed in the listing.
3. The AEJECT instruction does not affect the listing of statements generated when the macro is called.

\section*{AINSERT Instruction}

The AINSERT instruction, inside macro definitions, harnesses the power of macros to generate source statements, for instance, using variable substitution. Generated statements are queued in a special buffer and read after the macro generator finishes.

The specifications for the AINSERT instruction, which can also be used in open code, are described in "AINSERT Instruction" on page 108

\section*{AREAD Instruction}

The AREAD instruction assigns an arbitrary character string value to a SETC symbol.

The AREAD instruction has two formats. The first format lets you assign to a SETC symbol the character string value of a statement that is placed immediately after a macro instruction.

The AREAD instruction can only be used inside macro definitions.


The second format of the AREAD instruction assigns to a SETC symbol a character string containing the local time.

\section*{Assign Local Time}


SETC_symbol
is a SETC symbol. See "SETC Instruction" on page 369 .

\section*{NOSTMT}
specifies that the statement to be read by the AREAD instruction is printed in the assembly listing, but not given any statement number.

\section*{NOPRINT}
specifies that the statement does not appear in the listing, and no statement number is assigned to it.

\section*{CLOCKB}
assigns an 8-character string to SETC_symbol containing the local time in hundredths of a second since midnight.

\section*{CLOCKD}
assigns an 8-character string to SETC_symbol containing the local time in the format HHMMSSTH, where \(H H\) is a value between 00 and \(23, M M\) and SS each have a value between 00 and 59, and \(T H\) has a value between 00 and 99.

\section*{Assign Character String Value}

The first format of AREAD functions in much the same way as symbolic parameters, but instead of providing your input to macro processing as part of the macro instruction, you can supply full input records from either the AINSERT buffer (if any are present), or from the records in the primary input stream that follow immediately after the macro instruction. Any number of successive statements can be read into the macro for processing.

SETC_symbol may be subscripted. When the assembler encounters a Format-1 AREAD statement during the processing of a macro instruction, it reads the source statement following the macro instruction and assigns an 80-character string to the SETC symbol in the name field. In the case of nested macros, it reads the statement following the outermost macro instruction.

If no operand is specified, the statement to be read by AREAD is printed in the listing and assigned a statement number. The AREAD action is indicated in the listing by a minus sign between the statement number and the first character of the record.

Repeated AREAD instruction statements read successive statements. In the following example, the input record starting with INRECORD1 is read by the first AREAD statement, and assigned to the SETC symbol \&VAL. The input record starting with INRECORD2 is read by the second AREAD statement, and assigned to the SETC symbol \&VAL1.

\section*{Example:}
\begin{tabular}{ll} 
& MACRO \\
& MAC1 \\
\&VAL & AREAD \\
\&VAL1 & AREAD \\
& MEND \\
& CSECT \\
& MAC1 \\
INRECORD1 THIS IS THE STATEMENT TO BE PROCESSED FIRST \\
INRECORD2 THIS IS THE NEXT STATEMENT \\
& •. \\
&
\end{tabular}

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

\section*{Assign Local Time of Day}

The second format of AREAD functions in much the same way as a SETC instruction, but instead of supplying the value you want assigned to the SETC symbol as a character string in the operand of the AREAD instruction, the value is provided by the operating system in the form of an 8-character string containing the local time. A Format-2 AREAD instruction does not cause the assembler to read the statement following the macro instruction.

\section*{Example:}
\begin{tabular}{ll} 
& MACRO \\
& MAC2 \\
& \&VAL \\
& AREAD \\
\&VALOCKB \\
& DC \\
\& C'\&VAL' \\
& AREAD \\
& DCLOCKD \\
& DC'\&VAL1' \\
& MEND
\end{tabular}

When the macro definition described above is called, the following statements are generated:
\begin{tabular}{lll} 
& MAC2 \\
+ & DC & \(C^{\prime} 03251400^{\prime}\) \\
+ & DC & \(C^{\prime} 09015400^{\prime}\)
\end{tabular}

\section*{ASPACE Instruction}

Use the ASPACE instruction to insert one or more blank lines in the listing of a macro definition in your source module. This separates sections of macro definition code on the listing page.

sequence_symbol
is a sequence symbol.
number_of_lines
is a non-negative decimal integer that specifies the number of lines to be left blank. If number_of_lines is omitted, one line is left blank. If number_of_lines has a value greater than the number of lines remaining on the listing page, the instruction has the same effect as an AEJECT statement.

\section*{Notes:}
1. The ASPACE instruction can only be used inside a macro definition.
2. The ASPACE instruction itself is not printed in the listing.
3. The ASPACE instruction does not affect the listing of statements generated when the macro is called.

\section*{COPY Instruction}

The COPY instruction, inside macro definitions, lets you copy into the macro definition any sequence of statements allowed in the body of a macro definition. These statements become part of the body of the macro before macro processing takes place. You can also use the COPY instruction to copy complete macro definitions into a source module.

The specifications for the COPY instruction, which can also be used in open code, are described in "COPY Instruction" on page 122

\section*{MEXIT Instruction}

The MEXIT instruction provides an exit for the assembler from any point in the body of a macro definition. The MEND instruction provides an exit only from the end of a macro definition (see"MEND Statement" on page 245 for details).

The MEXIT instruction statement can be used only inside macro definitions.

sequence_symbol
is a sequence symbol.
The MEXIT instruction causes the assembler to exit from a macro definition to the next sequential instruction after the macro instruction that calls the definition. (This also applies to nested macro instructions, which are described in "Nesting Macro Instruction Definitions" on page 311.)

For example, the following macro definition contains an MEXIT statement:
\begin{tabular}{ll} 
MACRO \\
EXITS \\
DC & \\
DC & \(C^{\prime} A^{\prime}\) \\
DC & \(C^{\prime} B^{\prime}\) \\
MEXIT & \\
DC & \\
D' & \(C^{\prime} D^{\prime}\) \\
DC & \(C^{\prime} E^{\prime}\) \\
DC & \(C^{\prime} F^{\prime}\) \\
MEND &
\end{tabular}

When the macro definition described above is called, the following statements are generated:

EXITS
\begin{tabular}{lll}
+ & \(D C\) & \(C^{\prime} A^{\prime}\) \\
+ & \(D C\) & \(C^{\prime} B^{\prime}\) \\
+ & \(D C\) & \(C^{\prime} C^{\prime}\)
\end{tabular}

\section*{Comment Statements}

Two types of comment statements can be used within a macro definition:
- Ordinary comment statements
- Internal macro comment statements

\section*{Ordinary Comment Statements}

Ordinary comment statements let you make descriptive remarks about the generated output from a macro definition. Ordinary comment statements can be used in macro definitions and in open code.

An ordinary comment statement consists of an asterisk in the begin column, followed by any character string. The comment statement is used by the assembler to generate an assembler language comment statement, just as other model statements are used by the assembler to generate assembler statements. No variable symbol substitution is done.

\section*{Internal Macro Comment Statements}

You can also write internal macro comments in the body of a macro definition to describe the operations done during conditional assembly when the macro is processed.

An internal macro comment statement consists of a period in the begin column, followed by an asterisk, followed by any character string. No values are substituted for any variable symbols that are specified in internal macro comment statements.

Internal macro comment statements may appear anywhere in a macro definition.

\section*{Notes:}
1. Internal macro comments are not generated.
2. The comment character string may contain double-byte data.
3. Internal macro comment statements can be used in open code, however, they are processed as ordinary comment statements.

\section*{System Variable Symbols}

System variable symbols are a special class of variable symbols, starting with the characters \(\& S Y S\). Their values are set by the assembler according to specific rules. You cannot declare them in local-scope SET symbols or global-scope SET symbols, nor use them as symbolic parameters in macro prototype statements. You can use these symbols as points of substitution in model statements and conditional assembly instructions.

All system variable symbols are subject to the same rules of concatenation and substitution as other variable symbols.

A description of each system variable symbols begins on page 263 .
You should not prefix your SET symbols with the character sequence \&SYS. The assembler uses this sequence as a prefix to all system variable symbol names, and using them for other SET symbol names might cause future conflicts.

\section*{Scope and Variability of System Variable Symbols}

Global Scope: Some system variable symbols have values that are established at the beginning of an assembly and are available both in open code and from within macros. These symbols have global scope. Most system variable symbols with global scope have fixed values, although there are some whose value can change within a single macro expansion. The global-scope system variables symbols with variable values are \&SYSSTMT, \&SYSM_HSEV, and \&SYSM_SEV.

Local Scope: Some system variable symbols have values that are available only from within a macro expansion. These system variables have local scope. Since the value of system variable symbols with local scope is established at the beginning of a macro expansion and remains unchanged throughout the expansion, they are designated as having constant values, even though they might have different values in a later expansion of the same macro, or within inner macros.

Over half of the system variable symbols have local scope and therefore are not available in open code.

```

macro
getlocalsys
Define globals for values of interest
Gblc \&clock,\&location,\&dsname,\&nest
Gbla \&nesta
now update the globals from within the macro
\&clock setc '\&sysclock'
\&dsname setc '\&sysin_dsn'
\&nest setc '\&sysnest'
seta \&sysnest
mend
csect
define globals in opencode
Gblc \&clock,\&location,\&dsname,\&nest
Gbla \&nesta
invoke macro to update the global values
getlocalsys
now use the updated values
dc c'\&clock'
dc c'2004-06-11 17:48:42.914829'
dc c'\&nest'
dc c'1'
dc f'\&nesta'
dc f'1'
end r

```

Figure 65. Exposing the Value of a Local Scope Variable to Open Code
Uses, Values and Properties: System variable symbols have many uses, including:
- Helping to control conditional assemblies
- Capturing environmental data for inclusion in the generated object code
- Providing program debugging data

Refer to Appendix C, "Macro and Conditional Assembly Language Summary" on page 409 for a summary of the values and properties that can be assigned to system variable symbols.

\section*{\&SYSADATA_DSN System Variable Symbol}

Use \&SYSADATA_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the associated data.

The local-scope system variable symbol \&SYSADATA_DSN is assigned a read-only value each time a macro definition is called.

MVS When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSADATA_DSN is always the value stored in the JFCB for SYSADATA. If SYSADATA is allocated to DUMMY, or a NULLFILE, the value in \&SYSADATA_DSN is NULLFILE.

For example, \&SYSADATA_DSN might be assigned a value such as:
IBMAPC.SYSADATA

CMS When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to \&SYSADATA_DSN is determined as follows:
\begin{tabular}{ll}
\hline Figure 66. Contents of \&SYSADATA_DSN on CMS \\
\hline SYSADATA Allocated To: & Contents of \&SYSADATA_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Dummy file (no physical I/O) & DUMMY \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & \begin{tabular}{l} 
TAPn, where \(n\) is a value from 0 to 9, \\
\\
\end{tabular} \\
\hline
\end{tabular}

For example, \&SYSADATA_DSN might be assigned a value such as:
SAMPLE SYSADATA A1
CMS
VSE The value of the character string assigned to \&SYSADATA_DSN is the file id from the SYSADAT dibl.

For example, \&SYSADATA_DSN might be assigned a value such as:
MYDATA
\(<\) VSE

\section*{Notes:}
1. The value of the type attribute of \&SYSADATA_DSN (T'\&SYSADATA_DSN) is always \(U\).
2. The value of the count attribute of \&SYSADATA_DSN ( \(\mathrm{K}^{\prime} \& S Y S A D A T A \_D S N\) ) is equal to the number of characters assigned as a value to \&SYSADATA_DSN. In the CMS example above, the count attribute of \&SYSADATA_DSN is 20.

\section*{\&SYSADATA_MEMBER System Variable Symbol}

VSE The value of \&SYSADATA_MEMBER is always null. The value of the type attribute is O , and the value of the count attribute is 0 . LVSE

CMS, MVS - You can use \&SYSADATA_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the associated data.

The local-scope system variable symbol \&SYSADATA_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the associated data is not an MVS partitioned data set, \&SYSADATA_MEMBER is assigned a null character string.

\section*{Notes:}
1. The value of the type attribute of \&SYSADATA_MEMBER ( \(T^{\prime} \& S Y S A D A T A \_M E M B E R\) ) is \(U\), unless \&SYSADATA_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
2. The value of the count attribute of \&SYSADATA_MEMBER ( \(\mathrm{K}^{\prime} \& S Y\) SADATA_MEMBER) is equal to the number of characters assigned as a value to \&SYSADATA_MEMBER. If \&SYSADATA_MEMBER is assigned a null character string, the value of the count attribute is 0 .

\section*{\&SYSADATA_VOLUME System Variable Symbol}

Use \&SYSADATA_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler is writing the associated data.

The local-scope system variable symbol \&SYSADATA_VOLUME is assigned a read-only value each time a macro definition is called.

CMS If the assembler runs on the CMS component of the VM operating system, and the associated data is being written to a Shared File System CMS file, \&SYSADATA_VOLUME is assigned the value ** SFS.

CMS
If the volume on which the data set resides is not labeled, \&SYSADATA_VOLUME is assigned a null character string.

\section*{Notes:}
1. The value of the type attribute of \&SYSADATA_VOLUME ( \(T^{\prime} \& S Y S A D A T A \_V O L U M E\) ) is \(U\), unless \&SYSADATA_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
2. The value of the count attribute of \&SYSADATA_VOLUME ( \(\left.K^{\prime} \& S Y S A D A T A \_V O L U M E\right)\) is equal to the number of characters assigned as a value to \&SYSADATA_VOLUME. If \&SYSADATA_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .

\section*{\&SYSASM System Variable Symbol}

Use \&SYSASM to obtain the name of the assembler being used to assemble your source module. \&SYSASM has a global scope. For example, when IBM High Level Assembler for MVS \& VM \& VSE is used, \&SYSASM has the value:
high level assembler

\section*{Notes:}
1. The value of the type attribute of \&SYSASM ( \(T^{\prime} \& S Y S A S M\) ) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S A S M\) ) is the number of characters assigned. In the above example, the count attribute of \&SYSASM is 20.

\section*{\&SYSCLOCK System Variable Symbol}

Use \&SYSCLOCK to obtain the TOD clock date and time at which the macro was generated, based on Universal Time (GMT).

The local-scope system variable symbol \&SYSCLOCK is assigned a read-only value each time a macro definition is called.

The value of \&SYSCLOCK is a 26 -character string in the format:
YYYY-MM-DD HH:MM:SS.mmmmmm
where:
YYYY is a four-digit field that gives the year, including the century. It has a value between 0000 and 9999, inclusive.
\(M M\) is a two-digit field that gives the month of the year. It has a value between 01 and 12, inclusive.
\(D D\) is a two-digit field that gives the day of the month. It has a value between 01 and 31, inclusive.

HH is a two-digit field that gives the hour of the day. It has a value between 00 and 23, inclusive.
MM is a two-digit field that gives the minute of the hour. It has a value between 00 and 59 , inclusive.

SS is a two-digit field that gives the second of the minute. It has a value between 00 and 59 , inclusive.
mmmmmm is a six-digit field that gives the microseconds within the seconds. It has a value between 000000 and 999999, inclusive.

\section*{Example:}

2001-06-08 17:36:03 043284

\section*{Notes:}
1. The value of the type attribute of \&SYSCLOCK (T'\&SYSCLOCK) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S C L O C K\) ) is always 26 .

\section*{\&SYSDATC System Variable Symbol}

Use \&SYSDATC to obtain the date, including the century, on which your source module is assembled. \&SYSDATC has a global scope.

The value of \(\&\) SYSDATC is an 8 -character string in the format:
YYYYMMDD
where:
\(Y Y Y Y\) is four-digit field that gives the year, including the century. It has a value between 0000 and 9999, inclusive.
\(M M\) is two-digit field that gives the month of the year. It has a value between 01 and 12, inclusive.
\(D D\) is two-digit field that gives the day of the month. It has a value between 01 and 31 , inclusive.

\section*{Example:}

20000328

\section*{Notes:}
1. The date corresponds to the date printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of \&SYSDATC (T'\&SYSDATC) is always N .
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S D A T C\) ) is always 8.

\section*{\&SYSDATE System Variable Symbol}

Use \&SYSDATE to obtain the date, in standard format, on which your source module is assembled. \&SYSDATE has a global scope.

The value of \&SYSDATE is an 8 -character string in the format: MM/DD/YY
where:
\(M M\) is a two-digit field that gives the month of the year. It has a value between 01 and 12, inclusive.
\(D D\) is a two-digit field that gives the day of the month. It has a value between 01 and 31 , inclusive. It is separated from MM by a slash.
\(Y Y\) is a two-digit field that gives the year of the century. It has a value between 00 and 99 , inclusive. It is separated from DD by a slash.

\section*{Example:}

06/11/04

\section*{Notes:}
1. The date corresponds to the date printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of \&SYSDATE ( \(T^{\prime} \& S Y S D A T E\) ) is always \(U\).
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S D A T E\) ) is always 8.

\section*{\&SYSECT System Variable Symbol}

Use \&SYSECT in a macro definition to generate the name of the current control section. The current control section is the control section in which the macro instruction that calls the definition appears. You can't use \&SYSECT in open code.

The local-scope system variable symbol \&SYSECT is assigned a read-only value each time a macro definition is called.

The value assigned is the symbol that represents the name of the current control section from which the macro definition is called. Note that it is the control section in effect when the macro is called. A control section that has been initiated or continued by substitution does not affect the value of \&SYSECT for the expansion of the current macro. However, it may affect \&SYSECT for a subsequent macro call. Nested macros cause the assembler to assign a value to \&SYSECT that
depends on the control section in force inside the outer macro when the inner macro is called.

\section*{Notes:}
1. The control section whose name is assigned to \&SYSECT can be defined by a program sectioning statement. This can be a START, CSECT, RSECT, DSECT, or COM statement.
2. The value of the type attribute of \&SYSECT (T'\&SYSECT) is always \(U\).
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S E C T\) ) is equal to the number of characters assigned as a value to \&SYSECT.
4. Throughout the use of a macro definition, the value of \&SYSECT is considered a constant, independent of any program sectioning statements or inner macro instructions in that definition.

The next example shows these rules:
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & & \\
\hline & INNER & \&INCSECT & \\
\hline \&INCSECT & CSECT & & Statement 1 \\
\hline & DC & A(\&SYSECT) & Statement 2 \\
\hline & MEND & & \\
\hline & MACRO & & \\
\hline & OUTER1 & & \\
\hline CSOUT1 & CSECT & & Statement 3 \\
\hline & DS & 100C & \\
\hline & INNER & INA & Statement 4 \\
\hline & INNER & INB & Statement 5 \\
\hline & DC & A(\&SYSECT) & Statement 6 \\
\hline & MEND & & \\
\hline & MACRO & & \\
\hline & OUTER2 & & \\
\hline & DC & A(\&SYSECT) & Statement 7 \\
\hline & MEND & & \\
\hline MAINPROG & CSECT & & Statement 8 \\
\hline & DS & 200C & \\
\hline & OUTER1 & & Statement 9 \\
\hline & OUTER2 & & Statement 10 \\
\hline Generated & Program & & \\
\hline MAINPROG & CSECT & & \\
\hline & DS & 200C & \\
\hline CSOUT1 & CSECT & & \\
\hline & DS & 100C & \\
\hline INA & CSECT & & \\
\hline & DC & A(CSOUT1) & \\
\hline INB & CSECT & & \\
\hline & DC & A(INA) & \\
\hline & DC & A (MAINPROG) & \\
\hline & DC & A (INB) & \\
\hline
\end{tabular}

In this example:
- Statement 8 is the last program sectioning statement processed before statement 9 is processed. Therefore, \&SYSECT is assigned the value MAINPROG for macro instruction OUTER1 in statement 9. MAINPROG is substituted for \&SYSECT when it appears in statement 6.
- Statement 3 is the program sectioning statement processed before statement 4 is processed. Therefore, \&SYSECT is assigned the value CSOUT1 for macro instruction INNER in statement 4. CSOUT1 is substituted for \&SYSECT when it appears in statement 2.
- Statement 1 is used to generate a CSECT statement for statement 4. This is the last program sectioning statement that appears before statement 5 . Therefore, \&SYSECT is assigned the value INA for macro instruction INNER in statement 5. INA is substituted for \&SYSECT when it appears in statement 2.
- Statement 1 is used to generate a CSECT statement for statement 5. This is the last program sectioning statement that appears before statement 10. Therefore, \&SYSECT is assigned the value INB for macro instruction OUTER2 in statement 10. INB is substituted for \&SYSECT when it appears in statement 7.

\section*{\&SYSIN_DSN System Variable Symbol}

Use \&SYSIN_DSN in a macro definition to obtain the name of the data set from which the assembler is reading the source module.

The local system variable symbol \&SYSIN_DSN is assigned a read-only value each time a macro definition is called.

MVS If concatenated data sets are used to provide the source module, \&SYSIN_DSN has a value equal to the data set name of the data set that contains the open code source line of the macro call statement, irrespective of the nesting depth of the macro line containing the \&SYSIN_DSN reference.

When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSIN_DSN is always the value stored in the JFCB for SYSIN. \({ }^{\text {MVS }}\)

CMS When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to \&SYSIN_DSN is determined as follows:

Figure 67. Contents of \&SYSIN_DSN on CMS
\begin{tabular}{ll}
\hline SYSIN Allocated To: & Contents of \&SYSIN_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
\\
\\
\\
\\
\\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Reader & READER \\
\hline Terminal & TERMINAL \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & TAPn, where \(n\) is a value from 0 to 9, \\
& or A to F. \\
\hline
\end{tabular}

VSE When the assembler runs on the VSE operating system, the value of the character string assigned to \&SYSIN_DSN is determined as follows:

Figure 68. Contents of \&SYSIN_DSN on VSE
\begin{tabular}{ll}
\hline SYSIPT Assigned To: & Contents of \&SYSIN_DSN: \\
\hline Job stream (SYSIPT) & SYSIPT \\
\hline Disk & The file-id \\
\hline Labeled tape file & The file-id of the tape file \\
\hline Unlabeled tape file & SYSIPT \\
\hline
\end{tabular}

VSE

\section*{Examples:}

On MVS, \&SYSIN_DSN might be assigned a value such as:
IBMAPC.ASSEMBLE.SOURCE
On CMS, \&SYSIN_DSN might be assigned a value such as:
SAMPLE ASSEMBLE A1

\section*{Notes:}
1. If the SOURCE user exit provides the data set information then the value in \&SYSIN_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSIN_DSN (T'\&SYSIN_DSN) is always \(U\).
3. The value of the count attribute of \&SYSIN_DSN (K'\&SYSIN_DSN) is equal to the number of characters assigned as a value to \(\& S Y S I N \_D S N\). In the CMS example above, the count attribute of \&SYSIN_DSN is 20.
4. Throughout the use of a macro definition, the value of \&SYSIN_DSN is considered a constant.

\section*{\&SYSIN_MEMBER System Variable Symbol}

VSE The value of \&SYSIN_MEMBER is always null.
The value of the type attribute is O , and the value of the count attribute is 0 .

CMS, MVS You can use \&SYSIN_MEMBER in a macro definition to obtain the name of the data set member from which the assembler is reading the source module. If concatenated data sets are used to provide the source module, \&SYSIN_MEMBER has a value equal to the name of the data set member that contains the macro instruction that calls the definition.

The local-scope system variable symbol \&SYSIN_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set from which the assembler is reading the source module is not an MVS partitioned data set or a CMS MACLIB, \&SYSIN_MEMBER is assigned a null character string. CMS, MVS

\section*{Notes:}
1. If the SOURCE user exit provides the data set information then the value in \&SYSIN_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSIN_MEMBER (T'\&SYSIN_MEMBER) is U, unless \&SYSIN_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSIN_MEMBER (K'\&SYSIN_MEMBER) is equal to the number of characters assigned as a value to \&SYSIN_MEMBER. If \&SYSIN_MEMBER is assigned a null character string, the value of the count attribute is 0 .
4. Throughout the use of a macro definition, the value of \&SYSIN_MEMBER is considered a constant.

\section*{\&SYSIN_VOLUME System Variable Symbol}

Use \&SYSIN_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set from which the assembler is reading the source module.

CMS, MVS If concatenated data sets are used to provide the source module, \&SYSIN_VOLUME has a value equal to the volume identifier of the first volume containing the data set that contains the macro call instruction. CMS, MVS

The local-scope system variable symbol \&SYSIN_VOLUME is assigned a read-only value each time a macro definition is called.

CMS If the assembler runs on the CMS component of the VM operating system, and the source module is being read from a Shared File System CMS file, \&SYSIN_VOLUME is assigned the value ** SFS.

CMS

If the volume on which the input data set resides is not labeled, \&SYSIN_VOLUME is assigned a null character string.

\section*{Notes:}
1. If the SOURCE user exit provides the data set information then the value in \&SYSIN_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSIN_VOLUME (T'\&SYSIN_VOLUME) is U, unless \&SYSIN_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSIN_VOLUME (K'\&SYSIN_VOLUME) is equal to the number of characters assigned as a value to \&SYSIN_VOLUME. If \&SYSIN_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .
4. Throughout the use of a macro definition, the value of \&SYSIN_VOLUME is considered a constant.

\section*{\&SYSJOB System Variable Symbol}

Use \&SYSJOB to obtain the jobname of the assembly job used to assemble your source module. \&SYSJOB has a global scope.

When the assembler runs on the CMS component of the VM operating systems, \&SYSJOB is assigned a value of (NOJOB).

\section*{Notes:}
1. The value of the type attribute of \&SYSJOB ( \(T^{\prime} \& S Y S J O B\) ) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S J O B\) ) is the number of characters assigned.

\section*{\&SYSLIB_DSN System Variable Symbol}

Use \&SYSLIB_DSN in a macro definition to obtain name of the data set from which the assembler read the macro definition statements. If the macro definition is a source macro definition, \&SYSLIB_DSN is assigned the same value as \&SYSIN_DSN.

The local-scope system variable symbol \&SYSLIB_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSLIB_DSN is always the value stored in the JFCB for SYSLIB.

When the assembler runs on the CMS component of the VM operating systems, and the macro definition is a library macro definition, \&SYSLIB_DSN is assigned the file name, file type, and file mode of the data set.

> VSE When the macro definition is a library macro definition, \&SYSLIB_DSN is assigned the library name and sublibrary name of the VSE Librarian file. \(\angle\) VSE

\section*{Examples}

Under MVS, \&SYSLIB_DSN might be assigned a value such as:
SYS1.MACLIB
Under CMS, \&SYSLIB_DSN might be assigned a value such as:
DMSGPI MACLIB S2
Under VSE, \&SYSLIB_DSN might be assigned a value such as:
IJSYSRS.SYSLIB

\section*{Notes:}
1. If the LIBRARY user exit provides the data set information then the value in \&SYSLIB_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSLIB_DSN (T'\&SYSLIB_DSN) is always U.
3. The value of the count attribute of \&SYSLIB_DSN (K'\&SYSLIB_DSN) is equal to the number of characters assigned as a value to \&SYSLIB_DSN.
4. Throughout the use of a macro definition, the value of \&SYSLIB_DSN is considered a constant.

\section*{\&SYSLIB_MEMBER System Variable Symbol}

Use \&SYSLIB_MEMBER in a macro definition to obtain the name of the data set member from which the assembler read the macro definition statements. If the macro definition is a source macro definition, \&SYSLIB_MEMBER is assigned the same value as \&SYSIN_MEMBER.

The local-scope system variable symbol \&SYSLIB_MEMBER is assigned a read-only value each time a macro definition is called.

\section*{Notes:}
1. If the LIBRARY user exit provides the data set information then the value in \&SYSLIB_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide
2. The value of the type attribute of \&SYSLIB_MEMBER (T'\&SYSLIB_MEMBER) is \(U\), unless \&SYSLIB_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSLIB_MEMBER ( \(\mathrm{K}^{\prime} \& S Y S L I B \_M E M B E R\) ) is equal to the number of characters assigned as a value to \&SYSLIB_MEMBER. If \&SYSLIB_MEMBER is assigned a null character string, the value of the count attribute is 0 .
4. Throughout the use of a macro definition, the value of \&SYSLIB_MEMBER is considered a constant.

\section*{\&SYSLIB_VOLUME System Variable Symbol}

Use \&SYSLIB_VOLUME in a macro definition to obtain the volume identifier of the volume containing the data set from which the assembler read the macro definition statements. If the macro definition is a source macro definition, \&SYSLIB_VOLUME is assigned the same value as \&SYSIN_VOLUME.

The local-scope system variable symbol \&SYSLIB_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the source module is being read from a Shared File System CMS file, \&SYSLIB_VOLUME is assigned the value ** SFS.

\section*{Notes:}
1. If the LIBRARY user exit provides the data set information then the value in \&SYSLIB_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide,
2. The value of the type attribute of \&SYSLIB_VOLUME (T'\&SYSLIB_VOLUME) is \(U\), unless \&SYSLIB_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSLIB_VOLUME (K'\&SYSLIB_VOLUME) is equal to the number of characters assigned as a value to
\&SYSLIB_VOLUME. If \&SYSLIB_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .
4. Throughout the use of a macro definition, the value of \&SYSLIB_VOLUME is considered a constant.

\section*{\&SYSLIN_DSN System Variable Symbol}

Use \&SYSLIN_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the object records when assembler option OBJECT, GOFF or XOBJECT is specified.

The local-scope system variable symbol \&SYSLIN_DSN is assigned a read-only value each time a macro definition is called.

MVS The value of the character string assigned to \&SYSLIN_DSN is always the value stored in the JFCB for SYSLIN. If SYSLIN is allocated to DUMMY, or a NULLFILE, the value in \&SYSLIN_DSN is NULLFILE. MVS

CMS The value of the character string assigned to \&SYSLIN_DSN is determined as follows:

Figure 69. Contents of \&SYSLIN_DSN on CMS
\begin{tabular}{ll}
\hline SYSLIN Allocated To: & Contents of \&SYSLIN_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Dummy file (no physical I/O) & DUMMY \\
\hline Punch & PUNCH \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & \begin{tabular}{l} 
TAPn, where \(n\) is a value from 0 to 9, \\
or A to \(F\).
\end{tabular} \\
\hline
\end{tabular}

CMS
VSE The value of the character string assigned to \&SYSLIN_DSN is determined as follows:

Figure 70. Contents of \&SYSLIN_DSN on VSE
\begin{tabular}{ll}
\hline SYSLNK Assigned To: & Contents of \(\&\) SYSLIN_DSN: \\
\hline Disk file & The file-id \\
\hline Labeled tape file & The file-id of the tape file \\
\hline Unlabeled tape file & SYSLNK \\
\hline & \\
\hline VSE &
\end{tabular}

\section*{Examples:}

On MVS, \&SYSLIN_DSN might be assigned a value such as:

IBMAPC.OBJ

On CMS, \&SYSLIN_DSN might be assigned a value such as:
SAMPLE TEXT A1

\section*{Notes:}
1. If the OBJECT user exit provides the data set information then the value in \&SYSLIN_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSLIN_DSN (T'\&SYSLIN_DSN) is always U.
3. The value of the count attribute of \&SYSLIN_DSN (K'\&SYSLIN_DSN) is equal to the number of characters assigned as a value to \&SYSLIN_DSN.

\section*{\&SYSLIN_MEMBER System Variable Symbol}
vSE The value of \(\& S Y S L I N \_M E M B E R\) is always null.
The value of the type attribute is O , and the value of the count attribute is 0 .
```

< VSE

```

CMS, MVS You can use \(\& S Y S L I N \_M E M B E R\) in a macro definition to obtain the name of the data set member to which the assembler is writing the object module when the assembler option OBJECT, GOFF or XOBJECT is specified.

The local-scope system variable symbol \&SYSLIN_MEMBER is assigned a read-only value each time a macro definition is called.

If the library to which the assembler is writing the object records is not an MVS partitioned data set, \&SYSLIN_MEMBER is assigned a null character string.
CMS, MVS
Notes:
1. If the OBJECT user exit provides the data set information then the value in \&SYSLIN_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide,
2. The value of the type attribute of \&SYSLIN_MEMBER (T'\&SYSLIN_MEMBER) is \(U\), unless \&SYSLIN_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \(\& S Y S L I N \_M E M B E R\) ( \(\mathrm{K}^{\prime} \& S Y S L I N \_M E M B E R\) ) is equal to the number of characters assigned as a value to \&SYSLIN_MEMBER. If \&SYSLIN_MEMBER is assigned a null character string, the value of the count attribute is 0 .

\section*{\&SYSLIN_VOLUME System Variable Symbol}

Use \&SYSLIN_VOLUME in a macro definition to obtain the volume identifier of the object data set. The volume identifier is of the first volume containing the data set. \&SYSLIN_VOLUME is only assigned a value when you specify the OBJECT, GOFF or XOBJECT assembler option.

The local-scope system variable symbol \&SYSLIN_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the assembler listing is being written to a Shared File System CMS file, \&SYSLIN_VOLUME is assigned the value ** SFS.

If the volume on which the data set resides is not labeled, \&SYSLIN_VOLUME is assigned a null character string.

\section*{Notes:}
1. If the OBJECT user exit provides the data set information then the value in \&SYSLIN_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSLIN_VOLUME (T'\&SYSLIN_VOLUME) is \(U\), unless \&SYSLIN_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSLIN_VOLUME (K'\&SYSLIN_VOLUME) is equal to the number of characters assigned as a value to \&SYSLIN_VOLUME. If \&SYSLIN_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .

\section*{\&SYSLIST System Variable Symbol}

Use \&SYSLIST instead of a positional parameter inside a macro definition; for example, as a point of substitution. By varying the subscripts attached to \&SYSLIST, you can refer to any sublist entry in a macro call operand, or any positional operands in a macro call. You can also refer to positional operands for which no corresponding positional parameter is specified in the macro prototype statement.

The local-scope system variable symbol \&SYSLIST is assigned a read-only value each time a macro definition is called.
\&SYSLIST refers to the complete list of positional operands specified in a macro instruction. \&SYSLIST does not refer to keyword operands. However, \&SYSLIST cannot be specified as \&SYSLIST without a subscript. One of the two following forms must be used for references or as a point of substitution:
1. \&SYSLIST( \(n\) ) can be used to refer to the \(n\)-th positional operand
2. If the \(n\)-th operand is a sublist, then \(\& \operatorname{SYSLIST}(n, m)\) can be used to refer to the \(m\)-th operand in the sublist.
3. When referring to multilevel (nested) sublists in operands of macro instructions, refer to elements of inner sublists by using the applicable number of subscripts for \&SYSLIST.

The subscripts \(n\) and \(m\) can be any arithmetic expression allowed in the operand of a SETA instruction (See "SETA Instruction" on page 347). The subscript \(n\) must be greater than or equal to 0 . The subscript \(m\) and any additional subscripts after \(m\) must be greater than or equal to 1 .

The examples below show the values assigned to \&SYSLIST according to the value of its subscripts \(n\) and \(m\).
\begin{tabular}{|c|c|c|}
\hline \multicolumn{3}{|l|}{Macro instruction:} \\
\hline NAME MACALL & \multicolumn{2}{|l|}{ONE,TWO, \((3,(4,5,6),, 8)\), TEN, ()} \\
\hline \multicolumn{3}{|l|}{Use Within a} \\
\hline Macro Definition: & Value & See \\
\hline \&SYSLIST(2) & TWO & \\
\hline \&SYSLIST \((3,1)\) & 3 & \\
\hline \&SYSLIST \((3,2,2)\) & 5 & \\
\hline \&SYSLIST (4) & Nul 1 & 1 \\
\hline \&SYSLIST(12) & Nul 1 & 1 \\
\hline \&SYSLIST \((3,3)\) & Nul 1 & 2 \\
\hline \&SYSLIST \((3,5)\) & Nul 1 & 2 \\
\hline \&SYSLIST \((2,1)\) & TW0 & 3 \\
\hline \&SYSLIST \((2,2)\) & Nul 1 & \\
\hline \&SYSLIST(0) & NAME & 4 \\
\hline \&SYSLIST(3) & \((3,(4,5,6),, 8)\) & \\
\hline \&SYSLIST(11) & () & \\
\hline \&SYSLIST \((11,1)\) & Nul 1 & 2 \\
\hline
\end{tabular}

\section*{Notes:}
1. If the position indicated by \(n\) refers to an omitted operand, or refers to an entry past the end of the list of positional operands specified, the null character string is substituted for \&SYSLIST( \(n\) ).
2. If the position (in a sublist) indicated by the second subscript, \(m\), refers to an omitted entry, or refers past the end of the list of entries specified in the sublist referred to by the first subscript \(n\), the null character string is substituted for \&SYSLIST( \(n, m\) ).
3. If the \(n\)-th positional operand is not a sublist, \&SYSLIST \((n, 1)\) refers to the operand. However, \&SYSLIST \((n, m)\), where \(m\) is greater than 1 , will cause the null character string to be substituted.
4. If the value of subscript \(n\) is 0 , then \(\& \operatorname{SYSLIST}(n)\) is assigned the value specified in the name field of the macro instruction, except when it is a sequence symbol.

Attribute references can be made to the previously described forms of \&SYSLIST. The attributes are the attributes inherent in the positional operands or sublist entries to which you refer. However, the number attribute of \&SYSLIST ( \(\mathrm{N}^{\prime} \& S Y S L I S T\) ) is different from the number attribute described in "Data Attributes" on page 324
One of two forms can be used for the number attribute:
- To indicate the number of positional operands specified in a call, use the form N'\&SYSLIST.
- To indicate the number of sublist entries that have been specified in a positional operand, use the form \(\mathrm{N}^{\prime} \& S Y S L I S T(n)\).
- To indicate the number of entries in nested sublists, specify the appropriate set of subscripts need to reference the selected sublist.

\section*{Notes:}
1. N'\&SYSLIST includes any positional operands that are omitted. Positional operands are omitted by coding a comma where an operand is expected.
2. \(N^{\prime} \& S Y S L I S T(n)\) includes those sublist entries specifically omitted by specifying the comma that would normally have followed the entry.
3. If the operand indicated by \(n\) is not a sublist, \(\mathrm{N}^{\prime} \& \operatorname{SYSLIST}(n)\) is 1 . If it is omitted, \(\mathrm{N}^{\prime} \& \operatorname{SYSLIST}(n)\) is 0 .

The COMPAT(SYSLIST) assembler option instructs the assembler to treat sublists in macro instruction operands as character strings, not sublists. See the HLASM Programmer's Guide for a description of the COMPAT(SYSLIST) assembler option.

\section*{Examples of sublists:}
\begin{tabular}{lcc} 
Macro & Instruction & \(N^{\prime} \& S Y S L I S T\) \\
MACLST & \(1,2,3,4\) & 4 \\
MACLST & A,B, D, E & 5 \\
MACLST & A, B, C,D & 5 \\
MACLST & (A,B,C), (D,E,F) & 2 \\
MACLST & & 0 \\
MACLST & KEY1=A, KEY2=B & 0 \\
MACLST & A,B,KEY1=C & 2
\end{tabular}
\begin{tabular}{lll} 
MACSUB & A, \((1,2,3,4,5), B\) & 5 \\
MACSUB & A, (1, \(3,, 5), B\) & 5 \\
MACSUB & A, \((, 2,3,4,5), B\) & 5 \\
MACSUB & A,B,C & 1 \\
MACSUB & A, C & 0 \\
MACSUB & A, (),C & 1 \\
MACSUB & A,KEY=(A,B,\(C)\) & 0 \\
MACSUB & & 0
\end{tabular}

\section*{\&SYSLOC System Variable Symbol}

Use \&SYSLOC in a macro definition to generate the name of the location counter in effect. If you have not coded a LOCTR instruction between the macro instruction and the preceding START, CSECT, RSECT, DSECT, or COM instruction, the value of \(\& S Y S L O C\) is the same as the value of \(\& S Y S E C T\).

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

\section*{Notes:}
1. The value of the type attribute of \&SYSLOC ( \(\left.T^{\prime} \& S Y S L O C\right)\) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S L O C\) ) is equal to the number of characters assigned as a value to \&SYSLOC.
3. Throughout the use of a macro definition, the value of \(\& S Y S L O C\) is considered a constant.

\section*{\&SYSMAC System Variable Symbol}

By varying the subscripts attached to the \&SYSMAC you can refer to the name of any of the macros called between open code and the current nesting level, that is, \&SYSMAC(\&SYSNEST) returns 'OPEN CODE'. Valid subscripts are 0 to \&SYSNEST. If \&SYSMAC is used with a subscript greater than \&SYSNEST, a null character string is returned.
\&SYSMAC with no subscript is treated as \&SYSMAC(0) and so provides the name of the macro being expanded. This is not considered to be an error and so no message is issued.

The local-scope system variable symbol \&SYSMAC is assigned a read-only value each time a macro definition is called.

\section*{Notes:}
1. The value of the type attribute of \&SYSMAC (T'\&SYSMAC(n)) is \(U\), unless \&SYSMAC( n ) is assigned a null character string, in which case the value of the type attribute is O .
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S M A C(n)\) ) is equal to the number of characters assigned as a value to \(\& \operatorname{SYSMAC}(\mathrm{n})\).

\section*{\&SYSM_HSEV System Variable Symbol}

Use \&SYSM_HSEV to get the highest MNOTE severity so far for the assembly.
The global-scope system variable symbol \&SYSM_HSEV is assigned a read-only value. The assembler compares this value with the severity of MNOTE assembler instructions as they are encountered and, if lower, updates it with the higher value.

\section*{Notes:}
1. The value of the variable symbol is supplied as three numeric characters, not as an arithmetic (binary) value.
2. The value of the type attribute of \&SYSM_SEV (T'\&SYSM_SEV) is always N.
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S M \_S E V\) ) is always 3.
4. The value of \&SYSM_HSEV is unreliable if any MNOTE is incorrectly coded such that a diagnostic message is generated for the MNOTE statement. The cause of the diagnostic message must be corrected.

In Figure 71 on page 281 the \&SYSM_HSEV variable is updated immediately when an MNOTE is issued with a higher severity.

\section*{\&SYSM_SEV System Variable Symbol}

Use \&SYSM_SEV to get the highest MNOTE severity code for the macro most recently called directly from this level.

The global-scope system variable symbol \&SYSM_SEV is assigned a read-only value. The assembler assigns a value of zero when a macro is called and when a macro returns (MEND or MEXIT), the highest severity of all MNOTE assembler instructions executed in the called macro is used to update the variable.

\section*{Notes:}
1. The value of the variable symbol is supplied as three numeric characters, not as an arithmetic (binary) value.
2. The value of the type attribute of \&SYSM_SEV ( \(T^{\prime} \& S Y S M \_S E V\) ) is always \(N\).
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S M \_S E V\) ) is always 3.
4. The value of \(\& S Y S M\) SEV is unreliable if any MNOTE is incorrectly coded such that a diagnostic message is generated for the MNOTE statement. The cause of the diagnostic message must be corrected.

In Figure 71 on page 281the \(\& S Y S M\) _SEV variable has a value of 0 until INNER returns. The OUTER macro uses \&SYSM_SEV to determine which statements to generate, and in this case issues an MNOTE to pass the severity back to the open code.


Figure 71. Example of the behavior of the \&SYSM_HSEV and \&SYSM_SEV variables.

\section*{\&SYSNDX System Variable Symbol}

For each macro invocation, a new value of \&SYSNDX is assigned. The previous value is incremented by 1 . Thus, you can attach \&SYSNDX to the end of a symbol inside a macro definition to generate a unique suffix for that symbol each time you call the definition. Although an apparently identical symbol is to be generated by two or more calls to the same definition, the suffix provided by \&SYSNDX produces two or more unique symbols. For example, the symbol ABC\&SYSNDX could generate \(\mathrm{ABC0001}\) on one invocation of a macro, and \(\mathrm{ABC0002}\) on the next invocation. Thus you avoid an error being flagged for multiply defined symbols.

The local-scope system variable symbol \&SYSNDX is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to \&SYSNDX is a number from 1 to 9999999 . For the numbers 0001 through 9999, four digits are generated. For the numbers 10000 through 9999999, the value is generated with no zeros to the left. The value 0001 is assigned to the first macro called by a program, and is incremented by one for each subsequent macro call (including nested macro calls).

The maximum value for \&SYSNDX can be controlled by the MHELP instruction described under "MHELP Control on \&SYSNDX" on page 398

\section*{Notes:}
1. \&SYSNDX does not generate a valid symbol, and it must:
- Follow the alphabetic character to which it is concatenated
- Be concatenated to a symbol containing 59 characters or fewer
2. The value of the type attribute of \&SYSNDX (T'\&SYSNDX) is always \(N\).
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S N D X\) ) is equal to the number of digits generated. If a symbol generated by one macro is to be referenced by code generated by another macro, the two macros must provide means for communicating the necessary information. Their respective values of \&SYSNDX cannot be guaranteed to differ by any fixed amount.

The example that follows shows the use of \&SYSNDX, and a way to communicate local \&SYSNDX values among macro instructions. It is assumed that the first macro instruction processed, OUTER1, is the 106th macro instruction processed by the assembler.
\begin{tabular}{|c|c|c|c|c|}
\hline \multirow[t]{3}{*}{} & \multicolumn{4}{|l|}{MACRO} \\
\hline & INNER1 & & & \\
\hline & GBLC & \&NDXNUM & & \\
\hline \multirow[t]{5}{*}{A\&SYSNDX} & SR & 2,5 & Statement 1 & \\
\hline & CR & 2,5 & & \\
\hline & BE & B\&NDXNUM & Statement 2 & \\
\hline & B & A\&SYSNDX & Statement 3 & \\
\hline & \multicolumn{4}{|l|}{MEND} \\
\hline & MACRO & & & \\
\hline \multirow[t]{2}{*}{\&NAME} & \multicolumn{4}{|l|}{OUTER1} \\
\hline & GBLC & \&NDXNUM & & \\
\hline \&NDXNUM & SETC & '\&SYSNDX' & Statement 4 & \\
\hline \multirow[t]{3}{*}{\&NAME} & SR & 2,4 & & \\
\hline & AR & 2,6 & & \\
\hline & INNER1 & & Statement 5 & \\
\hline \multirow[t]{2}{*}{B\&SYSNDX} & S & \(2,=F^{\prime} 1000^{\prime}\) & Statement 6 & \\
\hline & \multicolumn{4}{|l|}{MEND} \\
\hline ALPHA & OUTER1 & & Statement 7 & \\
\hline BETA & OUTER1 & & Statement 8 & \\
\hline \multirow[t]{2}{*}{ALPHA} & SR & 2,4 & & \\
\hline & AR & 2,6 & & \\
\hline \multirow[t]{4}{*}{A0107} & SR & 2,5 & & \\
\hline & CR & 2,5 & & \\
\hline & BE & B0106 & & \\
\hline & B & A0107 & & \\
\hline B0106 & S & \(2,=\mathrm{F}^{\prime} 1000^{\prime}\) & & \\
\hline \multirow[t]{2}{*}{BETA} & SR & 2,4 & & \\
\hline & AR & 2,6 & & \\
\hline \multirow[t]{4}{*}{A0109} & SR & 2,5 & & \\
\hline & CR & 2,5 & & \\
\hline & BE & B0108 & & \\
\hline & B & A0109 & & \\
\hline B0108 & S & \(2,=\mathrm{F}^{\prime} 1000^{\prime}\) & & \\
\hline
\end{tabular}

Statement 7 is the 106th macro instruction processed. Therefore, \&SYSNDX is assigned the number 0106 for that macro instruction. The number 0106 is substituted for \&SYSNDX when it is used in statements 4 and 6. Statement 4 is used to assign the character value 0106 to the SETC symbol \&NDXNUM Statement 6 is used to create the unique name B 0106 .

Statement 5 is the 107th macro instruction processed. Therefore, \&SYSNDX is assigned the number 0107 for that macro instruction. The number 0107 is substituted for \&SYSNDX when it is used in statements 1 and 3 . The number 0106 is substituted for the global-scope SETC symbol \&NDXNUM in statement 2.

Statement 8 is the 108th macro instruction processed. Therefore, each occurrence of \(\&\) SYSNDX is replaced by the number 0108 . For example, statement 6 is used to create the unique name B0108.

When statement 5 is used to process the 108th macro instruction, statement 5 becomes the 109th macro instruction processed. Therefore, each occurrence of \&SYSNDX is replaced by the number 0109. For example, statement 1 is used to create the unique name A0109.

\section*{\&SYSNEST System Variable Symbol}

Use \&SYSNEST to obtain the current macro instruction nesting level.
The local-scope system variable symbol \&SYSNEST is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to \&SYSNEST is a number from 1 to 99999999. No leading zeros are generated as part of the number. When a macro is called from open code, the value assigned to \&SYSNEST is the number 1. Each time a macro definition is called by an inner macro instruction, the value assigned to \&SYSNEXT is incremented by 1. Each time an inner macro exits, the value is decremented by 1.

\section*{Notes:}
1. The value of the type attribute of \&SYSNEST (T'\&SYSNEST) is always N.
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S N E S T\) ) is equal to the number of digits assigned.

The following example shows the values assigned to \&SYSNEST:
\begin{tabular}{|c|c|c|}
\hline \multicolumn{3}{|l|}{MACRO} \\
\hline OUTER & & \\
\hline DC & A(\&SYSNEST) & Statement 1 \\
\hline INNER1 & & Statement 2 \\
\hline INNER2 & & Statement 3 \\
\hline \multicolumn{3}{|l|}{MEND} \\
\hline \multicolumn{3}{|l|}{MACRO} \\
\hline \multicolumn{3}{|l|}{INNER1} \\
\hline DC & A(\&SYSNEST) & Statement 4 \\
\hline INNER2 & & Statement 5 \\
\hline \multicolumn{3}{|l|}{MEND} \\
\hline \multicolumn{3}{|l|}{MACRO} \\
\hline \multicolumn{3}{|l|}{INNER2} \\
\hline DC & A(\&SYSNEST) & Statement 6 \\
\hline \multicolumn{3}{|l|}{MEND} \\
\hline OUTER & & Statement 7 \\
\hline DC & A(1) & \\
\hline DC & A(2) & \\
\hline DC & A (3) & \\
\hline DC & A(2) & \\
\hline
\end{tabular}

Statement 7 is in open code. It calls the macro OUTER. \&SYSNEST is assigned a value of 1 which is substituted in statement 1 .

Statement 2, within the macro definition of OUTER, calls macro INNER1. The value assigned to \&SYSNEST is incremented by 1 . The value 2 is substituted for \&SYSNEST in statement 4.

Statement 5, within the macro definition of INNER1, calls macro INNER2. The value assigned to \&SYSNEST is incremented by 1 . The value 3 is substituted for \&SYSNEST in statement 6.

When the macro INNER2 exits, the value assigned to \&SYSNEST is decremented by 1. The value of \(\& S Y N E S T\) is 2 .

When the macro INNER1 exits, the value assigned to \&SYSNEST is decremented by 1. The value of \(\& S Y S N E S T\) is 1 .

Statement 3, within the macro definition of OUTER, calls macro INNER2. The value assigned to \&SYSNEST is incremented by 1 . The value 2 is substituted for \&SYSNEST in statement 6.

\section*{\&SYSOPT_DBCS System Variable Symbol}

You can use \&SYSOPT_DBCS to determine if the DBCS assembler option was supplied for the assembly of your source module. \&SYSOPT_DBCS is a Boolean system variable symbol, and has a global scope.

If the DBCS assembler option was specified, \&SYSOPT_DBCS is assigned a value of 1 . If the DBCS assembler option was not specified, \&SYSOPT_DBCS is assigned a value of 0 .

For more information about the DBCS assembler option, see the HLASM Programmer's Guide.

\section*{Notes:}
1. The value of the type attribute of \&SYSOPT_DBCS (T'\&SYSOPT_DBCS) is always N .
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S O P T \_D B C S\) ) is always 1 .

\section*{\&SYSOPT_OPTABLE System Variable Symbol}

Use \&SYSOPT_OPTABLE to determine the value that was specified for the OPTABLE assembler option. \&SYSOPT_OPTABLE has a global scope.

The value that was specified for the OPTABLE assembler option indicates which operation code table the assembler has loaded, and is using.

For more information about the OPTABLE assembler option, see your HLASM Programmer's Guide.

\section*{Notes:}
1. The value of the type attribute of \&SYSOPT_OPTABLE (T'\&SYSOPT_OPTABLE) is always U.
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S O P T \_O P T A B L E\) ) is the number of characters assigned.

\section*{\&SYSOPT_RENT System Variable Symbol}

Use \&SYSOPT_RENT to determine if the RENT assembler option was specified for the assembly of your source module. The RENT option instructs the assembler to check for possible coding violations of program reenterability. \&SYSOPT_RENT is a Boolean system variable symbol, and has a global scope.

If the RENT assembler option was specified, \&SYSOPT_RENT is assigned a value of 1 . If the RENT assembler option was not specified, \&SYSOPT_RENT is assigned a value of 0 .

For more information about the RENT assembler option, see your HLASM Programmer's Guide.

\section*{Notes:}
1. The value of the type attribute of \&SYSOPT_RENT (T'\&SYSOPT_RENT) is always N .
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S O P T \_R E N T\) ) is always 1.

\section*{\&SYSOPT_XOBJECT System Variable Symbol}

The \&SYSOPT_XOBJECT system variable is set to 1 if GOFF or XOBJECT is specified, otherwise it is set to 0 .
\&SYSOPT_XOBJECT is a Boolean system variable symbol with global scope.

\section*{Notes:}
1. The value of the type attribute of \&SYSOPT_XOBJECT
( \(\mathrm{T}^{\prime} \& S Y S O P T \_X O B J E C T\) ) is always N .
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S O P T \_X O B J E C T\) ) is always 1 .

\section*{\&SYSPARM System Variable Symbol}

The \&SYSPARM system variable is assigned a read-only value from the assembler option SYSPARM. It is treated as a global-scope SETC symbol in a source module except that its value cannot be changed. (Refer to chapter 3 of the Programmer's Guide for information on assembler options.)

\section*{Notes:}
1. The largest value that \&SYSPARM can hold is 1024 characters. However, if the PARM field of the EXEC statement is used to specify its value, the PARM field restrictions reduce its maximum possible length.
2. No values are substituted for variable symbols in the specified value, however, on MVS and VSE, you must use double ampersands to represent a single ampersand.
3. On MVS and VSE, you must use two single quotation marks to represent a single quotation mark, because the entire EXEC PARM field is enclosed in single quotation marks.
4. If the SYSPARM assembler option is not specified, \&SYSPARM is assigned the default value that was specified when the assembler was installed on your system.

If a default value for SYSPARM was not specified when the assembler was installed on your system, \&SYSPARM is assigned a value of the null character string.
5. The value of the type attribute of \&SYSPARM (T'\&SYSPARM) is \(U\), unless \&SYSPARM is assigned a null value, in which case the value of the type attribute is O .
6. The value of the count attribute (K'\&SYSPARM) is the number of characters assigned as a value to \&SYSPARM. If \&SYSPARM is assigned a null character string, the value of the count attribute is 0 .
7. If the SYSPARM option is passed to the assembler via the ASMAOPT file (CMS and MVS) or Librarian member (VSE) and the option contains imbedded spaces, it must be enclosed in quotes.

\section*{\&SYSPRINT_DSN System Variable Symbol}

Use \&SYSPRINT_DSN in a macro definition to obtain the name of the data set to which the assembler writes the assembler listing.

The local-scope system variable symbol \&SYSPRINT_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSPRINT_DSN is always the value stored in the JFCB for SYSPRINT. If SYSPRINT is allocated to DUMMY, or a NULLFILE, the value in \&SYSPRINT_DSN is NULLFILE.

When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to \&SYSPRINT_DSN is determined as follows:

Figure 72. Contents of \&SYSPRINT_DSN on CMS
\begin{tabular}{ll}
\hline SYSPRINT Allocated To: & Contents of \&SYSPRINT_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Dummy file (no physical I/O) & DUMMY \\
\hline Printer & PRINTER \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & \begin{tabular}{l} 
TAPn, where \(n\) is a value from 0 to 9, \\
or A to F.
\end{tabular} \\
\hline Terminal & TERMINAL \\
\hline
\end{tabular}

When the assembler runs on VSE, the value of the character string assigned to \&SYSPRINT_DSN is determined as follows:

Figure 73. Contents of \&SYSPRINT_DSN on VSE
\begin{tabular}{ll}
\hline SYSLST Assigned To: & Contents of \&SYSPRINT_DSN: \\
\hline Disk file (not for dynamic partitions) & The file-id \\
\hline Printer & SYSLST \\
\hline Labeled tape file & The file-id of the tape file \\
\hline Unlabeled tape file & SYSLST \\
\hline
\end{tabular}

\section*{Examples:}

On MVS, \&SYSPRINT_DSN might be assigned a value such as:
IBMAPC.IBMAPCA.JOB06734.D0000102.?
On CMS, \&SYSPRINT_DSN might be assigned a value such as:
SAMPLE LISTING A1

\section*{Notes:}
1. If the LISTING user exit provides the listing data set information then the value in \&SYSPRINT_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide
2. The value of the type attribute of \&SYSPRINT_DSN (T'\&SYSPRINT_DSN) is always U.
3. The value of the count attribute of \&SYSPRINT_DSN (K'\&SYSPRINT_DSN) is equal to the number of characters assigned as a value to \&SYSPRINT_DSN.

\section*{\&SYSPRINT_MEMBER System Variable Symbol \\ VSE The value of \&SYSPRINT_MEMBER is always null.}

The value of the type attribute is O , and the value of the count attribute is 0 .

\begin{abstract}
VSE
CMS, MVS You can use \&SYSPRINT_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the assembler listing.
\end{abstract}

The local-scope system variable symbol \&SYSPRINT_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the assembler listing is not an MVS partitioned data set, \&SYSPRINT_MEMBER is assigned a null character string.

\section*{CMS, MVS}

Notes:
1. If the LISTING user exit provides the listing data set information then the value in \&SYSPRINT_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSPRINT_MEMBER ( \(T\) '\&SYSPRINT_MEMBER) is \(U\), unless \&SYSPRINT_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSPRINT_MEMBER
( \(\mathrm{K}^{\prime} \& S Y S P R I N T \_M E M B E R\) ) is equal to the number of characters assigned as a value to \&SYSPRINT_MEMBER. If \&SYSPRINT_MEMBER is assigned a null character string, the value of the count attribute is 0 .

\section*{\&SYSPRINT_VOLUME System Variable Symbol}

Use \&SYSPRINT_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler writes the assembler listing.

The local-scope system variable symbol \&SYSPRINT_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the assembler listing writes to a Shared File System CMS file, \&SYSPRINT_VOLUME is assigned the value ** SFS.

If the volume on which the data set resides is not labeled, \&SYSPRINT_VOLUME is assigned a null character string.

\section*{Notes:}
1. If the LISTING user exit provides the listing data set information then the value in \&SYSPRINT_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSPRINT_VOLUME ( \(\mathrm{T}^{\prime} \& S Y S P R I N T \_V O L U M E\) ) is U, unless \&SYSPRINT_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSPRINT_VOLUME ( \(\mathrm{K}^{\prime} \& S Y S P R I N T \_V O L U M E\) ) is equal to the number of characters assigned as a value to \&SYSPRINT_VOLUME. If \&SYSPRINT_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .

\section*{\&SYSPUNCH_DSN System Variable Symbol}

Use \&SYSPUNCH_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the object records when assembler option DECK is specified.

The local-scope system variable symbol \&SYSPUNCH_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSPUNCH_DSN is always the value stored in the JFCB for SYSPUNCH. If SYSPUNCH is allocated to DUMMY, or a NULLFILE, the value in \&SYSPUNCH_DSN is NULLFILE.

When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to \&SYSPUNCH_DSN is determined as follows:

Figure 74. Contents of \&SYSPUNCH_DSN on CMS
\begin{tabular}{ll}
\hline SYSPUNCH Allocated To: & Contents of \&SYSPUNCH_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Dummy file (no physical I/O) & DUMMY \\
\hline Punch & PUNCH \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & \begin{tabular}{l} 
TAPn, where \(n\) is a value from 0 to 9, \\
or A to \(F\).
\end{tabular} \\
\hline
\end{tabular}

On VSE, the value of the character string assigned to \&SYSPUNCH_DSN is determined as follows:

Figure 75. Contents of \&SYSPUNCH_DSN on VSE
\begin{tabular}{ll}
\hline SYSPCH Assigned To: & Contents of \&SYSPUNCH_DSN: \\
\hline Disk file & The file-id \\
\hline Punch & SYSPCH \\
\hline Labeled tape file & The file-id of the tape file \\
\hline Unlabeled tape file & SYSPCH \\
\hline
\end{tabular}

\section*{Examples:}

On MVS, \&SYSPUNCH_DSN might be assigned a value such as:
IBMAPC.IBMAPCA.JOB06734.D0000103.?

On CMS, \&SYSPUNCH_DSN might be assigned a value such as:
PUNCH

\section*{Notes:}
1. If the PUNCH user exit provides the punch data set information then the value in \&SYSPUNCH_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide,
2. The value of the type attribute of \&SYSPUNCH_DSN (T'\&SYSPUNCH_DSN) is always \(U\).
3. The value of the count attribute of \&SYSPUNCH_DSN (K'\&SYSPUNCH_DSN) is equal to the number of characters assigned as a value to \&SYSPUNCH_DSN.

\section*{\&SYSPUNCH_MEMBER System Variable Symbol}

VSE The value of \&SYSPUNCH_MEMBER is always null.
The value of the type attribute is O , and the value of the count attribute is 0 .

CMS, MVS You can use \&SYSPUNCH_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the object records when the assembler option DECK is specified.

The local system variable symbol \&SYSPUNCH_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the object records is not an MVS partitioned data set, \&SYSPUNCH_MEMBER is assigned a null character string.

CMS, MVS

\section*{Notes:}
1. If the PUNCH user exit provides the punch data set information then the value in \&SYSPUNCH_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSPUNCH_MEMBER (T'\&SYSPUNCH_MEMBER) is U, unless \&SYSPUNCH_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSPUNCH_MEMBER ( \(\mathrm{K}^{\prime} \& S Y S P U N C H \_M E M B E R\) ) is equal to the number of characters assigned as a value to \&SYSPUNCH_MEMBER. If \&SYSPUNCH_MEMBER is assigned a null character string, the value of the count attribute is 0 .

\section*{\&SYSPUNCH_VOLUME System Variable Symbol}

Use \&SYSPUNCH_VOLUME in a macro definition to obtain the volume identifier of the object data set. The volume identifier is of the first volume containing the data set. \&SYSPUNCH_VOLUME is only assigned a value when you specify the DECK assembler option.

The local-scope system variable symbol \&SYSPUNCH_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the object records are being written to a Shared File System CMS file, \&SYSPUNCH_VOLUME is assigned the value ** SFS.

If the volume on which the data set resides is not labeled, \&SYSPUNCH_VOLUME is assigned a null character string.

\section*{Notes:}
1. If the PUNCH user exit provides the punch data set information then the value in \&SYSPUNCH_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSPUNCH_VOLUME ( \(T^{\prime} \& S Y S P U N C H \_V O L U M E\) ) is \(U\), unless \&SYSPUNCH_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSPUNCH_VOLUME ( \(\mathrm{K}^{\prime}\) \&SYSPUNCH_VOLUME) is equal to the number of characters assigned as a value to \&SYSPUNCH_VOLUME. If \&SYSPUNCH_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .

\section*{\&SYSSEQF System Variable Symbol}

Use \&SYSSEQF in a macro definition to obtain the value of the identification-sequence field of the macro instruction in open code that caused, directly or indirectly, the macro to be called.

The local-scope system variable symbol \&SYSSEQF is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to \&SYSSEQF is determined as follows:
1. If no ICTL instruction has been specified and sequence checking is not active, the contents of columns 73 to 80 inclusive of the source statement are assigned to \&SYSSEQF.
2. If an ICTL instruction has been specified, but sequence checking is not active, the contents of the columns of the source statement to the right of the continuation-indicator column are assigned to \&SYSSEQF. If the end column or the continuation-indicator column is \(80, \&\) SYSSEQF is assigned a null character string.
3. If an ISEQ instruction with operands has been specified to start sequence checking, the contents of columns specified in the ISEQ instruction operand are assigned to \&SYSSEQF.
4. If an ISEQ instruction without an operand has been specified to end sequence checking, steps (1) and (2) are used to determine the value assigned to \&SYSSEQF.

\section*{Notes:}
1. The value of the type attribute of \&SYSSEQF (T'\&SYSSEQF) is \(U\), unless \&SYSSEQF is assigned a null character string, in which case the value of the type attribute is O .
2. The value of the count attribute of \(\& S Y S S E Q F\) ( \(K^{\prime} \& S Y S S E Q F\) ) is equal to the number of characters assigned as a value to \&SYSSEQF. If \&SYSSEQF is assigned a null character string, the value of the count attribute is 0 .
3. Throughout the use of a macro definition, the value of \(\&\) SYSSEQF is considered a constant.

\section*{\&SYSSTEP System Variable Symbol}

Use \&SYSSTEP to obtain the stepname of the job step used to assemble your source module. \&SYSSTEP has a global scope.

On CMS and VSE the value of \&SYSSTEP is always (NOSTEP).

\section*{Notes:}
1. The value of the type attribute of \&SYSSTEP (T'\&SYSSTEP) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S S T E P\) ) is the number of characters assigned.

\section*{\&SYSSTMT System Variable Symbol}

Use \&SYSSTMT to obtain the next statement number that is assigned to a statement by the assembler. \&SYSSTMT has a global scope.

The value assigned to \&SYSSTMT is an 8-character string, padded on the left with leading zero ( \(\mathrm{X}^{\prime}\) 'FO') characters. The following example shows the value assigned to \&SYSSTMT. It assumes that the DC statement is in open code, and is the 23rd statement in the source module.
\begin{tabular}{lll}
23 & \(D C\) & \(C^{\prime} \& S Y S S T M T\) \\
+ & \(D C\) & \(C^{\prime} 00000024^{\prime}\)
\end{tabular}

\section*{Notes:}
1. The value of the type attribute of \&SYSSTMT (T'\&SYSSTMT) is always N.
2. The value of the count attribute of \&SYSSTMT (K'\&SYSSTMT) is always 8.

\section*{\&SYSSTYP System Variable Symbol}

Use \&SYSSTYP in a macro definition to generate the type of the current control section. The current control section is the control section in which the macro instruction that calls the definition appears.

The local-scope system variable symbol \&SYSSTYP is assigned a read-only value each time a macro definition is called.

The value assigned is the symbol that represents the type of the current control section in effect when the macro is called. A control section that has been initiated or continued by substitution does not affect the value of \&SYSSTYP for the expansion of the current macro. However, it does affect \&SYSSTYP for a subsequent macro call. Nested macros cause the assembler to assign a value to
\&SYSSTYP that depends on the control section in force inside the calling macro when the inner macro is called.

The control section whose type is assigned to \&SYSSTYP can be defined by a program sectioning statement. This can be a START, CSECT, RSECT, DSECT, or COM statement, or, for the first control section, any instruction described in "First Section" on page 54 Depending upon the instruction used to initiate the current control section, the value assigned to \&SYSSTYP is either CSECT, RSECT, DSECT, or COM. If the current control section is an executable control section initiated by other than a CSECT or RSECT instruction, the value assigned to \&SYSSTYP is CSECT.

If a control section has not been initiated, \&SYSSTYP is assigned a null character string.

\section*{Notes:}
1. The value of the type attribute of \&SYSSTYP ( \(\mathrm{T}^{\prime} \& S Y S S T Y P\) ) is \(U\), unless \&SYSSTYP is assigned a null character string, in which case the value of the type attribute is O .
2. The value of the count attribute of \&SYSSTYP (K'\&SYSSTYP) is equal to the number of characters assigned as a value to \&SYSSTYP. If \&SYSSTYP is assigned a null character string, the value of the count attribute is 0 .
3. Throughout the use of a macro definition, the value of \&SYSSTYP is considered a constant.

\section*{\&SYSTEM_ID System Variable Symbol}

Use \&SYSTEM_ID to obtain the name and release of the operating system under which your source module is being assembled. \&SYSTEM_ID has a global scope.

For example, on MVS, \&SYSTEM_ID might contain one of the following:
z/OS 01.04.00
z/OS 01.05.00
... etc.
on CMS, \&SYSTEM_ID might contain one of the following:
CMS 18
CMS 19
... etc.
on VSE, \&SYSTEM_ID might contain one of the following:
VSE/AF 6.6.0
... etc.

\section*{Notes:}
1. The value of the type attribute of \&SYSTEM_ID (T'\&SYSTEM_ID) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \&\) SYSTEM_ID \(^{2}\) ) is the number of characters assigned.

\section*{\&SYSTERM_DSN System Variable Symbol}

Use \&SYSTERM_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the terminal records.

The local-scope system variable symbol \&SYSTERM_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the MVS operating systems, the value of the character string assigned to \&SYSTERM_DSN is always the value stored in the JFCB for SYSTERM. If SYSTERM is allocated to DUMMY, or a NULLFILE, the value in \&SYSTERM_DSN is NULLFILE.

When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to \&SYSTERM_DSN is determined as follows:

Figure 76. Contents of \&SYSTERM_DSN on CMS
\begin{tabular}{ll}
\hline SYSTERM Allocated To: & Contents of \&SYSTERM_DSN: \\
\hline CMS file & \begin{tabular}{l} 
The 8-character filename, the \\
8-character filetype, and the \\
2-character filemode of the file, each \\
separated by a space
\end{tabular} \\
\hline Dummy file (no physical I/O) & DUMMY \\
\hline Printer & PRINTER \\
\hline Labeled tape file & The data set name of the tape file \\
\hline Unlabeled tape file & \begin{tabular}{l} 
TAPn, where \(n\) is a value from 0 to 9, \\
\\
or A to \(F\).
\end{tabular} \\
\hline Terminal & TERMINAL \\
\hline
\end{tabular}

On VSE, the value of the character string assigned to \&SYSTERM_DSN is always SYSLOG.

\section*{Examples:}

On MVS, \&SYSTERM_DSN might be assigned a value such as:
IBMAPC.IBMAPCA.JOB06734.D0000104.?
On CMS, \&SYSTERM_DSN might be assigned a value such as:
TERMINAL

\section*{Notes:}
1. If the TERM user exit provides the terminal data set information then the value in \&SYSTERM_DSN is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSTERM_DSN (T'\&SYSTERM_DSN) is always U.
3. The value of the count attribute of \&SYSTERM_DSN (K'\&SYSTERM_DSN) is equal to the number of characters assigned as a value to \&SYSTERM_DSN.

\title{
\&SYSTERM_MEMBER System Variable Symbol \\ VSE The value of \&SYSTERM_MEMBER is always null.
}

The value of the type attribute is O , and the value of the count attribute is 0 .
```

vSE

```

CMS, MVS You can use \&SYSTERM_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the terminal records.

The local-scope system variable symbol \&SYSTERM_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the terminal records is not an MVS partitioned data set, \&SYSTERM_MEMBER is assigned a null character string.

\section*{CMS, MVS}

Notes:
1. If the TERM user exit provides the terminal data set information then the value in \&SYSTERM_MEMBER is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSTERM_MEMBER ( \(T\) '\&SYSTERM_MEMBER) is \(U\), unless \&SYSTERM_MEMBER is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSTERM_MEMBER ( \(\mathrm{K}^{\prime}\) \&SYSTERM_MEMBER) is equal to the number of characters assigned as a value to \&SYSTERM_MEMBER. If \&SYSTERM_MEMBER is assigned a null character string, the value of the count attribute is 0 .

\section*{\&SYSTERM_VOLUME System Variable Symbol}

VSE The value of \&SYSTERM_VOLUME is always null.
The value of the type attribute is U , and the value of the count attribute is 0 .

\section*{VSE}

CMS, MVS You can use \&SYSTERM_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler is writing the terminal records.

The local-scope system variable symbol \&SYSTERM_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the terminal records are being written to a Shared File System CMS file, \&SYSTERM_VOLUME is assigned the value ** SFS.

If the volume on which the data set resides is not labeled, \&SYSTERM_VOLUME is assigned a null character string. \(\qquad\)

\section*{Notes:}
1. If the TERM user exit provides the terminal data set information then the value in \&SYSTERM_VOLUME is the value extracted from the Exit-Specific Information block described in the HLASM Programmer's Guide.
2. The value of the type attribute of \&SYSTERM_VOLUME ( \(T^{\prime} \& S Y S T E R M \_V O L U M E\) ) is \(U\), unless \&SYSTERM_VOLUME is assigned a null character string, in which case the value of the type attribute is O .
3. The value of the count attribute of \&SYSTERM_VOLUME ( \(K^{\prime} \& S Y S T E R M \_V O L U M E\) ) is equal to the number of characters assigned as a value to \&SYSTERM_VOLUME. If \&SYSTERM_VOLUME is assigned a null character string, the value of the count attribute is 0 . The maximum length of this system variable symbol is 6 .

\section*{\&SYSTIME System Variable Symbol}

Use \&SYSTIME to obtain the time at which your source module is assembled. It has local scope, but can be used in open code. It is assigned a read-only value.

The value of \&SYSTIME is a 5 -character string in the format:
HH.MM
where:
HH is two-digit field that gives the hour of the day. It has a value between 00 and 23 , inclusive.
\(M M\) is two-digit field that gives the minute of the hour. It has a value between 00 and 59 , inclusive. It is separated from HH by a period.

\section*{Example:}
09.45

\section*{Notes:}
1. The time corresponds to the time printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of \&SYSTIME (T'\&SYSTIME) is always \(U\).
3. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S T I M E\) ) is always 5.

\section*{\&SYSVER System Variable Symbol}

Use \&SYSVER to obtain the version, release, and modification level of the assembler being used to assemble your source module. \&SYSVER has a global scope. For example, when IBM High Level Assembler for MVS \& VM \& VSE Release 5.0 is used, \&SYSVER has the value 1.5.0.

\section*{Notes:}
1. The value of the type attribute of \&SYSVER ( \(T^{\prime} \& S Y S V E R\) ) is always \(U\).
2. The value of the count attribute ( \(\mathrm{K}^{\prime} \& S Y S V E R\) ) is the number of characters assigned. In the above example, the count attribute of \&SYSVER is 5.

\section*{Chapter 8. How to Write Macro Instructions}

This chapter describes macro instructions; where you can use them and how you specify them.

The first section on page 297 describes the macro instruction format, including details on the name, operation, and operand entries, and what is generated as a result of a macro instruction.
"Sublists in Operands" on page 304 describes how you can use sublists to specify several values in an operand entry.
"Values in Operands" on page 307 describes the values you can specify in an operand entry when you call a macro definition.
"Nesting Macro Instruction Definitions" on page 311 describes how you can use nested macro call instructions to call macros from within a macro.

What is a Macro Instruction: The macro instruction provides the assembler with:
- The name of the macro definition to process
- The information or values to pass to the macro definition

This information is the input to a macro definition. The assembler uses the information either in processing the macro definition, or for substituting values into model statements in the definition.

The output from a macro definition can be:
- A sequence of statements generated from the model statements of the macro for further processing at assembly time.
- Values assigned to global-scope SET symbols. These values can be used in other macro definitions and in open code (see "SET Symbols" on page 319.

Where Macro Instructions Can Appear: A macro instruction can be written anywhere in your program, provided the assembler can find the macro definition. The macro definition can be found either in a macro library, or in the source program before the macro instruction, or be provided by a LIBRARY user exit. However, the statements generated from the called macro definition must be valid assembler language instructions and allowed where the calling macro instruction appears.

\section*{Macro Instruction Format}

name_field
is a special positional operand that can be used to pass a value into the called macro definition. For a detailed description of what form name_entry can take, see "Name Entry" on page 299.

\section*{Macro Instruction Format}
sequence_symbol
is a sequence symbol. If a sequence symbol is coded in the name entry of a macro instruction, the value of the symbol is not passed to the called macro definition and therefore cannot be used as a value for substitution in the macro definition.
operation_code
is the symbolic operation code which identifies the macro definition that you want the assembler to process. For more information, see "Operation Entry" on page 299.
operand
The positional operands or keyword operands that you use to pass values into the called macro definition. For more information, see "Operand Entry" on page 300 .

If no operands are specified in the operand field, and if the absence of the operand entry is indicated by a comma preceded and followed by one or more spaces, remarks are allowed.

The entries in the name, operation, and operand fields correspond to entries in the prototype statement of the called macro definition (see "Macro Instruction Prototype" on page 245.

\section*{Alternative Formats for a Macro Instruction}

A macro instruction can be specified in one of the three following ways:
- The normal way, with the operands preceding any remarks
- The alternative way, allowing remarks for each operand
- A combination of the first two ways

The following example show the normal statement format (NAME1), the alternative statement format (NAME2), and a combination of both statement formats (NAME3).
\begin{tabular}{llll} 
& \begin{tabular}{l} 
Opera- \\
tion
\end{tabular} Operand & Comment & Cont. \\
Name & OP1 & OPERAND1,0PERAND2,OPERAND3
\end{tabular} \begin{tabular}{l} 
This is the normal \\
statement format
\end{tabular}\(\quad\) X

\section*{Notes:}
1. Any number of continuation lines are allowed. However, each continuation line must be indicated by a non-space character in the column after the end column of the previous statement line (see "Continuation Lines" on page 15).
2. If the DBCS assembler option is specified, the continuation features outlined in "Continuation of double-byte data" on page 16 apply to continuation in the macro language. Extended continuation may be useful if a macro operand contains double-byte data.
3. Operands on continuation lines must begin in the continue column (column 16), or the assembler assumes that the current line and any lines that follow contain remarks.

If any entries are made in the columns before the continue column in continuation lines, the assembler issues an error message and the whole statement is not processed.
4. One or more spaces must separate the operand from the remarks.
5. A comma after an operand indicates more operands follow.
6. The last operand requires no comma following it, but using a comma does not cause an error.
7. You do not need to use the same format when you code a macro instruction as you use when you code the corresponding macro prototype statement.
8. Continued comments for a macro with an operand list that terminates in a null operand will be recognized provided each continued comment begins in the same or later column as the preceding line's comment.

\section*{Name Entry}

Use the name entry of a macro instruction to:
- Pass a value into a macro definition through the name entry declared in the macro definition
- Provide a conditional assembly label (see "Sequence Symbols" on page 339) so that you can branch to the macro instruction during conditional assembly if you want the called macro definition expanded.

The name entry of a macro instruction can be:
- Space
- An ordinary symbol, such as HERE
- A variable symbol, such as \&A.
- Any combination of variable symbols and other character strings concatenated together, such as HERE.\&A
- Any character string allowed in a macro instruction operand, such as 'Now is the hour' or STRING00, excluding sublist entries and certain attribute references (see "Values in Operands" on page 307)
- A sequence symbol, which is not passed to the macro definition, such as .SEQ

\section*{Operation Entry}

The operation entry is the symbolic name of the operation code that identifies a macro definition to process.

The operation entry must be a valid symbol, and must be identical to the operation field in the prototype statement of the macro definition.

The assembler searches for source macro definitions before library macro definitions. If you have a source macro definition that has the same name as a library macro definition, the assembler only processes the source macro definition.

\section*{Macro Instruction Format}

You can use a variable symbol as a macro instruction. For example if MAC1 has been defined as a macro, you can use the following statements to call it:
```

\&CALL SETC
'MAC1'
\&CALL

```

You cannot use a variable symbol as a macro instruction that passes operands to the macro. The second statement in the following example generates an error:
```

\&CALL SETC
\&CALL
'MAC1 OPERAND1=VALUE'

```

You must specify operand entries after the variable symbol, as shown in the following example:
```

\&CALL SETC 'MAC1'
\&CALL OPERAND1=VALUE

```

\section*{Operand Entry}

Use the operand entry of a macro instruction to pass values into the called macro definition. These values can be passed through:
- The symbolic parameters you have specified in the macro prototype, or
- The system variable symbol \&SYSLIST if it is specified in the body of the macro definition (see "\&SYSLIST System Variable Symbol" on page 276 .

The two types of operands allowed in a macro instruction are positional and keyword operands. You can specify a sublist with multiple values in both types of operands. Special rules for the various values you can specify in operands are also given below.

\section*{Positional Operands}

You can use a positional operand to pass a value into a macro definition through the corresponding positional parameter declared for the definition. You should declare a positional parameter in a macro definition when you want to change the value passed at every call to that macro definition.

You can also use a positional operand to pass a value to the system variable symbol \&SYSLIST. If \&SYSLIST, with the applicable subscripts, is specified in a macro definition, you do not need to declare positional parameters in the prototype statement of the macro definition. You can thus use \&SYSLIST to refer to any positional operand. This allows you to vary the number of operands you specify each time you call the same macro definition.

The positional operands of a macro instruction must be specified in the same order as the positional parameters declared in the called macro definition.

Each positional operand constitutes a character string. This character string is the value passed through a positional parameter into a macro definition.

The specification for each positional parameter in the prototype statement definition must be a valid variable symbol. Values are assigned (see 1 in Figure 77 on page 301 to the positional operands by the corresponding positional arguments (see 2 in Figure 77) specified in the macro instruction that calls the macro definition.
\begin{tabular}{|c|c|c|c|}
\hline \multirow{3}{*}{Macro Definition} & \multicolumn{3}{|l|}{Source Module} \\
\hline & \multicolumn{3}{|l|}{MACRO} \\
\hline & \[
\begin{aligned}
& \text { POSPAR } \\
& \cdot \\
& \text { MEND }
\end{aligned}
\] &  & \\
\hline \begin{tabular}{l}
Macro \\
Instruction
\end{tabular} & \begin{tabular}{l}
START \\
POSPAR \\
END
\end{tabular} &  & 2 \\
\hline
\end{tabular}

Figure 77. Positional Operands

\section*{Notes:}
1. An omitted operand has a null character value.
2. Each positional operand can be up to 1024 characters long.
3. If the DBCS assembler option is specified, the positional operand can be a string containing double-byte data. The string need not be quoted.

The following are examples of macro instructions with positional operands:
\begin{tabular}{ll} 
MACCALL & VALUE,9,8 \\
MACCALL & \&A,'QUOTED STRING' \\
MACCALL & EXPR+2, SYMBOL \\
MACCALL & (A,B,C,D,E),(1,2,3,4) \\
MACCALL & \(\& A\), ' \(^{\prime}\), S.T.R.I.N.G>'
\end{tabular}

The following list shows what happens when the number of positional operands in the macro instruction is equal to or differs from the number of positional parameters declared in the prototype statement of the called macro definition:
Equal Valid, if operands are correctly specified.
Greater than Meaningless, unless \&SYSLIST is specified in definition to refer to excess operands.

Less than Omitted operands give null character values to corresponding parameters (or \&SYSLIST specification).

\section*{Keyword Operands}

You can use a keyword operand to pass a value through a keyword parameter into a macro definition. The values you specify in keyword operands override the default values assigned to the keyword parameters. The default value should be a value you use frequently. Thus, you avoid having to write this value every time you code the calling macro instruction.

\section*{Macro Instruction Format}

When you need to change the default value, you must use the corresponding keyword operand in the macro instruction. The keyword can indicate the purpose for which the passed value is used.

Any keyword operand specified in a macro instruction must correspond to a keyword parameter in the macro definition called. However, keyword operands do not have to be specified in any particular order.

The general specifications for symbolic parameters also apply to keyword operands. The actual operand keyword must be a valid variable symbol. A null character string can be specified as the standard value of a keyword operand, and is generated if the corresponding keyword operand is omitted.

A keyword operand must be coded in this format:
KEYWORD=VALUE
where:
KEYWORD has up to 62 characters without an ampersand.
VALUE can be up to 1024 characters.
The corresponding keyword parameter in the called macro definition is specified as: \&KEYWORD=DEFAULT

If a keyword operand is specified, its value overrides the default value specified for the corresponding keyword parameter.

If the DBCS assembler option is specified, the keyword operand can be a string containing double-byte data. The string need not be quoted.

If the value of a keyword operand is a literal, two equal signs must be specified.
The following examples of macro instructions have keyword operands:
\begin{tabular}{ll} 
MACKEY & KEYWORD \(=(A, B, C, D, E)\) \\
MACKEY & KEY1=1,KEY2=2,KEY3=3 \\
MACKEY & KEY3=2000,KEY1=0,KEYWORD=HALLO \\
MACKEY & KEYWORD='<.S.T.R.I.N.G>' \\
MACKEY & KEYWORD \(==C^{\prime}\) STRING'
\end{tabular}

To summarize the relationship of keyword operands to keyword parameters:
- The keyword of the operand corresponds (see \(\mathbf{1}\) in Figure 78 on page 303 to a keyword parameter. The value in the operand overrides the default value of the parameter.
- If the keyword operand is not specified (see \(\mathbf{2}\) in Figure 78), the default value of the parameter is used.
- If the keyword of the operand does not correspond (see \(\mathbf{3}\) in Figure 78) to any keyword parameter, the assembler issues an error message, but the macro is generated using the default values of the other parameters.
- The default value specified for a keyword parameter can be the null character string (see 4 in Figure 78). The null character string is a character string with a length of zero; it is not a space, because a space occupies one character position.


Figure 78. Relationship between Keyword Operands and Keyword Parameters and Their Assigned Values

\section*{Combining Positional and Keyword Operands}

You can use positional and keyword operands in the same macro instruction. Use a positional operand for a value that you change often, and a keyword operand for a value that you change infrequently.

Positional and keyword parameters can be mixed freely in the macro prototype statement (see 1 in Figure 79). The same applies to the positional and keyword operands of the macro instruction (see 2 in Figure 79). Note, however, that the order in which the positional parameters appear (see 3 in Figure 79) determines
the order in which the positional operands must appear. Interspersed keyword parameters and operands (see 4 in Figure 79) do not affect this order.


Figure 79. Combining Positional and Keyword Parameters
\&SYSLIST( \(n\) ): The system variable symbol \&SYSLIST( \(n\) ) refers only to the positional operands in a macro instruction.

\section*{Sublists in Operands}

You can use a sublist in a positional or keyword operand to specify several values. A sublist is a character string that consists of one or more entries separated by commas and enclosed in parentheses.

If the COMPAT(SYSLIST) assembler option is not specified, a variable symbol that has been assigned a character string that consists of one or more entries separated by commas and enclosed in parentheses is also treated as a sublist. However, if the COMPAT(SYSLIST) assembler option is specified, a sublist assigned to a variable symbol is treated as a character string, not as a sublist.

A variable symbol is not treated as a sublist if the parentheses are not present. The following example shows two calls to macro MAC1. In the first call, the value of the operand in variable \&VAR1 is treated as a sublist. In the second call, the value of the operand is treated as a character string, not a sublist, because the variable \&VAR2 does not include parentheses.
\begin{tabular}{lll} 
\&VAR1 & SETC & \(\quad(1,2)^{\prime}\) \\
& MAC1 & KEY=\&VAR1 \\
\&VAR2 & SETC & \(11,2^{\prime}\) \\
& MAC1 & KEY \(=(\& V A R 2)\)
\end{tabular}

To refer to an entry of a sublist code, use:
- The corresponding symbolic parameter with an applicable subscript, or
- The system variable symbol \&SYSLIST with applicable subscripts, the first of which refers to the positional operand, and the second to the sublist entry in the operand. \&SYSLIST can refer only to sublists in positional operands.

Figure 80 on page 305 shows that the value specified in a positional or keyword operand can be a sublist.

A symbolic parameter can refer to the whole sublist (see \(\mathbf{1}\) in Figure 80), or to an individual entry of the sublist. To refer to an individual entry, the symbolic parameter (see \(\mathbf{2}\) in Figure 80) must have a subscript whose value indicates the position (see 3 in Figure 80) of the entry in the sublist. The subscript must have a value greater than or equal to 1 .

A sublist, including the enclosing parentheses, must not contain more than 1024 characters. It consists of one or more entries separated by commas and enclosed in parentheses; for example, ( \(\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\) ). () is a valid sublist with the null character string as the only entry.


Figure 80. Sublists in Operands
Figure 81 shows the relationship between subscripted parameters and sublist entries if:
- A sublist entry is omitted (see 1 in Figure 81).
- The subscript refers past the end of the sublist (see 2 in Figure 81 ).
- The value of the operand is not a sublist (see \(\mathbf{3}\) in Figure 81).
- The parameter is not subscripted (see 4 in Figure 81).
\&SYSLIST( \(n, m\) ): The system variable symbol, \&SYSLIST \((n, m)\), can also refer to sublist entries, but only if the sublist is specified in a positional operand.

Figure 81. Relationship between Subscripted Parameters and Sublist Entries
\begin{tabular}{|c|c|c|}
\hline Parameter & Sublist specified in corresponding operand or as default value of a keyword parameter & Value generated or used in computation \\
\hline 1 \&PARM1 (3) & \((1,2,4)\) & Null character string \\
\hline 2 \&PARM1 (5) & \((1,2,3,4)\) & Null character string \\
\hline \begin{tabular}{l}
\&PARM1 \\
3 \\
\&PARM1 (1) \\
\&PARM1 (2)
\end{tabular} & \[
\begin{aligned}
& \text { A } \\
& \text { A } \\
& \text { A }
\end{aligned}
\] & \begin{tabular}{l}
A \\
A \\
Null character string
\end{tabular} \\
\hline \begin{tabular}{l}
4 \&PARM1 \\
\&PARM1 (1) \\
2 \&PARM1(2) \\
\&PARM1 \\
\&PARM1 (1) \\
\&PARM1 (2)
\end{tabular} & \begin{tabular}{l}
(A) \({ }^{1}\) \\
(A) \({ }^{1}\) \\
(A) \({ }^{1}\) \\
() \({ }^{1}\) \\
() \({ }^{1}\) \\
() \({ }^{1}\)
\end{tabular} & \begin{tabular}{l}
(A) \\
A \\
Null character string \\
() \\
Null character string \\
Null character string
\end{tabular} \\
\hline \begin{tabular}{l}
\&PARM1 (2) \\
\&PARM1 (1)
\end{tabular} & \[
\begin{aligned}
& (A,, C, D)^{2} \\
& ()^{2}
\end{aligned}
\] & \begin{tabular}{l}
Nothing \({ }^{3}\) \\
Nothing \({ }^{3}\)
\end{tabular} \\
\hline \begin{tabular}{l}
\&PARM1 \\
\&PARM2 (3) \\
\&SYSLIST \((2,3)\)
\end{tabular} & \[
\begin{aligned}
& A,(1,2,3,4)^{4} \\
& A,(1,2,3,4)^{4} \\
& A,(1,2,3,4)^{4} \\
& \hline
\end{aligned}
\] & \[
\begin{aligned}
& \text { A } \\
& 3 \\
& 3
\end{aligned}
\] \\
\hline
\end{tabular}

Notes:
1. Considered a sublist.
2. The space indicates the end of the operand field.
3. Produces error diagnostic message ASMA088E Unbalanced parentheses in macro call operand.
4. Positional operands.

\section*{Multilevel Sublists}

You can specify multilevel sublists (sublists within sublists) in macro operands. The depth of this nesting is limited only by the constraint that the total operand length must not exceed 1024 characters. Inner elements of the sublists are referenced using additional subscripts on symbolic parameters or on \&SYSLIST.
\(\mathrm{N}^{\prime}\) \&SYSLIST( \(n\) ) gives the number of operands in the indicated \(n\)-th level sublist. The number attribute ( \(\mathrm{N}^{\prime}\) ) and a parameter name with an \(n\)-element subscript array gives the number of operands in the indicated \((n+1)\)-th operand sublist. Figure 82 shows the value of selected elements if \&P is the first positional parameter, and the value assigned to it in a macro instruction is ( \(\mathrm{A},(\mathrm{B},(\mathrm{C})\) ), D\()\).

Figure 82. Multilevel Sublists
\begin{tabular}{|c|c|c|}
\hline Selected Elements from \& P & Selected Elements from \&SYSLIST & Value of Selected Element \\
\hline \& \(P\) & \&SYSLIST(1) & (A, (B, (C)), D) \\
\hline \&P(1) & \&SYSLIST(1,1) & A \\
\hline \&P(2) & \&SYSLIST(1,2) & (B,(C)) \\
\hline \& \(\mathrm{P}(2,1)\) & \&SYSLIST(1,2,1) & B \\
\hline \& \(\mathrm{P}(2,2)\) & \&SYSLIST(1,2,2) & (C) \\
\hline \& \(\mathrm{P}(2,2,1)\) & \&SYSLIST(1,2,2,1) & C \\
\hline \& \(\mathrm{P}(2,2,2)\) & \&SYSLIST(1,2,2,2) & null \\
\hline N'\&P(2,2) & N'\&SYSLIST(1,2,2) & 1 \\
\hline N'\&P(2) & N'\&SYSLIST(1,2) & 2 \\
\hline N'\&P(3) & N'\&SYSLIST \((1,3)\) & 1 \\
\hline N'\&P & N'\&SYSLIST(1) & 3 \\
\hline
\end{tabular}

\section*{Passing Sublists to Inner Macro Instructions}

You can pass a suboperand of an outer macro instruction sublist as a sublist to an inner macro instruction. However, if you specify the COMPAT(SYSLIST) assembler option, a sublist assigned to a variable symbol is treated as a character string, not as a sublist.

\section*{Values in Operands}

You can use a macro instruction operand to pass a value into the called macro definition. The two types of value you can pass are:
- Explicit values or the actual character strings you specify in the operand
- Implicit values, or the attributes inherent in the data represented by the explicit values

The explicit value specified in a macro instruction operand is a character string that can contain zero or more variable symbols.

The character string must not be greater than 1024 characters after substitution of values for any variable symbols. This includes a character string that constitutes a sublist.

The character string values in the operands, including sublist entries, are assigned to the corresponding parameters declared in the prototype statement of the called macro definition. A sublist entry is assigned to the corresponding subscripted parameter.

\section*{Omitted Operands}

When a keyword operand is omitted, the default value specified for the corresponding keyword parameter is the value assigned to the parameter. When a positional operand or sublist entry is omitted, the null character string is assigned to the parameter.

\section*{Notes:}
1. Spaces appearing between commas (without surrounding single quotation marks) do not signify an omitted positional operand or an omitted sublist entry; they indicate the end of the operand field.
2. Adjacent commas indicate omission of positional operands; no comma is needed to indicate omission of the last or only positional operand.

The following example shows a macro instruction preceded by its corresponding prototype statement. The macro instruction operands that correspond to the third and sixth operands of the prototype statement are omitted in this example.
```

EXAMPLE
EXAMPLE 17,*+4,,AREA,FIELD(6)
macro prototype
macro instruction

```

\section*{Unquoted Operands}

The assembler normally retains the case of unquoted macro operands. However, to maintain uppercase alphabetic character set compatibility with earlier assemblers, High Level Assembler provides the COMPAT(MACROCASE) assembler option. When you specify this option, the assembler internally converts lowercase alphabetic characters (a through z) in unquoted macro instruction operands to uppercase alphabetic characters (A though Z), before macro expansion begins.

\section*{Special Characters}

Any of the 256 characters of the EBCDIC character set can appear in the value of a macro instruction operand (or sublist entry). However, the following characters require special consideration:

\section*{Ampersands}

A single ampersand indicates the presence of a variable symbol. The assembler substitutes the value of the variable symbol into the character string specified in a macro instruction operand. The resultant string is then the value passed into the macro definition. If the variable symbol is undefined, an error message is issued.

Double ampersands must be specified if a single ampersand is to be passed to the macro definition.

\section*{Examples:}
\&VAR
\(\& A+\& B+3+\& C * 10\)
' \&MESSAGE'
\&\&REGISTER

\section*{Single Quotation Marks}

A single quotation mark is used:
- To indicate the beginning and end of a quoted string
- In a length, type, integer, opcode, or scale attribute reference notation that is not within a quoted string

\section*{Examples:}
```

'QUOTED STRING'
L'SYMBOL
T'SYMBOL

```

\section*{Shift-out (SO) and Shift-in (SI)}

If the DBCS assembler option is specified, then SO (X'0E') and SI (X'0F') are recognized as shift codes. SO and SI delimit the start and end of double-byte data respectively.

\section*{Quoted Strings and Character Strings}

A "quoted string" is any sequence of characters that begins and ends with a single quotation mark (compare with conditional assembly character expressions described in "Character (SETC) Expressions" on page 371.

Two single quotation marks must be specified inside each quoted string. This includes substituted single quotation marks.

A "character string" is a sequence of characters, not delimited with quotes. So a character string is the contents of a quoted string.

Quoted strings can contain double-byte data, if the DBCS assembler option is specified. The double-byte data must be bracketed by the SO and SI delimiters. Only valid double-byte data is recognized between the SO and SI. The SI may be in any odd-numbered byte position after the SO. If the end of the operand is reached before SI is found, then error ASMA203E Unbalanced double-byte delimiters is issued.

Macro instruction operands can have values that include one or more quoted strings. Each quoted string can be separated from the following quoted string by one or more characters, and each must contain an even number of single quotation marks.

\section*{Examples:}

11
'L'SYMBOL'
'QUOTE1 'AND 'QUOTE2'
\(A^{\prime} B^{\prime} C\)

\section*{Attribute Reference Notation}

You can specify an attribute reference notation as a macro instruction operand value. The attribute reference notation must be preceded by a space or any other special character except the ampersand and the single quotation mark. See "Data Attributes" on page 324 for details about data attributes, and the format of attribute references.

\section*{Examples:}
MAC1
L'SYMBOL, 10+L'AREA*L'FIELD
MAC1 I'PACKED-S'PACKED

\section*{Parentheses}

In macro instruction operand values, there must be an equal number of left and right parentheses. They must be paired, that is, each left parenthesis needs a following right parenthesis at the same level of nesting. An unpaired (single) left or right parenthesis can appear only in a quoted string.

\section*{Examples:}
(PAIRED-PARENTHESES)
()
(A(B)C)D(E)
(IN'('STRING)

\section*{Spaces}

One or more spaces outside a quoted string indicates the end of the operands of a macro instruction. Thus spaces should only be used inside quoted strings.

\section*{Example:}
'SPACES ALLOWED'

\section*{Commas}

A comma outside a quoted string indicates the end of an operand value or sublist entry. Commas that do not delimit values can appear inside quoted strings or paired parentheses that do not enclose sublists.

\section*{Examples:}

A, B, C, D
\((1,2) 3^{\prime} 5,6^{\prime}\)

\section*{Equal Signs}

An equal sign can appear in the value of a macro instruction operand or sublist entry:
- As the first character
- Inside quoted strings
- Between paired parentheses
- In a keyword operand
- In a positional operand, provided the parameter does not resemble a keyword operand

\section*{Examples:}
\(=H^{\prime} 201^{\prime}\)
A' = 'B
\(C(A=B)\)
\(2 X=B\)
\(K E Y=A=B\)
The assembler issues a warning message for a positional operand containing an equal sign, if the operand resembles a keyword operand. Thus, if we assume that the following is the prototype of a macro definition:

MAC1 \&F
the following macro instruction generates a warning message:
```

MAC1 K=L (K appears to be a valid keyword)

```
while the following macro instruction does not:
```

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

```

\section*{Periods}

A period (.) can be used in the value of an operand or sublist entry. It is passed as a period. However, if it is used immediately after a variable symbol, it becomes a concatenation character. Two periods are required if one is to be passed as a character.

\section*{Examples:}
3.4
\&A. 1
\&A.. 1

\section*{Nesting Macro Instruction Definitions}

A nested macro instruction definition is a macro instruction definition you can specify as a set of model statements in the body of an enclosing macro definition. This lets you create a macro definition by expanding the outer macro that contains the nested definition.

Note that all nested inner macro definitions are effectively "black boxes": there is no visibility to the outermost macro definition of any variable symbol or sequence symbol within any of the nested macro definitions. This means that you cannot use an enclosing macro definition to tailor or parameterize the contents of a nested inner macro definition.

This lack of parameterization can be overcome in some cases by using the AINSERT statement. This lets you generate a macro definition from within another macro generation. A simple example is shown at "Where to Define a Macro in a Source Module" on page 243 In Figure 83 on page 312 macro ainsert_test_macro generates the macro mac1 using a combination of AINSERT and AREAD instructions. The mac1 macro is then called with a list of seven parameters.
\begin{tabular}{|c|c|}
\hline 1 & macro \\
\hline 2 \&name & ainsert_test_macro \\
\hline 3 & ainsert \({ }^{-}\)' Macro', back \\
\hline 4 & ainsert ' macl',back \\
\hline 5 & ainsert 'Blah blah blah',front \\
\hline 6 \&aread & aread \\
\hline 7 \&aread & setc '\&aread' \((1,10)\) \\
\hline 8 & ainsert '\&\&n seta n''\&\&syslist ',back \\
\hline 9 & ainsert ' dc a(\&\&n)',back \\
\hline 10 & ainsert ' dc c''\&aread'' ',back \\
\hline 11 & ainsert ' mend',back \\
\hline 12 & mend \\
\hline \multicolumn{2}{|l|}{13 *} \\
\hline \multicolumn{2}{|l|}{14 testains csect 0} \\
\hline \multicolumn{2}{|l|}{15 *} \\
\hline 16 & ainsert_test_macro \\
\hline 17+ & ainsert \({ }^{-}\)' Macro',back \\
\hline 18+ & ainsert ' macl',back \\
\hline 19+ & ainsert 'Blah blah blah',front \\
\hline \multicolumn{2}{|l|}{20-Blah blah blah} \\
\hline 21+ & ainsert '\&\&n seta n''\&\&syslist ',back \\
\hline 22+ & ainsert ' dc a(\&\&n)',back \\
\hline 23+ & ainsert ' dc c''Blah blah '' ',back \\
\hline 24+ & ainsert ' mend',back \\
\hline 25> & Macro \\
\hline 26> & mac1 \\
\hline \(27>8 n\) & seta n'\&syslist \\
\hline 28> & dc a (\&n) \\
\hline 29> & dc c'Blah blah ' \\
\hline 30> & mend \\
\hline \multicolumn{2}{|l|}{31 *} \\
\hline 32 & macl \(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}\) \\
\hline 33+ & dc a(7) \\
\hline 34+ & dc c'Blah blah ' \\
\hline 35 * & \\
\hline 36 & end \\
\hline
\end{tabular}

Figure 83. Expanding Nested Macro Definitions

\section*{Inner and Outer Macro Instructions}

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

\section*{Levels of Macro Call Nesting}

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

The \&SYSNEST system variable indicates how many levels you called. It has the value 1 in an outer macro, and is incremented by one at a macro call.

\section*{Recursion}

You can also call a macro definition recursively; that is, you can write macro instructions inside macro definitions that are calls to the containing definition. This is how you define macros to process recursive functions.

\section*{General Rules and Restrictions}

Macro instruction statements can be written inside macro definitions. Values are substituted in the same way as they are for the model statements of the containing macro definition. The assembler processes the called macro definition, passing to it the operand values (after substitution) from the inner macro instruction. In addition to the operand values described in "Values in Operands" on page 307 nested macro calls can specify values that include:
- Any of the symbolic parameters (see \(\mathbf{1}\) in Figure 84 ) specified in the prototype statement of the containing macro definition
- Any SET symbols (see \(\boldsymbol{2}\) in Figure 84) declared in the containing macro definition
- Any of the system variable symbols such as \&SYSDATE, \&SYSTIME, etc. (see 3 in Figure 84).


Figure 84. Values in Nested Macro Calls
The number of nesting levels permitted depends on the complexity and size of the macros at the different levels; that is, the number of operands specified, the number of local-scope and global-scope SET symbols declared, and the number of sequence symbols used.

When the assembler processes a macro exit instruction, either MEXIT or MEND, it selects the next statement to process depending on the level of nesting. If the macro exit instruction is from an inner macro, the assembler processes the next statement after the statement that called the outer macro. The next statement in open code might come from the AINSERT buffer. If the macro exit instruction is from an outer macro, the assembler processes the next statement in open code, after the statement that called the inner macro.

\section*{Passing Values through Nesting Levels}

The value contained in an outer macro instruction operand can be passed through one or more levels of nesting (see Figure 85 on page 315). However, the value specified (see 1 in Figure 85) in the inner macro instruction operand must be identical to the corresponding symbolic parameter (see \(\mathbf{2}\) in Figure 85) declared in the prototype of the containing macro definition.

Thus, a sublist can be passed (see \(\mathbf{3}\) in Figure 85) and referred to (see 4 in Figure 85) as a sublist in the macro definition called by the inner macro call. Also, any symbol (see 5 in Figure 85) that is passed carries its attribute values through the nesting levels.

If inner macro calls at each level are specified with symbolic parameters as operand values, values can be passed from open code through several levels of macro nesting.

COMPAT(SYSLIST) Assembler Option: If the COMPAT(SYSLIST) assembler option is specified, and a symbolic parameter is only a part of the value specified in an inner macro instruction operand, only the character string value given to the parameter by an outer call is passed through the nesting level. Inner sublist entries are, therefore, not available for reference in the inner macro.


\section*{Notes:}
1. The following inner macro call statement is generated, but not listed unless the PCONTROL(MCALL) option is specified, or the assembler instruction ACONTROL MCALL is active:

INNER (AREA, F200, SUM), TO, FROM
Figure 85. Passing Values Through Nesting Levels

\section*{System Variable Symbols in Nested Macros}

The fixed global-scope system variable symbols (see "System Variable Symbols" on page 262) are not affected by the nesting of macros. The variable global-scope system variable symbols have values which may change during the expansion of a macro definition. The following system variable is influenced by nested macros:
\&SYSM_SEV Provides the highest MNOTE severity code from the nested macro most recently called.

The local system variable symbols are given read-only values each time a macro definition is called.

The following system variable symbols can be affected by the position of a macro instruction in code or the operand value specified in the macro instruction:
\&SYSCLOCK The assembler assigns \&SYSCLOCK the constant string value representing the TOD clock value at the time at which a macro call is made. The time portion of this value is precise to the microsecond. For any inner macro call, the value assigned to \&SYSCLOCK differs from that of its parent.
\&SYSECT The assembler gives \&SYSECT the character string value of the name of the control section in use at the point at which a macro call is made. For a macro definition called by an inner macro call, the assembler assigns to \&SYSECT the name of the control section in effect in the macro definition that contains the inner macro call, at the time the inner macro is called.

If no control section is generated within a macro definition, the value assigned to \&SYSECT does not change. It is the same for the next level of macro definition called by an inner macro instruction.
\&SYSLIB_DSN, \(\boldsymbol{\&}\) SYSLIB_MEMBER, \(\boldsymbol{\&}\) SYSLIB_VOLUME
The assembler assigns the character string value of the \&SYSLIB system variable symbols at the point at which a macro is called. For an inner macro call whose definition is from a library member, these values may differ, if this is the first time this macro is invoked.
\&SYSLIST If \(\&\) SYSLIST is specified in a macro definition called by an inner macro instruction, \&SYSLIST refers to the positional operands of the inner macro instruction.
\&SYSLOC The assembler gives \&SYSLOC the character string value of the name of the location counter in use at the point at which a macro is called. For a macro definition called by an inner macro call, the assembler assigns to \&SYSLOC the name of the location counter in effect in the macro definition that contains the inner macro call. If no LOCTR or control section is generated within a macro definition, the value assigned to \&SYSLOC does not change. It is the same for the next level of macro definition called by an inner macro instruction.
\&SYSNDX The assembler increments \&SYSNDX by one each time it encounters a macro call. It retains the incremented value throughout the expansion of the macro definition called, that is, within the local scope of the nesting level.
\&SYSNEST The assembler increments \&SYSNEST by one each time it encounters a nested macro instruction. It retains the incremented value within the local scope of the macro definition called by the inner macro instruction. Subsequent nested macro instructions cause \&SYSNEST to be incremented by 1 . When the assembler exits from a nested macro it decreases the value in \&SYSNEST by 1.
\&SYSSEQF The assembler assigns \&SYSSEQF the character string value of the identification-field of the outer-most macro instruction statement. The value of \(\& S Y S S E Q F\) remains constant throughout the expansion of the called macro definition and all macro definitions called from within the outer macro.
\&SYSSTYP The assembler gives \&SYSSTYP the character string value of the type of the control section in use at the point at which a macro is called. For a macro definition called by an inner macro call, the assembler assigns to \&SYSSTYP the type of the control section in effect in the macro definition that contains the inner macro call, at the time the inner macro is called.

If no control section is generated within a macro definition, the value assigned to \&SYSSTYP does not change. It is the same for the next level of macro definition called by an inner macro instruction.

\section*{Chapter 9. How to Write Conditional Assembly Instructions}

This chapter describes the conditional assembly language. With the conditional assembly language, you can carry out general arithmetic and logical computations, and many of the other functions you can carry out with any other programming language. Also, by writing conditional assembly instructions in combination with other assembler language statements, you can:
- Select sequences of these source statements, called model statements, from which machine and assembler instructions are generated
- Vary the contents of these model statements during generation

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

The conditional assembly language is more versatile when you use it to interact with symbolic parameters and the system variable symbols inside a macro definition. However, you can also use the conditional assembly language in open code; that is, code that is not within a macro definition.

\section*{Elements and Functions}

The elements of the conditional assembly language are:
- SET symbols that represent data. See "SET Symbols" on page 319
- Attributes that represent different characteristics of symbols. See "Data Attributes" on page 324.
- Sequence symbols that act as labels for branching to statements during conditional assembly processing. See "Sequence Symbols" on page 339

The functions of the conditional assembly language are:
- Declaring SET symbols as variables for use locally and globally in macro definitions and open code. See "Declaring SET Symbols" on page 343 .
- Assigning values to the declared SET symbols. See "Assigning Values to SET Symbols" on page 347 .
- Selecting characters from strings for substitution in, and concatenation to, other strings; or for inspection in condition tests. See "Substring Notation" on page 371.
- Branching and exiting from conditional assembly loops. See "Branching" on page 390 .

The conditional assembly language can also be used in open code with few restrictions. See "Open Code" on page 342

The conditional assembly language provides instructions for evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, and as condition tests for branching. See "Conditional Assembly Instructions" on page 343 for details about the syntax and usage rules of each instruction.

\section*{SET Symbols}

SET symbols are variable symbols that provide you with arithmetic, binary, or character data, and whose values you can vary during conditional assembly processing.

Use SET symbols as:
- Terms in conditional assembly expressions
- Counters, switches, and character strings
- Subscripts for variable symbols
- Values for substitution

Thus, SET symbols let you control your conditional assembly logic, and to generate many different statements from the same model statement.

\section*{Subscripted SET Symbols}

You can use a SET symbol to represent a one-dimensional array of many values. You can then refer to any one of the values of this array by subscripting the SET symbol. For more information, see "Subscripted SET Symbol Specification" on page 322

\section*{Scope of SET Symbols}

The scope of a SET symbol is that part of a program for which the SET symbol has been declared. Local SET symbols need not be declared by explicit declarations. The assembler considers any undeclared variable symbol found in the name field of a SETx instruction as a local SET symbol.

If you declare a SET symbol to have a local scope, you can use it only in the statements that are part of either:
- The same macro definition, or
- Open code

If you declare a SET symbol to have a global scope, you can use it in the statements that are part of any one of:
- The same macro definition
- A different macro definition
- Open code

You must, however, declare the SET symbol as global for each part of the program (a macro definition or open code) in which you use it.

You can change the value assigned to a SET symbol without affecting the scope of this symbol.

\section*{Scope of Symbolic Parameters}

A symbolic parameter has a local scope. You can use it only in the statements that are part of the macro definition for which the parameter is declared. You declare a symbolic parameter in the prototype statement of a macro definition.

The scope of system variable symbols is described in Figure 86 on page 320 .

\section*{SET Symbol Specifications}

SET symbols can be used in model statements, from which assembler language statements are generated, and in conditional assembly instructions. The three types of SET symbols are: SETA, SETB, and SETC. A SET symbol must be a valid variable symbol.

The rules for creating a SET symbol are:
- The first character must be an ampersand (\&)
- The second character must be an alphabetic character
- The remaining characters must be 0 to 61 alphanumeric
- The first four characters should not be \&SYS, which are used for system variable symbols

\section*{Examples:}
\&ARITHMETICVALUE439
\&BOOLEAN
\&C
\&EASY_TO_READ
Local SET symbols need not be declared by explicit declarations. The assembler considers any undeclared variable symbol found in the name field of a SETx instruction as a local SET symbol, and implicitly declares it to have the type specified by the SETx instruction. The instruction that declares a SET symbol determines its scope and type.

The features of SET symbols and other types of variable symbols are compared in Figure 86.

Figure 86 (Page 1 of 3). Features of SET Symbols and Other Types of Variable Symbols
\begin{tabular}{|c|c|c|c|}
\hline Features & \begin{tabular}{l}
SETA, \\
SETB, \\
SETC \\
symbols
\end{tabular} & Symbolic Parameters & System Variable Symbols \\
\hline \multicolumn{4}{|l|}{Can be used in:} \\
\hline Open code & Yes & No & \&SYSASM \\
\hline & & & \&SYSDATC \\
\hline & & & \&SYSDATE \\
\hline & & & \&SYSJOB \\
\hline & & & \&SYSM_HSEV \\
\hline & & & \&SYSM_SEV \\
\hline & & & \&SYSOPT_DBCS \\
\hline & & & \&SYSOPT_OPTABLE \\
\hline & & & \&SYSOPT_RENT \\
\hline & & & \&SYSOPT_XOBJECT \\
\hline & & & \&SYSPARM \\
\hline & & & \&SYSSTEP \\
\hline & & & \&SYSSTMT \\
\hline & & & \&SYSTEM_ID \\
\hline & & & \&SYSTIME \\
\hline & & & \&SYSVER \\
\hline Macro definitions & Yes & Yes & All \\
\hline
\end{tabular}

Figure 86 (Page 2 of 3). Features of SET Symbols and Other Types of Variable Symbols
\begin{tabular}{|c|c|c|c|}
\hline Features & \begin{tabular}{l}
SETA, \\
SETB, \\
SETC \\
symbols
\end{tabular} & Symbolic Parameters & System Variable Symbols \\
\hline \multicolumn{4}{|l|}{Scope:} \\
\hline \multirow[t]{29}{*}{Local} & \multirow[t]{29}{*}{Yes} & \multirow[t]{29}{*}{Yes} & \&SYSADATA_DSN \\
\hline & & & \&SYSADATA_MEMBER \\
\hline & & & \&SYSADATA_VOLUME \\
\hline & & & \&SYSCLOCK \\
\hline & & & \&SYSECT \\
\hline & & & \&SYSIN_DSN \\
\hline & & & \&SYSIN_MEMBER \\
\hline & & & \&SYSIN_VOLUME \\
\hline & & & \&SYSLIB_DSN \\
\hline & & & \&SYSLIB_MEMBER \\
\hline & & & \&SYSLIB_VOLUME \\
\hline & & & \&SYSLIN_DSN \\
\hline & & & \&SYSLIN_MEMBER \\
\hline & & & \&SYSLIN_VOLUME \\
\hline & & & \&SYSLIST \\
\hline & & & \&SYSLOC \\
\hline & & & \&SYSMAC \\
\hline & & & \&SYSNDX \\
\hline & & & \&SYSNEST \\
\hline & & & \&SYSPRINT_DSN \\
\hline & & & \&SYSPRINT_MEMBER \\
\hline & & & \&SYSPRINT_VOLUME \\
\hline & & & \&SYSPUNCH_DSN \\
\hline & & & \&SYSPUNCH_MEMBER \\
\hline & & & \&SYSPUNCH_VOLUME \\
\hline & & & \&SYSSEQF \\
\hline & & & \&SYSTERM_DSN \\
\hline & & & \&SYSTERM_MEMBER \\
\hline & & & \&SYSTERM_VOLUME \\
\hline \multirow[t]{16}{*}{Global} & \multirow[t]{16}{*}{Yes} & \multirow[t]{16}{*}{No} & \&SYSASM \\
\hline & & & \&SYSDATC \\
\hline & & & \&SYSDATE \\
\hline & & & \&SYSJOB \\
\hline & & & \&SYSM_HSEV \\
\hline & & & \&SYSM_SEV \\
\hline & & & \&SYSOPT_DBCS \\
\hline & & & \&SYSOPT_OPTABLE \\
\hline & & & \&SYSOPT_RENT \\
\hline & & & \&SYSOPT_XOBJECT \\
\hline & & & \&SYSPARM \\
\hline & & & \&SYSSTEP \\
\hline & & & \&SYSSTMT \\
\hline & & & \&SYSTEM_ID \\
\hline & & & \&SYSTIME \\
\hline & & & \&SYSVER \\
\hline
\end{tabular}

Figure 86 (Page 3 of 3). Features of SET Symbols and Other Types of Variable Symbols
\begin{tabular}{llll}
\hline & \begin{tabular}{l} 
SETA, \\
Features
\end{tabular} & \begin{tabular}{l} 
Symbolic \\
SETB,
\end{tabular} & \begin{tabular}{l} 
System Variable \\
SETC
\end{tabular} \\
symbols
\end{tabular}

Notes:
1. The value assigned to a SET symbol can be changed by using the SETA, SETAF, SETB, SETC, or SETCF instruction within the declared or implied scope of the SET symbol.
2. A symbolic parameter and the system variable symbols (except for \&SYSSTMT, \&SYSM_HSEV, and \&SYSM_SEV) are assigned values that remain fixed throughout their scope. Wherever a SET symbol appears in a statement, the assembler replaces the symbol's current value with the value assigned to it.

SET symbols can be used in the name, operation, and operand fields of macro instructions. The value thus passed through the name field symbolic parameter into a macro definition is considered as a character string and is generated as such. If the COMPAT(SYSLIST) assembler option is specified, the value passed through an operand field symbolic into a macro definition is also considered a character string and is generated as such. However, if the COMPAT(SYSLIST) assembler option is not specified, SET symbols can be used to pass sublists into a macro definition.

\section*{Subscripted SET Symbol Specification}

The format of a subscripted SET symbol is shown below.
\(\square\)

\section*{\&symbol}
is a variable symbol.
subscript
is an arithmetic expression with a value greater than or equal to 1.

\section*{Example:}
\&ARRAY (20)
The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (see "Arithmetic (SETA) Expressions" on page 352.

The subscript refers to one of the many positions in an array of values identified by the SET symbol.

A subscripted SET symbol can be used anywhere an unsubscripted SET symbol is allowed. However, subscripted SET symbols must be declared as subscripted by a previous local or global declaration instruction, or implicitly as a local subscripted SET symbol in a SETx instruction of the desired type.

The dimension (the maximum value of the subscript) of a subscripted SET symbol is not determined by the explicit or implicit declaration of the symbol. The dimension specified can be exceeded in later SETx instructions. Note, however, that increasing the dimension of a subscripted SET symbol also increases the storage required. For example, referencing only \&ARRAY(1000000) still causes the preceding 999999 elements to be allocated. You can determine the maximum subscript using the N ' attribute (see "Number Attribute ( \(\mathrm{N}^{\prime}\) )" on page 336).

The subscript can be a subscripted SET symbol.

\section*{Created SET Symbols}

The assembler can create SET symbols during conditional assembly processing from other variable symbols and character strings. A SET symbol thus created has the form \& \((e)\), where \(e\) represents one or more of the following:
- Variable symbols, optionally subscripted
- Strings of alphanumeric characters
- Other created SET symbols

After substitution and concatenation, \(e\) must consist of a string of up to 62 alphanumeric characters, the first of which is alphabetic. The assembler considers the preceding ampersand and this string as the name of a SET variable. If this created SET symbol has the same name as an existing SET symbol, they are treated as identical. If this created SET symbol does not have the name of any existing SET symbol, the usual rules for assigning type and scope apply.

You can use created SET symbols wherever ordinary SET symbols are permitted, including declarations. A created SET symbol may not match the name of a system variable symbol, nor the name of a symbolic parameter in a macro prototype statement. You can also nest created SET symbols in other created SET symbols.

Consider the following example:
```

\&ABC(1) SETC 'MKT','27','\$5'
Let \&(e) equal \&(\&ABC(\&I)QUA\&I).
\&I \&ABC(\&I) Created SET Symbol Comment

| 1 | MKT | \&MKTQUA1 | Valid |
| :--- | :--- | :--- | :--- |
| 2 | 27 | \&27QUA2 | Invalid: character after '\&' not alphabetic |
| 3 | $\$ 5$ | \&\$5QUA3 | Valid |
| 4 |  | \&QUA4 | Valid |

```

The name of a created SET symbol cannot match the name of a system variable symbol or of a symbolic parameter in a macro definition.

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

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

\section*{Data Attributes}
this example indicates, created SET symbols may be declared and used as arrays of dimensioned variables.

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

\section*{Data Attributes}

The data, such as instructions, constants, and areas, that you define in a source module, can be described by its:
- Type, which distinguishes a property of a named object or macro argument, for example, fixed-point constants from floating-point constants, or machine instructions from macro instructions
- Length, which gives the number of bytes occupied by the object code of the named data
- Scaling, which shows the number of positions occupied by the fractional portion of named fixed-point, floating-point, and decimal constants in their object code form
- Integer, which shows the number of positions occupied by the integer portion of named fixed-point and decimal constants in their object code form
- Count, which gives the number of characters that would be required to represent the named data, such as a macro instruction operand, as a character string
- Number, which gives the number of sublist entries in a macro instruction operand
- Defined, which determines whether a symbol has been defined prior to the point where the attribute reference is coded
- Operation Code, which shows if an operation code, such as a macro definition or machine instruction, is defined prior to the point where the attribute reference is coded

These characteristics are called the attributes of the symbols naming the data. The assembler assigns attribute values to the ordinary symbols and variable symbols that represent the data.

Specifying attributes in conditional assembly instructions allows you to control conditional assembly logic, which, in turn, can control the sequence and contents of the statements generated from model statements. The specific purpose for which you use an attribute depends on the kind of attribute being considered. The attributes and their main uses are shown below:

Figure 87. Data Attributes
\begin{tabular}{|c|c|c|}
\hline Attribute & Purpose & Main Uses \\
\hline Type & Gives a letter that identifies type of data represented & \begin{tabular}{l}
- In tests to distinguish between different data types \\
- For value substitution \\
- In macros to discover missing operands
\end{tabular} \\
\hline Length & Gives number of bytes that data occupies in storage & \begin{tabular}{l}
- For substitution into length fields \\
- For computation of storage requirements
\end{tabular} \\
\hline Scaling & Refers to the position of the decimal point in fixed-point, floating-point and decimal constants & \begin{tabular}{l}
- For testing and regulating the position of decimal points \\
- For substitution into a scale modifier
\end{tabular} \\
\hline Integer & Is a function of the length and scale attributes of decimal, fixed-point, and floating-point constants & - To keep track of significant digits (integers) \\
\hline Count & Gives the number of characters required to represent data & \begin{tabular}{l}
- For scanning and decomposing character strings \\
- As indexes in substring notation
\end{tabular} \\
\hline Number \({ }^{1}\) & Gives the number of sublist entries in a macro instruction operand sublist, or the maximum subscript of a dimensioned SET symbol to which a value has been assigned. & \begin{tabular}{l}
- For scanning sublists \\
- As a counter to test for end of sublist \\
- For testing array limits
\end{tabular} \\
\hline Defined & Shows whether the symbol referenced has been defined prior to the attribute reference & - To avoid defining a symbol again if the symbol referenced has been previously defined \\
\hline Operation Code & Shows whether a given operation code has been defined prior to the attribute reference & - To avoid assembling a macro or instruction if it does not exist. \\
\hline
\end{tabular}

\section*{Notes:}
1. The number attribute of \(\& \operatorname{SYSLIST}(n)\) and \(\& \operatorname{SYSLIST}(n, m)\) is described in " \(\& \operatorname{SYSLIST}\) System Variable Symbol" on page 276

\section*{Attribute Reference}


\section*{attribute_notation'}
is the attribute whose value you want, followed by a single quotation mark. Valid attribute letters are "D," "O," "N," "S," "K," "I," "L," and "T."
ordinary_symbol
is an ordinary symbol that represents the data that possesses the attribute. An ordinary symbol cannot be specified with the operation code attribute.

\section*{Data Attributes}

\section*{variable_symbol}
is a variable symbol that represents the data that possesses the attribute.
literal
is a literal that represents the data that possesses the attribute. A literal cannot be specified with the operation code attribute or count attribute.
character_string
is a character string that represents the operation code in the operation code attribute.

\section*{Examples:}

T'SYMBOL
L'\&VAR
K'\&PARAM
0 'MVC
\(S^{\prime}=\) P' \(^{\prime} 975.32^{\prime}\)
The assembler substitutes the value of the attribute for the attribute reference.
Reference to the count ( \(\mathrm{K}^{\prime}\) ), defined ( \(\mathrm{D}^{\prime}\) ), number ( \(\mathrm{N}^{\prime}\) ), operation code ( \(\mathrm{O}^{\prime}\) ), and type ( \(\mathrm{T}^{\prime}\) ) attributes can be used only in conditional assembly instructions or within macro definitions. The length ( \(L^{\prime}\) ), integer ( \(I^{\prime}\) ), and scale ( \(\mathrm{S}^{\prime}\) ) attribute references can be in conditional assembly instructions, machine instructions, assembler instructions, and the operands of macro instructions.

\section*{Attributes of Symbols and Expressions}

Figure 88 shows attribute references (in the columns) and types of symbols (in the rows). Each intersection shows whether ("Yes") or not ("No") you can validly apply the attribute reference to that symbol type, or (in the case of SET symbols) to the value of the symbol.

Figure 88 (Page 1 of 2). Attributes and Related Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline Symbols Specified & Type T' & Length L' & Scale S' & Integer I' & Count K' & Number
\[
\mathbf{N}^{\prime}
\] & Defined
\[
D^{\prime}
\] & Operation Code O' \\
\hline \multicolumn{9}{|l|}{In open code:} \\
\hline Ordinary symbols & Yes & Yes & Yes & Yes & No & No & Yes & No \\
\hline System variable symbols with global scope & Yes & No & No & No & Yes & Yes & No & No \\
\hline Literals & Yes & Yes & Yes & Yes & No & No & Yes & No \\
\hline
\end{tabular}

Figure 88 (Page 2 of 2). Attributes and Related Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline Symbols Specified & Type T' & Length L' & Scale S' & Integer I' & Count \(K^{\prime}\) & Number
\[
N^{\prime}
\] & Defined D' & \begin{tabular}{l}
Operation \\
Code \({ }^{\circ}\)
\end{tabular} \\
\hline \multicolumn{9}{|l|}{In macro definitions:} \\
\hline Ordinary symbols & Yes & Yes & Yes & Yes & No & No & Yes & No \\
\hline Symbolic parameters & Yes & Yes & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline \multicolumn{9}{|l|}{System variable symbols:} \\
\hline \&SYSLIST & Yes & Yes & Yes & Yes & Yes & Yes & Yes & No \\
\hline All others & Yes & No & No & No & Yes & Yes & No & No \\
\hline Literals in macro instruction operands & Yes & Yes & Yes & Yes & No & No & Yes & No \\
\hline
\end{tabular}

The values of attribute references may be used in ordinary and conditional assembly expressions, as shown in Figure 89.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{9}{|l|}{Figure 89. Using Attribute Values} \\
\hline Symbols Specified & \[
\begin{aligned}
& \text { Type } \\
& \text { T' }^{\prime}
\end{aligned}
\] & Length L' & Scale
S' & Integer I' & Count
\[
K^{\prime}
\] & Number \(N^{\prime}\) & Defined
D' & \begin{tabular}{l}
Operation \\
Code O'
\end{tabular} \\
\hline In open code: SET symbols & \[
\begin{aligned}
& \text { SETB }^{1} \\
& \text { SETC }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB² }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB² }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB² }
\end{aligned}
\] & SETA,
\[
\text { SETB }{ }^{2}
\] & SETA,
SETB2 & SETA,
SETB2 & \[
\begin{aligned}
& \text { SETB }^{1}, \\
& \text { SETC }
\end{aligned}
\] \\
\hline In ordinary assembly: & No & Yes & Yes & Yes & No & No & No & No \\
\hline In macro definitions: SET symbols & \[
\begin{aligned}
& \text { SETB }^{1} \\
& \text { SETC }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB² }
\end{aligned}
\] & \[
\begin{aligned}
& \hline \text { SETA, } \\
& \text { SETB2}
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB² }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB } 2
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB22 }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETA, } \\
& \text { SETB22 }
\end{aligned}
\] & \[
\begin{aligned}
& \text { SETB }^{1}, \\
& \text { SETC }
\end{aligned}
\] \\
\hline
\end{tabular}

\section*{Note:}
1. Only in character relations.
2. Only in arithmetic relations.

The value of an attribute for an ordinary symbol specified in an attribute reference comes from the item named by the symbol. The symbol must appear in the name field of an assembler or machine instruction, or in the operand field of an EXTRN or WXTRN instruction.

The value of an attribute reference to an expression is the value of that attribute reference to its leftmost term.

\section*{Notes:}
1. You cannot refer to the names of instructions generated by conditional assembly substitution or macro generation until the instruction is generated.
2. If you use a symbol qualifier to qualify an ordinary symbol in an attribute reference, the qualifier is ignored.

\section*{Data Attributes}

The value of an attribute for a variable symbol specified in an attribute reference comes from the value substituted for the variable symbol as follows:

SET Symbols and System Variable Symbols: For SET symbols and all system variable symbols other than \&SYSLIST, the attribute values come from the current value of these symbols.

Symbolic Parameters and \&SYSLIST: For symbolic parameters and the system variable symbol, \&SYSLIST, the values of the count and number attributes come from the operands of macro instructions. The name field entry of the call is an "operand," and is referenceable as \&SYSLIST(0). The values of the type, length, scale, and integer attributes, however, come from the values represented by the macro instruction operands, as follows:
1. If the operand is a sublist, the entire sublist and each entry of the sublist can possess attributes; all the individual entries and the whole sublist have the same attributes as those of the first suboperand in the sublist (except for the count attribute, which can be different, and the number attribute which is relevant only for the whole sublist).
2. If the first character or characters of the operand (or sublist entry) constitute an ordinary symbol, and this symbol is followed by either an arithmetic operator (+, -, *, or /), a left parenthesis, a comma, or a space, then the value of the attributes for the operand are the same as for the ordinary symbol.
3. If the operand (or sublist entry) is a character string other than a sublist or the character string described in the previous point, the type attribute is undefined \((\mathrm{U})\) and the length, scale, and integer attributes are invalid.

Because the count ( \(\mathrm{K}^{\prime}\) ), number ( \(\mathrm{N}^{\prime}\) ), and defined \(\left(\mathrm{D}^{\prime}\right)\) attribute references are allowed only in conditional assembly instructions, their values are available only during conditional assembly processing. They are not available at ordinary assembly time.

The system variable symbol \&SYSLIST, with a valid subscript, can be used in an attribute reference to refer to a macro instruction operand, and, in turn, to an ordinary symbol. Thus, any of the attribute values for macro instruction operands and ordinary symbols listed below can also be substituted for an attribute reference containing \&SYSLIST (see "\&SYSLIST System Variable Symbol" on page 276).

\section*{Type Attribute ( \(\mathrm{T}^{\prime}\) )}

The type attribute has a value of a single alphabetic character that shows the type of data represented by:
- An ordinary symbol
- A macro instruction operand
- A SET symbol
- A literal
- A system variable symbol

The type attribute can change during an assembly. The lookahead search might assign one attribute, whereas the symbol table at the end of the assembly might display another.

The type attribute reference can be used in the operand field of a SETC instruction or as one of the values used for comparison in the operand field of a SETB or AIF instruction.

The type attribute can also be specified outside conditional assembly instructions. Then, the type attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

The following letters are used for the type attribute of data represented by ordinary symbols and outer macro instruction operands that are symbols that name DC or DS statements.

A A-, J-type address constant, implied length, aligned (also CXD instruction label)
B Binary constant
C Character constant
D Long floating-point constant, implicit length, aligned
E Short floating-point constant, implicit length, aligned
F Fullword fixed-point constant, implicit length, aligned
G Fixed-point constant, explicit length
H Halfword fixed-point constant, implicit length, aligned
K Floating-point constant, explicit length
L Extended floating-point constant, implicit length, aligned
P Packed decimal constant
Q Q-type address constant, implicit length, aligned
R A-, S-, Q-, J-, R-, V-, or Y-type address constant, explicit length
S S-type address constant, implicit length, aligned
V R-, V-type address constant, implicit length, aligned
X Hexadecimal constant
Y Y-type address constant, implicit length, aligned
Z Zoned decimal constant
@ Graphic (G) constant
When a literal is specified as the name field on a macro call instruction, and if the literal has previously been used in a machine instruction, the type attribute of the literal is the same as for data represented by ordinary symbols or outer macro instructions operands.

The following letters are used for the type attribute of data represented by ordinary symbols (and outer macro instruction operands that are symbols) that name statements other than DC or DS statements, or that appear in the operand field of an EXTRN or WXTRN statement:

I Machine instruction
J Control section name
M The name field on a macro instruction, when the name field is:
- a valid symbol not previously defined
- a valid literal not previously defined

T Identified as an external symbol by EXTRN instruction
w CCW, CCWO, or CCW1 instruction
\$ Identified as an external symbol by WXTRN instruction
The following letter is used for the type attribute of data represented by inner and outer macro instruction operands only:

O Omitted operand (has a value of a null character string)

\section*{Data Attributes}

The following attribute is used for the type attribute of the value of variable symbols:

\section*{N The value is numeric}

The following letter is used for symbols or macro instruction operands that cannot be assigned any of the above letters:
U Undefined, unknown, or unassigned
The common use of the \(U\) type attribute is to describe a valid symbol that has not been assigned any of the type attribute values described above. If the assembler is not able to determine what the named symbol represents, it also assigns the \(U\) type attribute. Thus, the \(U\) type attribute can mean undefined, or unknown, or unassigned at the time of the reference. Consider the following macro definition:
\begin{tabular}{lll} 
Name & \begin{tabular}{l} 
Operation \\
macro
\end{tabular} & Operand \\
& MAC1 \&op1, \&op2
\end{tabular}\(\quad\).

When the macro MAC1 is called in Figure 90, neither of the operands has previously been defined, however GOOD_SYMBOL is a valid symbol name, whereas ?BAD_SYMBOL? is not a valid symbol name. The type attribute for both operands is U , meaning GOOD_SYMBOL is undefined, and ?BAD_SYMBOL? is unknown.
\begin{tabular}{lccc}
\hline 000000 & 0000000004 & 8 a csect & \\
& & 9 & mac1 GOOD SYMBOL, ?BAD_SYMBOL? \\
000000 E4 & \(10+\) & DC C' ' ' & DC containing type attribute for op1 \\
000001 E4 & \(11+\) & DC C'U' & DC containing type attribute for op2
\end{tabular}

Figure 90. Undefined and Unknown Type Attributes
When the macro MAC1 is called in Figure 91, GOOD_SYMBOL is a valid symbol name, and has been defined in the DC instruction at statement 12. ?BAD_SYMBOL? is a not valid symbol name, and the assembler issues an error message at statement 13. The type attribute for GOOD_SYMBOL is C, meaning the symbol represents a character constant. The type attribute for ?BAD_SYMBOL? is U , meaning the type is unknown.


Figure 91. Unknown Type Attribute for Invalid Symbol
The type attribute value U , meaning undefined, unknown, or unassigned, is assigned to the following:
- Ordinary symbols used as labels:

\section*{Data Attributes}
- For the LTORG instruction
- For the EQU instruction without a third operand
- For DC and DS statements that contain variable symbols, for example, U1 DC \& ' \(1{ }^{\prime}\)
- That are defined more than once, even though only one instance of the label is generated due to conditional assembly statements. A lookahead scan for attributes of a symbol may encounter more than one occurrence of a symbol, in which case the assembler can't yet tell which statement(s) will be generated. In such cases, type attribute \(U\) is assigned. At a later time, when the symbol has been generated, its type attribute is changed to the correct value for the type of statement it names.
- SETC variable symbols that have a value other than a null character string or the name of an instruction that can be referred to be a type attribute reference
- System variable symbols except:
- \&SYSDATC, \&SYSM_HSEV, \&SYSM_SEV, \&SYSNDX, \&SYSNEST, \&SYSOPT_DBCS, \&SYSOPT_RENT, \&SYSOPT_XOBJECT, and \&SYSSTMT, which always have a type attribute value of \(N\)
- Some other character type system variable symbols which may be assigned a value of a null string, when they have a type attribute value of O
- Macro instruction operands that specify a literal that is not a duplicate of a literal used in a machine instruction
- Inner macro instruction operands that are ordinary symbols

\section*{Notes:}
1. Ordinary symbols used in the name field of an EQU instruction have the type attribute value U. However, the third operand of an EQU instruction can be used explicitly to assign a type attribute value to the symbol in the name field.
2. The type attribute of a sublist is set to the same value as the type attribute of the first element of the sublist.
3. High Level Assembler and earlier assemblers treat the type attribute differently:
- Because High Level Assembler allows attribute references to statements generated through substitution, certain cases in which a type attribute of \(U\) (undefined, unknown, or unassigned) or M (macro name field) is given under the DOS/VSE Assembler, may give a valid type attribute under High Level Assembler. If the value of the SETC symbol is equal to the name of an instruction that can be referred to by the type attribute, High Level Assembler lets you use the type attribute with a SETC symbol.
- Because High Level Assembler allows attribute references to literals, certain cases in which a type attribute of \(U\) (undefined, unknown, or unassigned) is given by Assembler F and Assembler H for a macro operand that specifies a literal, may give a valid type attribute under High Level Assembler. If the literal specified in the macro instruction operand is a duplicate of a literal specified in open code, or previously generated by conditional assembly processing or macro generation, High Level

Assembler gives a type attribute that shows the type of data specified in the literal. The COMPAT(LITTYPE) option causes High Level Assembler to behave like Assembler \(H\), always giving a type attribute of \(U\) for the \(\mathrm{T}^{\prime}\) literal.

\section*{Length Attribute (L')}

The length attribute has a numeric value equal to the number of bytes occupied by the data that is named by the symbol specified in the attribute reference.

If the length attribute value is desired for conditional assembly processing, the symbol specified in the attribute reference must ultimately represent the name entry of a statement in open code. In such a statement, the length modifier (for DC and DS instructions) or the length field (for a machine instruction), if specified, must be a self-defining term, of a predefined absolute symbol. If multiple operands are specified in a DC or DS instruction, the attribute of the statement label is determined from the first operand. The length modifier or length field must not be coded as a multiterm expression, because the assembler does not evaluate this expression until assembly time.

The assembler lets you use the length attribute with a SETC symbol, a system variable symbol, or a symbolic parameter, if the value of the SETC symbol is an ordinary symbol that can be referenced by the length attribute.

The length attribute can also be specified outside conditional assembly instructions. Then, the length attribute value is not available for conditional assembly processing, but is used as a value at assembly time.

Figure 92 is an example showing the evaluation of the length attribute for an assembler instruction in statement 1 and for a conditional assembly instruction in statement 8.
\begin{tabular}{|c|c|c|}
\hline 000000 E740 & \[
\begin{aligned}
& 1 \text { CSYM DC } \\
& 2 \text { \&LEN SETA }
\end{aligned}
\] & CL(L'ZLOOKAHEAD)'X' Length resolved later L'CSYM \\
\hline \multicolumn{3}{|l|}{** ASMA042E Length attribute of symbol is unavailable; default=1} \\
\hline & 3 DC & C'\&LEN ' REAL LENGTH NOT AVAILABLE \\
\hline \multirow[t]{3}{*}{000002 F140} & \(+\quad \mathrm{DC}\) & C'1 ' REAL LENGTH NOT AVAILABLE \\
\hline & 4 \&TYP SETC & T'CSYM \\
\hline & 5 DC & C'\&TYP ' TYPE IS KNOWN \\
\hline \multirow[t]{3}{*}{000004 C340} & \(+\quad \mathrm{DC}\) & \(C^{\prime} C^{\prime}\) TYPE IS KNOWN \\
\hline & 6 \&DEF SETA & D'CSYM \\
\hline & 7 DC & C'\&DEF ' SYMBOL IS DEFINED \\
\hline \multirow[t]{3}{*}{000006 F140} & \(+\quad D C\) & C'1 ' SYMBOL IS DEFINED \\
\hline & 8 \&LEN SETA & L'zlookahead Length resolved immediately \\
\hline & 9 CSYM2 DC & CL(\&len)'X' \\
\hline \multirow[t]{3}{*}{000008 E740} & +CSYM2 DC & CL(2) ' \(\mathrm{X}^{\prime}\) \\
\hline & 10 \&LEN SETA & L'CSYM2 \\
\hline & 11 DC & C'\&LEN ' REAL LENGTH NOW AVAILABLE \\
\hline 00000A F240 & \(+\quad \mathrm{DC}\) & C'2' REAL LENGTH NOW AVAILABLE \\
\hline \multirow[t]{2}{*}{00000C 0001} & 12 ZLOOKAHEAD & DC H'1' \\
\hline & 13 END & \\
\hline
\end{tabular}

Figure 92. Evaluation of Length Attribute References
In statement 2 the length of CSYM has not been established because the definition of CSYM in statement 1 is not complete. The reference to the length attribute results in a length of 1 and error message ASMA042E. However, statement 5 shows that the
type attribute is assigned, and statement 7 shows that the defined attribute is assigned. In comparison, the length attribute for symbol CSYM2 is available immediately, as it was retrieved indirectly using the conditional assembly instruction in statement 8.

During conditional assembly, an ordinary symbol used in the name field of an EQU instruction has a length attribute value that depends on the order of the symbol's definition and the reference to its length attribute.
- If the first operand of the EQU instruction is a self-defining term, the length attribute value is 1 .
- If the first operand of the EQU instruction is a symbol whose value and length attribute are defined, the length attribute value is that of the symbol in the first operand.
- If the first operand of the EQU instruction is a defined symbol and the EQU instruction specifies a length value in the second operand, the length attribute value is that of the second operand.
At assembly time, the symbol has the same length attribute value as the first term of the expression in the first operand of the EQU instruction. However, the second operand of an EQU instruction can be used to assign a length attribute value to the symbol in the name field. This second operand can not be a forward reference to another EQU instruction.

\section*{Notes:}
1. The length attribute reference, when used in conditional assembly processing, can be specified only in arithmetic expressions.
2. When used in conditional assembly processing, a length attribute reference to a symbol with the type attribute value of \(\mathrm{M}, \mathrm{N}, \mathrm{O}, \mathrm{T}, \mathrm{U}\), or \(\$\) is flagged. The length attribute for the symbol has the default value of 1 .

\section*{Scale Attribute (S')}

The scale attribute can be used only when referring to fixed-point, floating-point, or decimal constants. The following table shows the numeric value assigned to the scale attribute:
\begin{tabular}{|c|c|c|}
\hline Constant Types Allowed & Type of DC or DS Allowed & Value of Scale Attribute Assigned \\
\hline Fixed-Point & H and F & Equal to the value of the scale modifier (-187 through +346) \\
\hline Floating Point & D, E, and L & \begin{tabular}{l}
Equal to the value of the scale modifier (0 through 14 - D, E) \\
(0 through \(28-\mathrm{L}\) )
\end{tabular} \\
\hline Decimal & \(P\) and \(Z\) & \begin{tabular}{l}
Equal to the number of decimal digits specified to the right of the decimal point \\
(0 through \(31-\mathrm{P}\) ) \\
(0 through \(16-\mathrm{Z}\) )
\end{tabular} \\
\hline
\end{tabular}

\section*{Data Attributes}

The scale attribute can also be specified outside conditional assembly instructions. Then, the scale attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

\section*{Notes:}
1. The scale attribute reference can be used only in arithmetic expressions.
2. When no scale attribute value can be determined, the reference is flagged and the scale attribute is 1 .
3. If the value of the SETC symbol is equal to the name of an instruction that can validly define the scale attribute, the assembler lets you use the scale attribute with a SETC symbol.
4. Binary floating-point constants return an attribute of 0 .
5. The scale attribute reference can only be used in arithmetic expressions in conditional assembly instructions, and in absolute and relocatable expressions in assembler and machine instructions.

\section*{Integer Attribute (I')}

The integer attribute has a numeric value that depends on the length and scale attribute values of the data being referred to by the attribute reference. The formulas relating the integer attribute to the length and scale attributes are given in Figure 93.

The integer attribute can also be specified outside conditional assembly instructions. Then, the integer attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

\section*{Notes:}
1. The integer attribute reference can be used only in arithmetic expressions.
2. When no integer attribute value can be determined, the reference is flagged and the integer attribute is 1.
3. If the value of the SETC symbol is equal to the name of an instruction that can validly define the integer attribute, the assembler lets you use the integer attribute with a SETC symbol.
4. Binary floating-point constants return an attribute of 0 .
5. The integer attribute reference can only be used in arithmetic expressions in conditional assembly instructions, and in absolute and relocatable expressions in assembler and machine instructions.

Figure 93 (Page 1 of 2). Relationship of Integer to Length and Scale Attributes
\begin{tabular}{l|l|l|l}
\hline & \begin{tabular}{l} 
Formula Relating Integer \\
to Length and Scale \\
Attributes
\end{tabular} & Examples & \begin{tabular}{l} 
Values of the \\
Integer Attribute
\end{tabular} \\
\hline Constant Type & \(I^{\prime}=8 * L^{\prime}-S^{\prime}-1\) & HALFCON DC HS6' \(-25.93^{\prime}\) & \(I^{\prime}=8 * 2-6-1\) \\
\(=\)\begin{tabular}{l} 
Fixed-point \\
\((H\) and F)
\end{tabular} & ONECON DC FS8'100.3E-2' & \(I^{\prime}=8 * 4-8-1\) \\
\(=23\)
\end{tabular}

Figure 93 (Page 2 of 2). Relationship of Integer to Length and Scale Attributes


\section*{Notes:}
1. The value of the integer attribute is equal to the number of digits to the left of the assumed decimal point after the constant is assembled, and the value of the scale attribute is equal to the number of digits to the right of the assumed decimal point.

\section*{Count Attribute (K')}

The count attribute applies only to macro instruction operands, to SET symbols, and to the system variable symbols. It has a numeric value equal to the number of characters:
- That constitute the macro instruction operand, or
- That would be required to represent as a character string the current value of the SET symbol or the system variable symbol.

\section*{Notes:}
1. The count attribute reference can be used only in arithmetic expressions.
2. The count attribute of an omitted macro instruction operand has a value of 0 .
3. Doubled quotes ( \({ }^{\prime}\) ') in quoted character strings count as one character. Doubled ampersands (\&\&) in quoted character strings count as two characters. For more information about character pairs see "Evaluation of Character Expressions" on page 382
4. These pairing rules mean that the length attribute of a character variable substituted into a character constant may be different from the count attribute of the substituted variable.
5. The count attribute differs from the Number ( \(N^{\prime}\) ) attribute, described below.

\section*{Number Attribute ( \({ }^{\prime}\) ')}

The number attribute applies to the operands of macro instructions and subscripted SET symbols.

When applied to a macro operand, the number attribute is a numeric value equal to the number of sublist entries.

When applied to a subscripted SET symbol, the number attribute is equal to the highest element to which a value has been assigned in a SETx instruction. Consider the example in Figure 94.
\begin{tabular}{|c|c|c|}
\hline & 1 & macro \\
\hline & 2 & MAC1 \& \({ }^{\text {a }} 1\) \\
\hline & 3 & 1cla \&SETSUB(100) \\
\hline & 4 \&SETSUB(5) & seta 20, , 70 \\
\hline & 5 \& B & seta N'\&SETSUB \\
\hline & 6 \& C & seta N'\&op1 \\
\hline & 7 & DC C'Highest referenced element of SETSUB \(=\) \& \({ }^{\prime}\) \\
\hline & 8 & DC C'Number of sublist entries in OP1 = \& ' \\
\hline & 9 & mend \\
\hline 000000 00000 0004C & 10 a & csect \\
\hline & 11 & MAC1 (1, (3), (4)) \\
\hline 000000 C889878885A2A340 & 12+ & DC C'Highest referenced element of SETSUB \(=81\) \\
\hline 000028 D5A4948285994096 & 13+ & DC C'Number of sublist entries in OP1 \(=3{ }^{\prime}\) \\
\hline & 14 & end \\
\hline
\end{tabular}

Figure 94. Number Attribute Reference
N'\&op1 is equal to 3 because there are three subscripts in the macro operand in statement 11: 1, (3), and (4).
\(N ' \& S E T S U B\) is equal to 8 because \&SETSUB(8), assigned the value 70 in statement 4 , is the highest referenced element of the \&SETSUB array entries.

\section*{Notes:}
1. The number attribute reference can be used only in arithmetic expressions.
2. N'\&SYSLIST refers to the number of positional operands in a macro instruction, and \(\mathrm{N}^{\prime} \& S Y S L I S T(n)\) refers to the number of sublist entries in the \(n\)-th operand.
3. For positional macro parameters, either explicitly named or implicitly named as \&SYSLIST(n):
a. If the first character of an operand is a left parenthesis, count the number of unquoted and un-nested commas between it and the next matching right parenthesis. That number plus one is the number attribute of the operand.
b. If there is no initial left parenthesis, the number attribute is one.
4. For all other system variable symbols, the number attribute value is always one. This is also true for \&SYSMAC. The range of the subscript for \&SYSMAC is zero to \&SYSNEST inclusive.
5. \(\mathrm{N}^{\prime}\) is always zero for unsubscripted set symbols. The number attribute ( \(\mathrm{N}^{\prime}\) ), when used with a macro instruction operand, examines its list structure, not the number of characters in the operand. (The number of characters is determined by the count ( \(\mathrm{K}^{\prime}\) ) attribute.)

\section*{Data Attributes}

\section*{Defined Attribute (D')}

The defined attribute shows whether or not the ordinary symbol or literal referenced has been defined prior to the attribute reference. A symbol is defined if it has been encountered in the operand field of an EXTRN or WXTRN statement, or in the name field of any other statement except a TITLE statement or a macro instruction. A literal is defined if it has been encountered in the operand field of a machine instruction. The value of the defined attribute is an arithmetic value that can be assigned to a SETA symbol, and is equal to 1 if the symbol has been defined, or 0 if the symbol has not been defined.

The defined attribute can reference:
- Ordinary symbols not constructed by substitution
- Macro instruction operands
- SETC symbols whose value is an ordinary symbol
- System variable symbols whose value is an ordinary symbol
- Literals

The following is an example of how you can use the defined attribute:
\begin{tabular}{lll} 
Name & Operation & Operand \\
& AIF & (D'A).AROUND \\
A & LA & 1,4 \\
.AROUND & ANOP &
\end{tabular}

In this example, assuming there has been no previous definition of the symbol A, the statement labeled A would be assembled, since the conditional-assembly branch around it would not be taken. However, if by an AGO or AIF conditional-assembly branch the same statement were processed again, the statement at A would not be assembled:
\begin{tabular}{lll} 
Name & Operation & Operand \\
.UP & AIF & (D'A).AROUND \\
A & LA & 1,4 \\
.AROUND & ANOP & \\
& . & \\
& AGO & .UP
\end{tabular}

You can save assembly time using the defined attribute which avoids lookahead mode (see "Lookahead" on page 340 for more information. You can use the defined attribute in your program to prevent the assembler from making this time-consuming forward scan. This attribute reference can be used in the operand field of a SETA instruction or as one of the values in the operand field of a SETB or AIF instruction.

\section*{Operation Code Attribute ( \(\mathrm{O}^{\prime}\) )}

The operation code attribute shows whether a given operation code has been defined prior to the attribute reference. The operation code can be represented by a character string or by a variable symbol containing a character string. The variable must be set using a SETC assembler instruction prior to being referenced by the operation code \(\left(O^{\prime}\right)\) attribute.

The operation code attribute has a value of a single alphabetic character that shows the type of operation represented.

This attribute reference can be used in the operand field of the SETC instruction or as one of the values used in the operand field of a SETB or AIF instruction.

The following letters are used for the value of the operation code attribute:
A Assembler operation code
E Extended mnemonic operation code
M Macro definition
O Machine operation code
S Macro definition found in library
U Undefined, unknown, unassigned, or deleted operation code

\section*{Notes:}
1. The operation code ( \(\mathrm{O}^{\prime}\) ) attribute can only be used in a conditional assembly statement.
2. The assembler does not enter lookahead mode to resolve the operation code type, therefore only operation codes defined at the time the attribute is referenced return an operation code type value other than U .
3. When the operation code is not an assembler instruction or a machine instruction, and the operation code is not a previously defined macro, then all libraries in the library data set definition list are searched. This may have an adverse impact on the performance of the assembly, depending on the number of libraries assigned in the assembly job and the number of times the operation code attribute is used.

Examples:
\begin{tabular}{lll} 
Name & Operation & Operand \\
\& A & SETC & \(0^{\prime}\) MVC
\end{tabular}
\& A contains the letter 0 , because MVC is a machine operation code:
\begin{tabular}{lll} 
Name & Operation & Operand \\
\(\& A\) & SETC & 'DROP' \\
\(\& B\) & SETC & \(0^{\prime} \& A\)
\end{tabular}
\& contains the letter \(A\), because DROP is an assembler operation code.
The following example checks to see if the macro MAC1 is defined. If not, the MAC1 macro instruction is bypassed. This prevents the assembly from failing when the macro is not available.
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&CHECKIT & SETC & \(0^{\prime}\) 'MAC1 \\
& AIF & ('\&CHECKIT' EQ 'U').NOMAC \\
& MAC1 & \\
. NOMAC & ANOP & \\
&. &
\end{tabular}

Redefined Operation Codes: If an operation code is redefined using the OPSYN instruction then the value returned by a subsequent operation code attribute reference represents the new operation code. If the operation code is deleted using the OPSYN instruction then the value returned is \(U\).

\section*{Sequence Symbols}

You can use a sequence symbol in the name field of a statement to branch to that statement during conditional assembly processing, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can select the model statements from which the assembler generates assembler language statements for processing at assembly time.

A sequence symbol consists of a period (.) followed by an alphabetic character, followed by 0 to 61 alphanumeric characters.

\section*{Examples:}
```

.BRANCHING_LABEL\#1
.A

```

Sequence symbols can be specified in the name field of assembler language statements and model statements; however, sequence symbols must not be used as name entries in the following assembler instructions:
\begin{tabular}{lllll}
1 & ALIAS & EQU & OPSYN & SETC \\
1 & AREAD & ICTL & SETA & SETAF \\
1 & CATTR & LOCTR & SETB & SETCF
\end{tabular}

Also, sequence symbols cannot be used as name entries in macro prototype instructions, or in any instruction that already contains an ordinary or a variable symbol in the name field.

Sequence symbols can be specified in the operand field of an AIF or AGO instruction to branch to a statement with the same sequence symbol as a label.

Scope: A sequence symbol has a local scope. Thus, if a sequence symbol is used in an AIF or an AGO instruction, the sequence symbol must be defined as a label in the same part of the program in which the AIF or AGO instruction appears; that is, in the same macro definition or in open code.

Symbolic Parameters: If a sequence symbol appears in the name field of a macro instruction, and the corresponding prototype statement contains a symbolic parameter in the name field, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro definition. The value of the symbolic parameter is a null character string.

\section*{Example:}

\section*{Lookahead}
\begin{tabular}{|c|c|c|c|}
\hline \multirow{7}{*}{\begin{tabular}{l}
\&NAME \\
\&NAME
\end{tabular}} & \multicolumn{2}{|l|}{MACRO} & \multirow[b]{3}{*}{\begin{tabular}{l}
Statement 1 \\
Statement 2
\end{tabular}} \\
\hline & MOVE & \&TO, \&FROM & \\
\hline & ST & 2,SAVEAREA & \\
\hline & L & 2,\&FROM & \\
\hline & ST & 2,\&T0 & \\
\hline & L & 2,SAVEAREA & \\
\hline & \multicolumn{2}{|l|}{MEND} & \\
\hline . SYM & MOVE & FIELDA, FIELDB & Statement 3 \\
\hline + & ST & 2,SAVEAREA & Statement 4 \\
\hline + & L & 2,FIELDB & \\
\hline + & ST & 2,FIELDA & \\
\hline + & L & 2,SAVEAREA & \\
\hline
\end{tabular}

The symbolic parameter \&NAME is used in the name field of the prototype statement (Statement 1) and the first model statement (Statement 2). In the macro instruction (Statement 3), a sequence symbol (.SYM) corresponds to the symbolic parameter \&NAME. \&NAME is not replaced by .SYM and, therefore, the generated statement (Statement 4) does not contain an entry in the name field.

\section*{Lookahead}

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

Lookahead mode is entered:
- When the assembler processes a conditional assembly instruction and encounters an attribute reference (other than \(\mathrm{D}^{\prime}\) and \(\mathrm{O}^{\prime}\) ) to an ordinary symbol that is not yet defined.
- When the assembler encounters a forward AGO or AIF branch in open code to a sequence symbol that is not yet defined.

Lookahead is a sequential, statement-by-statement, forward scan over the source text.

If the attribute reference is made in a macro, forward scan begins with the first source statement following the outermost macro instruction. During lookahead the assembler:
- Bypasses macro definition and generation
- Does not generate object text
- Does not perform open-code variable substitution
- Ignores AIF and AGO branch instructions
- Establishes interim data attributes for undefined symbols it encounters in operand fields of instructions. The data attributes are replaced when a symbol is subsequently encountered in definition mode.

Lookahead mode ends when the desired symbol or sequence symbol is found, or when the END statement or end of file is reached. All statements read by
lookahead are saved on an internal file, and are fully processed when the lookahead scan ends.

If a COPY instruction is encountered during lookahead, it is fully processed at that time, the assembler copies the statements from the library, scans them, and saves them on the lookahead file. When lookahead mode has ended any COPY instructions saved to the lookahead file are ignored, as the statements from the copy member have already been read and saved to the lookahead file.

If a variable symbol is used for the member name of a COPY that is expanded during lookahead, the value of the variable symbol at the time the COPY is expanded is used.

For purposes of attribute definition, a symbol is considered partially defined if it depends in any way upon a symbol not yet defined. For example, if the symbol is defined by a forward EQU that is not yet resolved, that symbol is assigned a type attribute of \(U\).

In this case, it is quite possible that, by the end of the assembly, the type attribute has changed to some other value.

Generating END statements: Because no variable symbol substitution is carried out during lookahead, you should consider the following effects of using macro, AINSERT or open code substitution to generate END statements that separate source modules assembled in one job step (BATCH assembler option). If a symbol is undefined within a module, lookahead might read statements past the point where the END statement is to be generated. Lookahead stops when:
1. It finds the symbol
2. It finds an END statement
3. It reaches the end of the source input data set

In the first two cases, the assembler begins the next module at the statement after lookahead stopped, which could be after the point where you wanted to generate the END statement.

\section*{Lookahead Restrictions}

The assembler analyzes the statements it processes during lookahead, only to establish attributes of symbols in their name fields.

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

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

\section*{Sequence Symbols}

The conditional assembly instructions AGO and AIF in open code control the sequence in which source statements are processed. Using these instructions it is possible to branch back to a sequence symbol label and re-use previously processed statements. Due to operating system restrictions, the primary input source can only be read sequentially, and cannot be re-read. Whenever a sequence symbol in the name field is encountered in open code, the assembler must assume that all subsequent statements may need to be processed more than once. The assembler uses the lookahead file to save the statement containing the sequence symbol label and all subsequent statements as they are read and processed. Any subsequent AGO or AIF to a previously encountered sequence symbol is resolved to an offset into the lookahead file and input continues from that point.

\section*{Open Code}

Conditional assembly instructions in open code let you:
- Select, during conditional assembly, statements or groups of statements from the open code portion of a source module according to a predetermined set of conditions. The assembler further processes the selected statements at assembly time.
- Pass local variable information from open code through parameters into macro definitions.
- Control the computation in and generation of macro definitions using global SET symbols.
- Substitute values into the model statements in the open code of a source module and control the sequence of their generation.

All the conditional assembly elements and instructions can be specified in open code.

The specifications for the conditional assembly language described in this chapter also apply in open code. However, the following restrictions apply:

To Attributes In Open Code: For ordinary symbols, only references to the type, length, scale, integer, defined, and operation code attributes are allowed.

References to the number attribute have no meaning in open code, because \&SYSLIST is not allowed in open code, and symbolic parameters have no meaning in open code.

To Conditional Assembly Expressions: Figure 95 shows the restrictions for different expression types.

Figure 95 (Page 1 of 2). Restrictions on Coding Expressions in Open Code
\begin{tabular}{ll}
\hline Expression & Must not contain \\
\hline Arithmetic & - \&SYSLIST \\
(SETA) & - Symbolic parameters \\
& - Any attribute references to symbolic parameters, or system variable \\
& symbols with local scope \\
\hline
\end{tabular}

Figure 95 (Page 2 of 2). Restrictions on Coding Expressions in Open Code
\begin{tabular}{ll}
\hline Expression & Must not contain \\
\hline Character & - System variables with local scope \\
(SETC) & - Attribute references to system variables with local scope \\
& - Symbolic parameters \\
& - Arithmetic expressions with the items listed above \\
\hline Logical & - Character expressions with the items listed above \\
\hline
\end{tabular}

\section*{Conditional Assembly Instructions}

The remainder of this chapter describes, in detail, the syntax and rules for use of each conditional assembler instruction. The following table lists the conditional assembler instructions by type, and provides the page number where the instruction is described in detail.

Figure 96. Assembler Instructions
\begin{tabular}{llr|}
\hline Type of Instruction & Instruction & Page No. \\
\hline Establishing SET symbols & GBLA & 344 \\
\cline { 2 - 3 } & GBLB & 344 \\
\cline { 2 - 3 } & GBLC & 344 \\
\hline & LCLA & 345 \\
\hline & LCLB & \(\boxed{345}\) \\
\hline & LCLC & 345 \\
\hline & SETA & 347 \\
\hline Branching & SETB & 362 \\
\hline & SETC & 369 \\
\hline & ACTR & 394 \\
\hline & AGO & 392 \\
\hline & AIF & 390 \\
\hline & ANOP & 395 \\
\hline & SETAF & 388 \\
\hline & SETCF & 389 \\
\hline
\end{tabular}

\section*{Declaring SET Symbols}

You must declare a global SET symbol before you can use it. The assembler assigns an initial value to a global SET symbol at its first point of declaration.

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

\section*{GBLA, GBLB, and GBLC Instructions}

Use the GBLA, GBLB, and GBLC instructions to declare the global SETA, SETB, and SETC symbols you need. The SETA, SETB, and SETC symbols are assigned the initial values of 0,0 , and null character string, respectively.

sequence_symbol
is a sequence symbol.
variable_symbol
is a variable symbol, with or without the leading ampersand (\&).
These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

Any variable symbols declared in the operand field have a global scope. They can be used as SET symbols anywhere after the pertinent GBLA, GBLB, or GBLC instructions. However, they can be used only within those parts of a program in which they have been declared as global SET symbols; that is, in any macro definition and in open code.

The assembler assigns an initial value to the SET symbol only when it processes the first GBLA, GBLB, or GBLC instruction in which the symbol appears. Later GBLA, GBLB, or GBLC instructions do not reassign an initial value to the SET symbol.

Multiple GBLx statements can declare the same variable symbol so long as only one declaration for a given symbol is encountered during the expansion of a macro.

The following rules apply to the global SET variable symbol:
- Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any local variable symbol declared within the same local scope.
- The same variable symbol must not be declared or used as two different types of global SET symbol; for example, as a SETA or SETB symbol.
- A global SET symbol should not begin with \(\& S Y S\) because these characters are used for system variable symbols.

\section*{Subscripted Global SET Symbols}

A global subscripted SET symbol is declared by the GBLA, GBLB, or GBLC instruction.

sequence_symbol
is a sequence symbol.
variable_symbol
is a variable symbol, with or without the leading ampersand (\&).
dimension
is the dimension of the array. It must be an unsigned, decimal, self-defining term greater than zero.

\section*{Example:}
\[
\text { GBLA } \quad \& G A(25), \& G A 1(15)
\]

There is no limit on the maximum subscript allowed. Also, the limit specified in the global declaration (GBLx) can be exceeded. The dimension shows the number of SET variables associated with the subscripted SET symbol. The assembler assigns an initial value to every variable in the array thus declared.

\section*{Notes:}
1. Global arrays are assigned initial values only by the first global declaration processed, in which a global subscripted SET symbol appears.
2. A subscripted global SET symbol can be used only if the declaration has a subscript, which represents a dimension; an unsubscripted global SET symbol can be used only if the declaration had no subscript, except for a number attribute reference to the name of a dimensioned SET symbol.

\section*{Alternative Format for GBLx Statements}

The assembler permits the alternative statement format for GBLx instructions:
\begin{tabular}{lll} 
& Cont. \\
GBLA & \&GLOBAL_SYMBOL_FOR_DC_GEN, & X \\
& \&LOOP_CONTRL_A, & X \\
& \&VALUE_PASSED_TO_FIDO, & X \\
& \&VALUE_RETURNED_FROM_FIDO &
\end{tabular}

\section*{LCLA, LCLB, and LCLC Instructions}

Use the LCLA, LCLB, and LCLC instructions to declare the local SETA, SETB, and SETC symbols you need. The SETA, SETB, and SETC symbols are assigned the initial values of 0,0 , and null character string, respectively.

sequence_symbol
is a sequence symbol.
variable_symbol
is a variable symbol, with or without the leading ampersand (\&).
These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

Any variable symbols declared in the operand field have a local scope. They can be used as SET symbols anywhere after the pertinent LCLA, LCLB, or LCLC instructions, but only within the declared local scope. Multiple LCLx statements can declare the same variable symbol so long as only one declaration for a given symbol is encountered during the expansion of a macro.

The following rules apply to a local SET variable symbol:
- Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any global variable symbol declared within the same local scope.
- The same variable symbol must not be declared or used as two different types of SET symbols; for example, as a SETA and a SETB symbol, within the same local scope.
- A local SET symbol should not begin with \&SYS because these characters are used for system variable symbols.

\section*{Subscripted Local SET Symbols}

A local subscripted SET symbol is declared by the LCLA, LCLB, or LCLC instruction.

sequence_symbol
is a sequence symbol.
variable_symbol
is a variable symbol, with or without the leading ampersand (\&).

\section*{dimension}
is the dimension of the array. It must be an unsigned, decimal, self-defining term greater than zero.

\section*{Example:}

There is no limit to SET symbol dimensioning. The limit specified in the explicit (LCLx) or implicit (SETx) declaration can also be exceeded by later SETx statements. The dimension shows the number of SET variables associated with
the subscripted SET symbol. The assembler assigns an initial value to every variable in the array thus declared.

Subscripted Local SET Symbol: A subscripted local SET symbol can be used only if the declaration has a subscript, which represents a dimension; an unsubscripted local SET symbol can be used only if the declaration had no subscript, except for a number attribute reference to the dimensioned SET symbol.

\section*{Alternative Format for LCLx Statements}

The assembler permits an alternative statement format for LCLx instructions:
\begin{tabular}{clc} 
& & Cont. \\
LCLA & \&LOCAL_SYMBOL_FOR_DC_GEN, & \(x\) \\
& \&COUNTER_FOR_INNER_LOOP, & \(x\) \\
& \&COUNTER_FOR_OUTER_LOOP, & X \\
& \&COUNTER_FOR_TRAILING_LOOP &
\end{tabular}

\section*{Assigning Values to SET Symbols}

You can assign values to SET symbols by using the SETA, SETB, SETC, SETAF and SETCF instructions (SETx). You can also use these instructions to implicitly define local SET symbols. Local SET symbols need not be declared explicitly with LCLA, LCLB, or LCLC statements. The assembler considers any undeclared variable symbol found in the name field of a SETx statement to be a local SET symbol. It is given the initial value specified in the operand field of SETA, SETB and SETC instructions, and the value returned from the external function specified in the operand of SETAF and SETCF instructions. If the symbol in the name field is subscripted, it is declared as a subscripted SET symbol.

Note that spaces do not terminate the operand field when used in logical expressions and in build-in functions. For more information, see "Logical (SETB) Expressions" on page 365 .

\section*{SETA Instruction}

The SETA instruction assigns an arithmetic value to a SETA symbol. You can specify a single value or an arithmetic expression from which the assembler computes the value to assign.

You can change the values assigned to an arithmetic or SETA symbol. This lets you use SETA symbols as counters, indexes, or for other repeated computations that require varying values.

variable_symbol
is a variable symbol.
A global variable symbol in the name field must have been previously declared as a SETA symbol in a GBLA instruction. Local SETA symbols need not be declared in a LCLA instruction. The assembler considers any undeclared variable symbol found in the name field of a SETA instruction as a local SET symbol. The variable symbol is assigned a type attribute value of N .

\section*{SETA Instruction}

\section*{expression}
is an arithmetic expression evaluated as a signed 32 -bit arithmetic value that is assigned to the SETA symbol in the name field. The minimum and maximum allowable values of the expression are \(-2^{31}\) and \(+2^{31}-1\), respectively.

Figure 97 defines an arithmetic expression.


Figure 97. Defining Arithmetic (SETA) Expressions
Figure 98 shows the variable symbols that are allowed as terms in an arithmetic expression.

Figure 98 (Page 1 of 2). Variable Symbols Allowed as Terms in Arithmetic Expressions
\begin{tabular}{llll} 
Variable symbol & Restrictions & Example & Valid value \\
\hline SETA & None & --- & --- \\
\hline SETB & None & --- & --- \\
\hline SETC & \begin{tabular}{l} 
Value must evaluate to an \\
unsigned binary, \\
hexadecimal or decimal \\
self-defining term
\end{tabular} & 123 & 123 \\
\hline Symbolic parameters & \begin{tabular}{l} 
Value must be a \\
self-defining term
\end{tabular} & \&PARAM & X'A1' \(^{\text {CSUBLIST(3) }}\)
\end{tabular}

Figure 98 (Page 2 of 2). Variable Symbols Allowed as Terms in Arithmetic Expressions
\begin{tabular}{|c|c|c|c|}
\hline Variable symbol & Restrictions & Example & Valid value \\
\hline \&SYSLIST( \(n\) ) & Corresponding operand or sublist entry must be a & \&SYSLIST(3) & 24 \\
\hline \&SYSLIST( \(n, m\) ) & self-defining term & \&SYSLIST (3,2) & B'101' \\
\hline \&SYSDATC & None & --- & --- \\
\hline \&SYSM_HSEV & & & \\
\hline \&SYSM_SEV & & & \\
\hline \&SYSNDX & & & \\
\hline \&SYSNEST & & & \\
\hline \&SYSOPT_DBCS & & & \\
\hline \&SYSOPT_RENT & & & \\
\hline \&SYSOPT_XOBJECT & & & \\
\hline \&SYSSTMT & & & \\
\hline
\end{tabular}

The following example shows a SETA statement with a valid self-defining term in its operand field:
```

\&ASYM1 SETA C'D' \&ASYM1 has value 196 (C'D')

```

The second statement in the following example is valid because in the two positions in the SETA operand where a term is required (either side of the + sign), the assembler finds a valid self-defining term:
```

\&CSYM2 SETC 'C''A''' \&CSYM2 has value C'A'
\&ASYM3 SETA \&CSYM2+\&CSYM2 \&ASYM3 has value 386 (C'A' + C'A')

```

A SET statement is not rescanned by the assembler to see whether substitutions might affect the originally-determined syntax. The original syntax of the self-defining term must be correct. Therefore the assembler does not construct a self-defining term in a SETA statement. The third statement of the next example shows this:
\begin{tabular}{llll} 
\&CSYM3 & SETC & '3' & \&CSYM has value 3 (C'3') \\
\&ASYM3 & SETA & \&CSYM3 & \&ASYM has value 3 \\
\&ASYM4 & SETA & C'\&ASYM3' & Invalid self-defining term
\end{tabular}

In this example \(C^{\prime} \& A S Y M 3\) ' is not a valid term.

\section*{Subscripted SETA Symbols}

The SETA symbol in the name field can be subscripted, but only if the same SETA symbol has been previously declared in a GBLA or LCLA instruction with an allowable dimension.

The assembler assigns the value of the expression in the operand field to the position in the declared array given by the value of the subscript. The subscript expression must not be 0 or have a negative value.

\section*{Introducing Built-In Functions}

The assembler provides built-in functions for the SETA, SETB and SETC expressions.

Each function returns one value - an arithmetic value for SETA, a binary bit for SETB, and a character string for SETC.

There are two different forms of invocation for the built-in functions:
- The logical-expression format encloses the function and operands in parentheses. In the unary format, the function is followed by the one operand. In the binary format, the function is placed between the two operands. For both unary and binary formats, the function is separated from the operand or operands by spaces.

\section*{Logical-expression unary format}
-(-built-in function-operand-) \(\qquad\)

\section*{Logical-expression binary format}
\(\rightarrow\) (-operand-built-in function-operand-) \(\longrightarrow\)
(A OR B) and (\&J SLL 2) are examples of binary logical-expression format functions, and (NOT C) and (SIGNED \&J) are examples of unary logical-expression format functions.
- The function-invocation format has the function first, followed by one or more operands in parentheses.


FIND('abcde', ' \(\mathrm{d}^{\prime}\) ) is an example of a function-invocation format. (The equivalent logical-expression format is ('abcde' FIND 'd').)
Spaces are not allowed between the arguments of functions in function-invocation format.

In either format, the operand is an expression of the type expected by the built-in function. (The particular details of the number of operands and the operand type are provided with the information for each built-in function.)

Some functions are available in one format, some are available in both. Figure 99, which provides a summary of all the built-in functions, shows the forms in which a function is available.

Figure 99 (Page 1 of 3). Summary of Built-In Functions
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline Function & Type & L-E \({ }^{1}\) & F-I \({ }^{2}\) & Result \({ }^{3}\) & Operands \({ }^{3}\) & Page \\
\hline A2B & Representation conversion & & \(\checkmark\) & C & A & 374 \\
\hline A2C & Representation conversion & & \(\checkmark\) & C & A & 375 \\
\hline A2D & Representation conversion & & \(\checkmark\) & C & A & 375 \\
\hline A2X & Representation conversion & & \(\checkmark\) & C & A & 375 \\
\hline AND & Logical & \(\checkmark\) & & A & A & 353 \\
\hline AND & Logical & \(\checkmark\) & & B & B & 365 \\
\hline AND NOT & Logical & \(\checkmark\) & & B & B & 365 \\
\hline B2A & Representation conversion & & \(\checkmark\) & A & C & 353 \\
\hline
\end{tabular}

Figure 99 (Page 2 of 3). Summary of Built-In Functions
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline Function & Type & L-E \({ }^{1}\) & F-I² & Result \({ }^{3}\) & Operands \({ }^{3}\) & Page \\
\hline B2C4 & Representation conversion & & \(\checkmark\) & C & C & 375 \\
\hline B2D & Representation conversion & & \(\checkmark\) & C & C & 376 \\
\hline B2X \({ }^{4}\) & Representation conversion & & \(\checkmark\) & C & C & 376 \\
\hline BYTE & Representation conversion & \(\checkmark\) & \(\checkmark\) & C & A & 376 \\
\hline C2A & Representation conversion & & \(\checkmark\) & A & C & 353 \\
\hline C2B4 & Representation conversion & & \(\checkmark\) & C & C & 377 \\
\hline C2D & Representation conversion & & \(\checkmark\) & C & C & 377 \\
\hline \(\mathrm{C} 2 \mathrm{X}^{4}\) & Representation conversion & & \(\checkmark\) & C & C & 377 \\
\hline D2A & Representation conversion & & \(\checkmark\) & A & C & 354 \\
\hline D2B & Representation conversion & & \(\checkmark\) & C & C & 377 \\
\hline D2C & Representation conversion & & \(\checkmark\) & C & C & 378 \\
\hline D2X & Representation conversion & & \(\checkmark\) & C & C & 378 \\
\hline DCLEN & String manipulation & & \(\checkmark\) & A & C & 354 \\
\hline DCVAL & String manipulation & & \(\checkmark\) & C & C & 378 \\
\hline DEQUOTE & String manipulation & & \(\checkmark\) & C & C & 379 \\
\hline DOUBLE & String manipulation & \(\checkmark\) & \(\checkmark\) & C & C & 379 \\
\hline FIND & String scanning & \(\checkmark\) & \(\checkmark\) & A & C & 355 \\
\hline INDEX & String scanning & \(\checkmark\) & \(\checkmark\) & A & C & 355 \\
\hline ISBIN & Validity checking & & \(\checkmark\) & B & C & 365 \\
\hline ISDEC & Validity checking & & \(\checkmark\) & B & C & 366 \\
\hline ISHEX & Validity checking & & \(\checkmark\) & B & C & 366 \\
\hline ISSYM & Validity checking & & \(\checkmark\) & B & C & 366 \\
\hline LOWER & String manipulation & \(\checkmark\) & \(\checkmark\) & C & C & 379 \\
\hline NOT & Logical & \(\checkmark\) & & A & A & 356 \\
\hline NOT & Logical & \(\checkmark\) & & B & B & 366 \\
\hline OR & Logical & \(\checkmark\) & & A & A & 356 \\
\hline OR & Logical & \(\checkmark\) & & B & B & 366 \\
\hline OR NOT & Logical & \(\checkmark\) & & B & B & 367 \\
\hline SIGNED & Representation conversion & \(\checkmark\) & \(\checkmark\) & C & A & 380 \\
\hline SLA & Shift & \(\checkmark\) & & A & A & 356 \\
\hline SLL & Shift & \(\checkmark\) & & A & A & 356 \\
\hline SRA & Shift & \(\checkmark\) & & A & A & 357 \\
\hline SRL & Shift & \(\checkmark\) & & A & A & 357 \\
\hline SYSATTRA & Information retrieval & & \(\checkmark\) & C & C & 380 \\
\hline SYSATTRP & Information retrieval & & \(\checkmark\) & C & C & 380 \\
\hline UPPER & String manipulation & \(\checkmark\) & \(\checkmark\) & C & C & 380 \\
\hline X2A & Representation conversion & & \(\checkmark\) & A & C & 358 \\
\hline X2B4 & Representation conversion & & \(\checkmark\) & C & C & 381 \\
\hline
\end{tabular}

Figure 99 (Page 3 of 3). Summary of Built-In Functions
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline Function & Type & L-E \({ }^{1}\) & F-I \({ }^{2}\) & Result \({ }^{3}\) & Operands \({ }^{3}\) & Page \\
\hline X2C4 & Representation conversion & & \(\checkmark\) & C & C & 381 \\
\hline X2D & Representation conversion & & \(\checkmark\) & C & C & 381 \\
\hline XOR & Logical & \(\checkmark\) & & A & A & 358 \\
\hline XOR & Logical & \(\checkmark\) & & B & B & 367 \\
\hline XOR NOT & Logical & \(\checkmark\) & & B & B & 367 \\
\hline
\end{tabular}

\section*{Notes:}
1. If \(a v\) is in this column, the function is available in the "logical-expression" format.
2. If \(a \sqrt{ }\) is in this column, the function is available in the "function-invocation" format.
3. Possible values in these columns are:

A Arithmetic
B Binary
C Character
4. For these functions, the maximum length of the operand (and output) is the maximum string length that the assembler supports, currently 1024.

\section*{Arithmetic (SETA) Expressions}

Figure 100 shows how arithmetic expressions can be used.

Figure 100. Use of Arithmetic Expressions
\begin{tabular}{|c|c|c|}
\hline Used in & Used as & Example \\
\hline SETA instruction & Operand & \&A1 SETA \& \(41+2\) \\
\hline AIF or SETB instruction & Term in arithmetic relation & AIF (\&A*10 GT 30).A \\
\hline Subscripted SET symbols & Subscript & \&ASYM (\&A \(+10-\& C\) ) \\
\hline Substring notation & Subscript & 'STRING' (\&A*2, \& \(4-1\) ) \\
\hline Sublist notation & Subscript & Given sublist ( \(A, B, C, D\) ) named \&PARAM, if \(\& A=1\) then \(\& \operatorname{PARAM}(\& A+1)=B\) \\
\hline \&SYSLIST & Subscript & \begin{tabular}{l}
\&SYSLIST ( \& M \(+1, \& N-2\) ) \\
\&SYSLIST(N'\&SYSLIST)
\end{tabular} \\
\hline SETC instruction & Character string in operand & \begin{tabular}{l}
Given \&C SETC '5-10*\&A' 1 if \(\& A=10\) then \& \(\mathrm { C } = 5 - 1 0 * 1 0 \longdiv { 2 }\) \\
Given \&D SETC '5-10*\&A' 1 if \(\& A=-10\) then \& \(D=5-10 * 103\)
\end{tabular} \\
\hline Built-in functions & Operand & \&VAR SETA (NOT \&OP1) \&VAR SETA BYTE(64) \\
\hline
\end{tabular}

When an arithmetic expression is used in the operand field of a SETC instruction (see 1 in Figure 100), the assembler assigns the character value representing the arithmetic expression to the SETC symbol, after substituting values (see \(\mathbf{2}\) in Figure 100) into any variable symbols. It does not evaluate the arithmetic expression. The mathematical sign (+ or - ) is not included in the substituted value
of a variable symbol (see 3 in Figure 100), and any insignificant leading zeros are removed.

Here are the built-in functions for arithmetic (SETA) expressions:
AND
Format: Logical-expression
Operands: Arithmetic
Output: (aexpr1 AND aexpr2) provides an arithmetic result where each bit position in the result is set to 1 if the corresponding bit positions in both operands contain 1 , otherwise, the result bit is set to 0 .

\section*{Example}

After the following statements \&VAR contains the arithmetic value +2 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 AND \&OP2)
\end{tabular}

B2A
Format: Function-invocation
Operands: Character
Output: B2A('bitstring') converts a character string argument containing ' 0 ' and ' 1 ' characters to an arithmetic value.
- Fewer than 32 characters are padded internally on the left with ' 0 ' characters to a length of 32 characters.
- Error conditions are detected if the argument contains invalid characters, or if the argument length exceeds 32 characters, generating the message ASMA214E.
- Null argument strings return zero.

The result of the B2A function is the same as
\&value SETA B'bitstring'
except that null strings are allowed by B2A but not by SETA.
Examples
B2A('') has value 0
B2A('0000000101') has value 5
B2A('11111111111111111111111111111110') has value -2
C2A
Format: Function-invocation
Operands: Character
Output: C2A('charstring') converts a character string of zero to four characters to a binary arithmetic value having the same bit pattern.
- Fewer than four characters are padded internally on the left with EBCDIC null characters to a length of four characters.
- An error condition is detected if the argument length exceeds 4 characters, generating the message ASMA214E.
- Null argument strings return zero.
```

|

```

The result of C2A is the same as would be obtained from
\&value SETA C'charstring'
except that C2A gives a zero result for null strings, and does not pair apostrophes or ampersands before conversion.

\section*{Example}
\begin{tabular}{ll} 
C2A('') & has value 0 \\
C2A('+') & has value 78 \\
C2A('1') & has value 241 \\
C2A('0000') & has value -252645136
\end{tabular}

\section*{D2A}

Format: Function-invocation
Operands: Character
Output: D2A('decstring') converts a character string argument containing an optional leading plus or minus sign followed by decimal digits to an arithmetic value. Error conditions are detected if
- the argument contains invalid characters
- no digits are present following a sign
- the argument length exceeds 11 characters
- the resulting value is too large
- the argument string is null.

The result of the D2A function is the same as
\&value SETA decstring
except that SETA does not allow leading plus or minus signs.

\section*{Examples}

D2A(' ') indicates an error condition
D2A('000') has value 0
D2A('10') has value 10
D2A('+100') has value 100
D2A('-5') has value -5

\section*{DCLEN}

\section*{Format: Function-invocation \\ Operands: Character}

Output: DCLEN('cexpr') returns the length of its argument string after pairs of apostrophes and ampersands have been internally replaced by single occurrences. No change is made to the argument. Such pairing occurs only once; that is, three successive occurrences of an apostrophe or ampersand will result in two occurrences, not one.

\section*{Examples}
```

DCLEN('') has value 0 (null string)
DCLEN('''') has value 1 (argument is a single apostrophe)
DCLEN('''''') has value 2 (argument is two apostrophes)
DCLEN('\&\&') has value 1 (argument is two ampersands)
DCLEN('a''''b') has value 3 (DCVAL string would be "a'b")
DCLEN('a''''b\&\&C') has value 5 (DCVAL string would be "a'b\&c")
DCLEN('\&\&\&''''''') has value 4 (DCVAL string would be "\&\&''")

```

Note: DCLEN is similar to DCVAL, except that DCLEN returns only the length of the result, not the paired string.

\section*{FIND}

Format: Logical-expression, function-invocation
Operands: Character
Output: (string1 FIND string2) or FIND(string1,string2) finds the first match of any character from operand2 within operand1. The value returned by FIND indicates the position where the match occurs. FIND returns 0 if no match occurs or if either operand is a null string.

\section*{Examples}

After the following statements \&VAR contains the arithmetic value 3 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETC & 'abcdef' \\
\&OP2 & SETC & 'cde' \\
\&VAR & SETA & ('\&OP1' FIND '\&OP2')
\end{tabular}

In the above example the character c in \&OP2 is the first character found in \&OP1. Consider the following example where the character c , in \&OP1, has been replaced with the character g .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETC & 'abcdef' \\
\&OP2 & SETC & 'gde' \\
\&VAR & SETA & ('\&OP1' FIND '\&OP2')
\end{tabular}
\&VAR contains the arithmetic value 4 . The character \(d\) in \&OP2 is the first character found in \&OP1.
In the following example, the ordering of the characters in the second operand is changed to egd.
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETC & 'abcdef' \\
\&OP2 & SETC & 'egd' \\
\&VAR & SETA & FIND('\&OP1', '\&OP2')
\end{tabular}
\&VAR still contains the arithmetic value 4. Because FIND is looking for a single character from the character string, the order of the characters in the second operand string is irrelevant.

\section*{INDEX}

Format: Logical-expression, function-invocation
Operands: Character
Output: InDEX('cexpr1','cexpr2') or ('cexpr1' index 'cexpr2') locates the first occurrence of the second argument within the first argument, and returns the position of the match. A zero value is returned if:
- Either argument is null
- No match is found
- The second argument is longer than the first

\section*{Examples}
\begin{tabular}{ll} 
INDEX('ABC','B') & has value 2 \\
INDEX('ABC','D') & has value 0
\end{tabular}

NOT
Format: Logical-expression
Operands: Arithmetic
Operands: Arithmetic
Output: (NOT aexp) provides the ones complement of the value contained or evaluated in the operand.

\section*{Example}

After the following statements \&VAR contains the arithmetic value -11 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETA & 10 \\
\&VAR & SETA & (NOT \&OP1)
\end{tabular}

OR
Format: Logical-expression
Operands: Arithmetic
Output: Each bit position in the result is set to 1 if the corresponding bit positions in one or both operands contains a 1, otherwise the result bit is set to 0 .

\section*{Example}

After the following statements \&VAR contains the arithmetic value +10 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 OR \&OP2)
\end{tabular}

SLA
Format: Logical-expression
Operands: Arithmetic
Output: The 31-bit numeric part of the signed first operand is shifted left the number of bits specified in the rightmost six bits of the second operand. The sign of the first operand remains unchanged. Zeros are used to fill the vacated bit positions on the right.

Example
After the following statements \&VAR contains the arithmetic value +8 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & 2 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 SLA \&OP2)
\end{tabular}

SLL
Format: Logical-expression
Operands: Arithmetic
Output: (aexp1 SLL aexp2) shifts the 32-bit first operand left the number of bits specified in the rightmost six bits of the second operand. Bits shifted out of bit position 0 are lost. Zeros are used to fill the vacated bit positions on the right.

\section*{Example}

After the following statements \&VAR contains the arithmetic value +40 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 SLL \&OP2)
\end{tabular}

\section*{SRA}

Format: Logical-expression
Operands: Arithmetic
Output: The 31-bit numeric part of the signed first operand is shifted right the number of bits specified in the rightmost six bits of the second operand. The sign of the first operand remains unchanged. Bits shifted out of bit position 31 are lost. Bits equal to the sign are used to fill the vacated bit positions on the left.

\section*{Examples}

After the following statements \&VAR contains the arithmetic value +2 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 SRA \&OP2)
\end{tabular}

After the following statements \&VAR contains the arithmetic value -1 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & -344 \\
\&OP2 & SETA & 40 \\
\&VAR & SETA & (\&OP1 SRA \&OP2)
\end{tabular}

Compare this result with the result in the second example under SRL below.
SRL
Format: Logical-expression
Operands: Arithmetic
Output: The 32-bit first operand is shifted right the number of bits specified in the rightmost six bits of the second operand. Bits shifted out of bit position 31 are lost. Zeros are used to fill the vacated bit positions on the left.

\section*{Examples}

After the following statements \&VAR contains the arithmetic value +2 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 SRL \&OP2)
\end{tabular}

After the following statements \&VAR contains the arithmetic value 0 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & -344 \\
\&OP2 & SETA & 40 \\
\&VAR & SETA & (\&OP1 SRL \&OP2)
\end{tabular}

X2A
Format: Function-invocation
Operands: Character
Output: X2A('hexstring') converts a character string argument containing hexadecimal digits to an arithmetic value.
- If the character string contains fewer than 8 characters, it is padded internally on the left with ' 0 ' characters.
- Error conditions are detected if the argument contains invalid characters, or if the argument length exceeds 8 characters, generating the message ASMA214E.
- Null argument strings return zero.

The result of the X2A function is the same as
\&value SETA X'hexstring'
except that null strings are allowed by X2A but not by SETA.
Examples
```

X2A('00000101') has value 257
X2A('C1') has value 193
X2A('') has value 0
X2A('FFFFFFF0') has value -16

```

\section*{XOR}

Format: Logical-expression
Operands: Arithmetic
Output: Each bit position in the result is set to 1 if the corresponding bit positions in the two operands are unlike, otherwise the result bit is set to 0 .
Example After the following statements \&VAR contains the arithmetic value +8 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
& & \\
\&OP1 & SETA & 10 \\
\&OP2 & SETA & 2 \\
\&VAR & SETA & (\&OP1 XOR \&OP2)
\end{tabular}

Rules for Coding Arithmetic Expressions: The following is a summary of coding rules for arithmetic expressions:
1. Unary (operating on one value) operators and binary (operating on two values) operators are allowed in arithmetic expressions.
2. An arithmetic expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression. The unary operators are + (positive) and - (negative).
3. The binary operators that can be used to combine the terms of an expression are + (addition), - (subtraction), * (multiplication), and / (division).
4. An arithmetic expression must not begin with a binary operator, and it must not contain two binary operators in succession.
5. An arithmetic-valued function is a term.
6. An arithmetic expression must not contain two terms in succession.
7. An arithmetic expression must not contain a decimal point. For example, 123.456 is not a valid arithmetic term, but 123 is.
8. An arithmetic expression must not contain spaces between an operator and a term, nor between two successive operators except for built-in functions using the "logical-expression format" described at "Logical (SETB) Expressions" on page 365
9. Ordinary symbols specified in arithmetic expressions must be defined before the arithmetic expression is encountered, and must have an absolute value.
10. An arithmetic expression can contain up to 24 unary and binary operators, and is limited to 255 levels of parentheses. The parentheses required for sublist notation, substring notation, and subscript notation count toward this limit.
An arithmetic expression must not contain two terms in succession; however, any term may be preceded by up to 24 unary operators. \(+\& A *-\& B\) is a valid operand for a SETA instruction. The expression \&FIELD+- is invalid because it has no final term.

Evaluation of Arithmetic Expressions: The assembler evaluates arithmetic expressions during conditional assembly processing as follows:
1. It evaluates each arithmetic term.
2. It carries out arithmetic operations from left to right. However,
a. It carries out unary operations before binary operations, and
b. It carries out the binary operations of multiplication and division before the binary operations of addition and subtraction.
3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives a 0 result.
4. In parenthesized arithmetic expressions, the assembler evaluates the innermost expressions first, and then considers them as arithmetic terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated.
5. The computed result, including intermediate values, must lie in the range \(-2^{31}\) through \(+2^{31}-1\). (Note that if the value \(-2^{31}\) is substituted in a SETC expression, its magnitude, 2147483648 , is invalid if subsequently substituted in a SETA expression.)

SETC Variables in Arithmetic Expressions: The assembler permits a SETC variable to be used as a term in an arithmetic expression if the character string value of the variable is a self-defining term. The value represented by the string is assigned to the arithmetic term. A null string is treated as zero.

\section*{Examples:}
\begin{tabular}{lll} 
& LCLC & \(\& C(5)\) \\
\(\& C(1)\) & SETC & 'B''101''' \\
\(\& C(2)\) & SETC & 'C''A''' \\
\(\& C(3)\) & SETC & '23' \\
\(\& A\) & SETA & \(\& C(1)+\& C(2)-\& C(3)\)
\end{tabular}

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

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

Assume that \(\& X\) is a character string with the value \(A B C\).
\begin{tabular}{lll}
\(\& I\) & SETC & 'C'''.'\&X' \((1,1) . '^{\prime \prime '}\) \\
\(\& V A L\) & SETA & \&TRANS(\&I)
\end{tabular}

The first statement sets \&I to C'A'. The second statement extracts the 193rd element of \&TRANS ( \(C^{\prime} A^{\prime}=X^{\prime} C 1 '=193\) ).

The following code converts a hexadecimal value in \&H into a decimal value in \&VAL:
\begin{tabular}{lll}
\(\& X\) & SETC & 'X''\&H' \({ }^{\prime \prime}\) \\
\(\& V A L\) & SETA & \(\& X\)
\end{tabular}

The following code converts the double-byte character Da into a decimal value in \&VAL. \&VAL can then be used to find an alternative code in a subscripted SETC variable:
\begin{tabular}{lll} 
\&DA & SETC & 'G''<Da>''' \\
\&VAL & SETA & \(\& D A\)
\end{tabular}

Although you can use a predefined absolute symbol as an operand in a SETA expression, you cannot substitute a SETC variable whose value is the same as the symbol. For example:
\begin{tabular}{llll} 
ABS & EQU & 5 & \\
\&ABS & SETA & ABS & \&ABS has value 5 \\
\&CABS & SETC & 'ABS' & \&CABS has value 'ABS' \\
\&ABS & SETA & \&CABS & \begin{tabular}{l} 
invalid usage
\end{tabular}
\end{tabular}

DBCS Assembler Option: The G-type self-defining term is valid only if the DBCS assembler option is specified.

\section*{Using SETA symbols}

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic expression. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is converted to a character string containing its absolute value, with leading zeros removed. If the value is 0 , it is converted to a single 0 .

\section*{Example:}
\begin{tabular}{|c|c|c|c|}
\hline \multirow{3}{*}{\&NAME} & MACRO & & \\
\hline & MOVE & \&TO, \&FROM & \\
\hline & LCLA & \& \(, \& B, \& C, \& D\) & \\
\hline \& \({ }^{\text {A }}\) & SETA & 10 & Statement 1 \\
\hline \& \({ }^{\text {c }}\) & SETA & 12 & Statement 2 \\
\hline \&C & SETA & \& \(A-\& B\) & Statement 3 \\
\hline \&D & SETA & \& \(A+\& C\) & Statement 4 \\
\hline \multirow[t]{5}{*}{\&NAME} & ST & 2,SAVEAREA & \\
\hline & L & 2,\&FROM\&C & Statement 5 \\
\hline & ST & 2,\&TO\&D & Statement 6 \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & FIELDA, FIELDB & \\
\hline +HERE ST & & 2,SAVEAREA & \\
\hline + L & & 2,FIELDB2 & \\
\hline + ST & & 2,FIELDA8 & \\
\hline \(+\quad \mathrm{L}\) & & 2,SAVEAREA & \\
\hline
\end{tabular}

Statements 1 and 2 assign the arithmetic values +10 and +12 , respectively, to the SETA symbols \&A and \&B. Therefore, statement 3 assigns the SETA symbol \&C the arithmetic value -2 . When \(\& C\) is used in statement 5 , the arithmetic value -2 is converted to the character 2 . When \&C is used in statement 4 , however, the arithmetic value -2 is used. Therefore, \(\& D\) is assigned the arithmetic value +8 . When \(\& D\) is used in statement 6 , the arithmetic value +8 is converted to the character 8.

The following example shows how the value assigned to a SETA symbol may be changed in a macro definition.
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & & \\
\hline \multirow[t]{2}{*}{\&NAME} & MOVE & \&TO, \&FROM & \\
\hline & LCLA & \& A & \\
\hline \& A & SETA & 5 & Statement 1 \\
\hline \multirow[t]{2}{*}{\&NAME} & ST & 2,SAVEAREA & \\
\hline & L & 2,\&FROM\&A & Statement 2 \\
\hline \multirow[t]{4}{*}{\& \(A\)} & SETA & 8 & Statement 3 \\
\hline & ST & 2,\&T0\&A & Statement 4 \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & FIELDA, FIE & \\
\hline \multicolumn{2}{|l|}{+HERE ST} & 2,SAVEAREA & \\
\hline \(+\quad \mathrm{L}\) & & 2,FIELDB5 & \\
\hline + ST & & 2,FIELDA8 & \\
\hline \(+\quad \mathrm{L}\) & & 2,SAVEAREA & \\
\hline
\end{tabular}

Statement 1 assigns the arithmetic value +5 to SETA symbol \&A. In statement \(2, \& A\) is converted to the character 5 . Statement 3 assigns the arithmetic value +8 to \(\& A\). In statement 4, therefore, \(\& A\) is converted to the character 8 , instead of 5 .

A SETA symbol may be used with a symbolic parameter to refer to an operand in an operand sublist. If a SETA symbol is used for this purpose, it must have been assigned a positive value.

Any expression that may be used in the operand field of a SETA instruction may be used to refer to an operand in an operand sublist. Sublists are described in "Sublists in Operands" on page 304

The following macro definition adds the last operand in an operand sublist to the first operand in an operand sublist and stores the result at the first operand. A sample macro instruction and generated statements follow the macro definition.

\&NUMBER is the first symbolic parameter in the operand field of the prototype statement (statement 1). The corresponding characters ( \(A, B, C, D, E\) ) of the macro instruction (statement 4) are a sublist. Statement 2 assigns to \&LAST the arithmetic value +5 , which is equal to the number of operands in the sublist. Therefore, in statement \(3, \& N U M B E R(\& L A S T)\) is replaced by the fifth operand of the sublist.

\section*{SETB Instruction}

Use the SETB instruction to assign a bit value to a SETB symbol. You can assign the bit values, 0 or 1 , to a SETB symbol directly and use it as a switch.

If you specify a logical (Boolean) expression in the operand field, the assembler evaluates this expression to determine whether it is true or false, and then assigns the value 1 or 0 , respectively, to the SETB symbol. You can use this computed value in condition tests or for substitution.
\(\rightarrow\) variable_symbol—SETB—binary_value— \(\longrightarrow\)
variable_symbol
is a variable symbol.
A global variable symbol in the name field must have been previously declared as a SETB symbol in a GBLB instruction. Local SETB symbols need not be declared in a LCLB instruction. The assembler considers any undeclared variable symbol found in the name field of a SETB instruction as a local SET symbol. The variable symbol is assigned a type attribute value of N .
binary_value
is a binary bit value that may be specified as:
- A binary digit (0 or 1 )
- A binary value enclosed in parentheses

An arithmetic value enclosed in parentheses is allowed. This value can be represented by:
- An unsigned, self-defining term
- A SETA symbol
- A previously defined ordinary symbol with an absolute value
- An attribute reference other than the type attribute reference.

If the value is 0 , the assembler assigns a value of 0 to the symbol in the name field. If the value is not 0 , the assembler assigns a value of 1 .
- A logical expression enclosed in parentheses

A logical expression is evaluated to determine if it is true or false; the SETB symbol in the name field is then assigned the binary value 1 or 0 , corresponding to true or false, respectively. The assembler assigns the explicitly specified binary value ( 0 or 1 ) or the computed logical value ( 0 or 1) to the SETB symbol in the name field.

Rules for Coding Logical Expressions: The following is a summary of coding rules for logical expressions:
- A logical expression must not contain two logical terms in succession.
- A logical expression can contain two logical operators in succession; however, the only allowed combinations are OR NOT, XOR NOT and AND NOT. The two operators must be separated from each other by one or more spaces.
- Any logical term, relation, or inner logical expression can be optionally enclosed in parentheses.
- The relational and logical operators must be immediately preceded and followed by at least one space, except when written (NOT bexpr).
- A logical expression can begin with the logical unary operator NOT.
- A logical expression can contain up to 18 logical operators. The relational and other operators used by the arithmetic and character expressions in relations do not count toward this total.
- Up to 255 levels of nested parentheses are allowed.
- Absolute ordinary symbols specified in logical expressions must be defined before the logical expression is encountered.
- The assembler determines the type of a logical relation by the first comparand. If the first comparand is a character expression that begins with a single quotation mark, then the logical relation is a character relation, otherwise the assembler treats it as an arithmetic relation.

\section*{SETB Instruction}


\section*{Notes:}
1. Outermost expression must be enclosed in parentheses in SETB and AIF instructions.
2. Optional parentheses around terms and expressions at this level.
| 3. Must be in the range 0 through 1024 characters.
4. Must stand alone and not be enclosed in single quotation marks.

Figure 101. Defining Logical Expressions

\section*{Subscripted SETB Symbols}

The SETB symbol in the name field can be subscripted, but only if the same SETB symbol has been previously declared in a GBLB or LCLB instruction with an allowable dimension.

The assembler assigns the binary value explicitly specified, or implicit in the logical expression present in the operand field, to the position in the declared array given by the value of the subscript. The subscript expression must not be 0 or have a negative value.

\section*{Logical (SETB) Expressions}

You can use a logical expression to assign a binary value to a SETB symbol. You can also use a logical expression to represent the condition test in an AIF instruction. This use lets you code a logical expression whose value (0 or 1 ) varies according to the values substituted into the expression and thereby determine whether or not a branch is to be taken.

Figure 101 on page 364 defines a logical expression.
It is important to note that logical expressions contain unquoted spaces that do not terminate the operand field. This is called "logical-expression format," and such expressions are always enclosed in parentheses.

A logical expression can consist of a logical expression and a logical term separated by a logical operator delimited by spaces. The logical operators are:

\section*{AND}

Format: Logical-expression
Operands: Binary
Output: (aexpr1 AND aexpr2) has value 1, if each logical expression evaluates to 1 , otherwise the value is 0 .

\section*{Example}

After the following statements \&VAR contains the arithmetic value 0 .
\begin{tabular}{lll} 
Name & Operation & Operand \\
\&OP1 & SETB & 1 \\
\&OP2 & SETB & 0 \\
\&VAR & SETB & (\&OP1 AND \&OP2)
\end{tabular}

\section*{AND NOT}

Format: Logical-expression
Operands: Binary
Output: The value of the second logical term is inverted, and the expression is evaluated as though the AND operator was specified.

\section*{Example}
( 1 AND NOT 0) is equivalent to (1 AND 1).

\section*{ISBIN}

Format: Function-invocation
Operands: Character
Output: ISBIN('cexpr') determines the validity of cexpr, a string of 1 to 32 characters, as the nominal value of a binary self-defining term usable in a SETA expression. If valid, ISBIN returns 1; otherwise, it returns zero. The argument string may not be null.

\section*{Example}
```

ISBIN('10101') returns 1
ISBIN('101010101010101010101010101010101') returns 0 (excess digits)
ISBIN('12121') returns 0 (non-binary digits)
ISBIN('') indicates an error condition

```

\section*{ISDEC}

Format: Function-invocation
Operands: Character
Output: ISDEC('cexpr') determines the validity of cexpr, a string of 1 to 10 characters, as the nominal value of a decimal self-defining term usable in a SETA expression. If valid, ISDEC returns 1; otherwise, it returns zero. The argument string may not be null.

\section*{Example}
```

ISDEC('12345678') returns 1
ISDEC('+25') returns 0 (non-decimal character)
ISDEC('2147483648') returns 0 (value too large)
ISDEC('00000000005') returns 0 (too many characters)
ISDEC('') indicates an error condition

```

\section*{ISHEX}

Format: Function-invocation
Operands: Character
Output: ISHEX ('cexpr') determines the validity of cexpr, a string of 1 to 8 characters, as the nominal value of a hexadecimal self-defining term usable in a SETA expression. If valid, ISHEX returns 1 ; otherwise, it returns zero. The argument string may not be null.

\section*{Example}
```

ISHEX('ab34CD9F') returns 1
ISHEX('abcdEFGH') returns 0 (non-hexadecimal digits)
ISHEX('123456789') returns 0 (too many characters)
ISHEX('') indicates an error condition

```

ISSYM
Format: Function-invocation
Operands: Character
Output: ISSYM ('cexpr') determines the validity of cexpr, a string of 1 to 63 characters, for use as an ordinary symbol. If valid, ISSYM returns 1; otherwise, it returns zero. The argument string may not be null.

\section*{Examples}
```

ISSYM('Abcd_1234') returns 1
ISSYM('_Abcd1234') returns 1
ISSYM('\#\#@\$_') returns 1
ISSYM('1234_Abcd') returns 0 (invalid initial character)
ISSYM('') - indicates an error condition

```
NOT

Format:
Logical-expression, function-invocation
Operands: Binary
Output: NOT (bexp) inverts the value of the logical expression.
OR
Format: Logical-expression
Operands: Binary

Output: (bexp1 OR bexp2) returns a value of 1, if either of the logical expressions contain or evaluate to 1 . If they both contain or evaluate to 0 then the value is 0 .

\section*{OR NOT}

Format: Logical-expression
Operands: Binary
Output: (bexp1 OR NOT bexp2) inverts the value of the second logical term, and the expression is evaluated as though the OR operator was specified. For example, ( 1 OR NOT 1) is equivalent to ( 1 OR 0).

\section*{XOR}

Format: Logical-expression
Operands: Binary
Output: (bexp1 XOR bexp2) evaluates to 1 if the logical expressions contain or evaluate to opposite bit values. If they both contain or evaluate to the same bit value, the result is 0 .

\section*{XOR NOT}

Format: Logical-expression
Operands: Binary
Output: (bexp1 XOR NOT bexp2) inverts the second logical term, and the expression is evaluated as though the XOR operator was specified.

Example ( 1 XOR NOT 1) is equivalent to ( 1 XOR 0).
Relational Operators: Relational operators provide the means for comparing two items. A relational operator plus the items form a relation. An arithmetic relation is two arithmetic expressions separated by a relational operator, and a character relation is two character strings (for example, a character expression and a type attribute reference) separated by a relational operator.

The relational operators are:
EQ equal
NE not equal
LE less than or equal
LT less than
GE greater than or equal
GT greater than
Evaluation of Logical Expressions: The assembler evaluates logical expressions as follows:
1. It evaluates each logical term, which is given a binary value of 0 or 1 .
2. If the logical term is an arithmetic or character relation, the assembler evaluates:
a. The arithmetic or character expressions specified as values for comparison in these relations
b. The arithmetic or character relation
c. The logical term, which is the result of the relation. If the relation is true, the logical term it represents is given a value of 1 ; if the relation is false, the term is given a value of 0 .

The two comparands in a character relation are compared, character by character, according to binary (EBCDIC) representation of the characters. If two comparands in a relation have character values of unequal length, the assembler always takes the shorter character value to be less.
3. The assembler carries out logical operations from left to right. However,
a. It carries out logical NOTs before logical ANDs, ORs and XORs
b. It carries out logical ANDs before logical ORs and XORs
c. It carries out logical ORs before logical XORs
4. In parenthesized logical expressions, the assembler evaluates the innermost expressions first, and then considers them as logical terms in the next outer level of expressions. It continues this process until it evaluates the outermost expression.

Using SETB Symbols: The logical value assigned to a SETB symbol is used for the SETB symbol appearing in the operand field of an AIF instruction or another SETB instruction.

If a SETB symbol is used in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of AIF and SETB instructions, the binary values 1 (true) and 0 (false) are converted to the arithmetic values 1 and 0 , respectively.

If a SETB symbol is used in the operand field of a SETC instruction, in character relations in the operand fields of AIF and SETB instructions, or in any other statement, the binary values 1 (true) and 0 (false), are converted to the character values ' 1 ' and ' 0 ', respectively.

The following example illustrates these rules. It assumes that (L'\&TO EQ 4) is true, and ( \(S^{\prime} \& T 0\) EQ 0 ) is false.
\begin{tabular}{|c|c|c|c|}
\hline \multirow{5}{*}{\&NAME} & MACRO & & \\
\hline & MOVE & \&TO, \&FROM & \\
\hline & LCLA & \& \({ }^{\text {1 }}\) & \\
\hline & LCLB & \& \(\mathrm{B} 1, \& \mathrm{~B} 2\) & \\
\hline & LCLC & \&C1 & \\
\hline \&B1 & SETB & (L'\&TO EQ 4) & Statement 1 \\
\hline \& \(\mathrm{B}^{2}\) & SETB & (S'\&TO EQ 0) & Statement 2 \\
\hline \& \({ }^{1}\) & SETA & \&B1 & Statement 3 \\
\hline \multirow[t]{6}{*}{\& C1} & SETC & '\&B2' & Statement 4 \\
\hline & ST & 2,SAVEAREA & \\
\hline & L & 2,\&FROM\&A1 & \\
\hline & ST & 2,\&T0\&C1 & \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & FIELDA, FIELDB & \\
\hline +HERE ST & & 2,SAVEAREA & \\
\hline + L & & 2,FIELDB1 & \\
\hline + ST & & 2,FIELDA0 & \\
\hline \(+\mathrm{L}\) & & 2,SAVEAREA & \\
\hline
\end{tabular}

Because the operand field of statement 1 is true, \&B1 is assigned the binary value 1. Therefore, the arithmetic value +1 is substituted for \(\& B 1\) in statement 3 .

Because the operand field of statement 2 is false, \(\& B 2\) is assigned the binary value 0 . Therefore, the character value 0 is substituted for \(\& B 2\) in statement 4.

\section*{SETC Instruction}

The SETC instruction assigns a character value to a SETC symbol. You can assign whole character strings, or concatenate several smaller strings together. The assembler assigns the composite string to your SETC symbol. You can also assign parts of a character string to a SETC symbol by using the substring notation; see "Substring Notation" on page 371.

You can change the character value assigned to a SETC symbol. This lets you use the same SETC symbol with different values for character comparisons in several places, or for substituting different values into the same model statement.
-จ-variable_symbol—SETC—character_value
variable symbol
is a variable symbol.
A global variable symbol in the name field must have been previously declared as a SETC symbol in a GBLC instruction. Local SETC symbols need not be declared in a LCLC instruction. The assembler considers any undeclared variable symbol found in the name field of a SETC instruction as a local SET symbol. The variable symbol is assigned a type attribute value of \(U\).
character_value
is a character value that may be specified by one of the following:
- An operation code attribute reference
- A type attribute reference
- A character expression

The assembler assigns the character string value represented in the operand field to the SETC symbol in the name field. The string length must be in the range 0 (null character string) through 1024 characters.

When a SETA or SETB symbol is specified in a character expression, the unsigned decimal value of the symbol (with leading zeros removed) is the character value given to the symbol.

A duplication factor can precede a character expression or substring notation. The duplication factor can be any non-negative arithmetic expression allowed in the operand of a SETA instruction. For example:
\&C1 SETC (3) 'ABC'
assigns the value ' \(A B C A B C A B C\) ' to \&C1.
A zero duplication factor results in a null (zero-length) string.

\section*{Notes:}
1. The assembler evaluates the represented character string (in particular, the substring, see "Substring Notation" on page 371] before applying the duplication factor. The resulting character string is then assigned to the SETC symbol in the name field. For example:
\&C2 SETC 'ABC'. (3) 'ABCDEF' \((4,3)\)
assigns the value 'ABCDEFDEFDEF' to \&C2.
2. If the character string contains double-byte data, then redundant SI/SO pairs are not removed on duplication. For example:
\&C3
SETC
(3) \({ }^{\prime}<. A . B>1\)
assigns the value '<.A.B><.A.B><.A.B>' to \&C3.
3. To duplicate double-byte data, without including redundant SI/SO pairs, use the substring notation. For example:
\&C4
SETC
(3) \(1<. A . B>1(2,4)\)
assigns the value '.A.B.A.B.A.B' to \&C4.
4. To duplicate the arithmetic value of a previously defined ordinary symbol with an absolute value, first assign the arithmetic value to a SETA symbol. For example:
\begin{tabular}{lll} 
A & EQU & 123 \\
\(\& A 1\) & SETA & \(A\) \\
\(\& C 5\) & SETC & \((3)^{\prime} \& A 1^{\prime}\)
\end{tabular}
assigns the value ' 123123123 ' to \&C5.

\section*{Subscripted SETC Symbols}

The SETC symbol (see 1 in Figure 102 on page 371) in the name field can be subscripted, but only if the same SETC symbol has been previously declared (see 2 in Figure 102) in a GBLC or an LCLC instruction with an allowable dimension.

The assembler assigns the character value represented in the operand field to the position in the declared array (see 3 in Figure 102) given by the value of the subscript. The subscript expression must not be 0 or have a negative value.


Figure 102. Subscripted SETC Symbols

\section*{Character (SETC) Expressions}

The main purpose of a character expression is to assign a character value to a SETC symbol. You can then use the SETC symbol to substitute the character string into a model statement.

You can also use a character expression as a value for comparison in condition tests and logical expressions. Also, a character expression provides the string from which characters can be selected by the substring notation.

Substitution of one or more character values into a character expression lets you use the character expression wherever you need to vary values for substitution or to control loops.

A character string consists of any combination of characters enclosed in single quotation marks. Variable symbols are allowed. The assembler substitutes the representation of their values as character strings into the character expression before evaluating the expression. Up to 1024 characters are allowed in a character expression.

An attribute reference must be the only term in a character expression.
Substring Notation: The substring notation lets you refer to one or more characters within a character string. You can, therefore, either select characters from the string and use them for substitution or testing, or scan through a complete string, inspecting each character. By concatenating substrings with other substrings or character strings, you can rearrange and build your own strings.

The substring notation can be used only in conditional assembly instructions. Figure 103 shows how to use the substring notation.

Figure 103. Substring Notation in Conditional Assembly Instructions
\begin{tabular}{|c|c|c|c|c|c|}
\hline Used in & Used as & \multicolumn{3}{|l|}{Example} & Value assigned to SETC Symbol \\
\hline \multirow[t]{2}{*}{SETC instruction operand} & Operand & & SETC & 'ABC' \((1,3)\) & ABC \\
\hline & Part of operand & \&C2 & SETC & '\&C1' \((1,2)\). 'DEF' & ABDEF \\
\hline AIF or SETB instruction operand (logical expression) & Character value in comparand of character relation & & AIF SETB & \begin{tabular}{l}
('\&STRING' \((1,4)\) EQ 'AREA').SEQ \\
('\&STRING' \((1,4) .{ }^{\prime} 9 '\) EQ 'FULL9')
\end{tabular} & --- \\
\hline
\end{tabular}

The substring notation must be specified as follows:
'CHARACTER STRING' (e1,e2)
where the CHARACTER STRING is a character expression from which the substring is to be extracted. The first subscript (e1) shows the position of the first character that is to be extracted from the character string. The second subscript (e2) shows the number of characters to be extracted from the character string, starting with the character indicated by the first subscript. Thus, the second subscript specifies the length of the resulting substring.

The second subscript value of the substring notation can be specified as an asterisk (*), to indicate that all the characters beginning at the position of the first expression should be used. The extracted string is equal to the length of the character expression, less the number of characters before the starting character.

The character string must be a valid character expression with a length, \(n\), in the range 1 through 1024 characters. The length of the resulting substring must be in the range 0 through 1024.

The subscripts, e1 and e2, must be arithmetic expressions.
When you use subscripted variable symbols in combination with substring notation, take care to distinguish variable subscripts from substring-operation subscripts.
\begin{tabular}{|c|c|c|c|}
\hline & LCLC & \&DVAR(10), \&SVAR, \&C (10) & \\
\hline \&C(1) & SETC & '\&DVAR(5)' & Select 5th element of \&DVAR \\
\hline \&C(2) & SETC & '\&SVAR' \((1,3)\) & Select substring of \&SVAR \\
\hline \&C(3) & SETC & '\&DVAR 5 ) ' \((1,3)\) & Select substring of \&DVAR(5) \\
\hline \&C(4) & SETC & '\&SYSLIST \((1,3)\) ' \((1,3)\) & Select substring of \&SYSLIST \((1,3)\) \\
\hline
\end{tabular}

Evaluation of Substrings: The following examples show how the assembler processes substrings depending on the value of the elements \(n, e 1\), and \(e 2\).
- In the usual case, the assembler generates a correct substring of the specified length:

Notation
' \(\operatorname{ABCDE}\) ( 1,5 )
' \(\operatorname{ABCDE}\) ' \((2,3)\)
'ABCDE' \((2, *)\)
'ABCDE' \((4, *)\)
'\&C' \((3,3)\)
'\&PARAM' \((3,3)\)
\begin{tabular}{ll}
\begin{tabular}{l} 
Value of Variable \\
Symbol
\end{tabular} & \begin{tabular}{l} 
Character Value \\
of Substring
\end{tabular} \\
& ABCDE \\
& BCD \\
& BCDE \\
ABCDE & DE \\
\(((A+3) * 10)\) & \(C D E\) \\
& \(A+3\)
\end{tabular}
- When e1 has a zero or negative value, the assembler generates a null string and issues error message ASMA093E.
\begin{tabular}{ll} 
Notation & \begin{tabular}{l} 
Character Value \\
of Substring
\end{tabular} \\
'ABCDE' \((0,5)\) & null character string \\
'ABCDE' \((0, *)\) & null character string
\end{tabular}
- When the value of e1 exceeds \(n\), the assembler generates a null string and issues error message ASMA092E.
\begin{tabular}{lll} 
Notation & \begin{tabular}{l} 
Value of Variable \\
Symbol
\end{tabular} & \begin{tabular}{l} 
Character Value \\
of Substring
\end{tabular} \\
'ABCDE' \((7,3)\) & & null character string \\
'ABCDE' \((6, *)\) & & null character string
\end{tabular}
- When e2 has a value less than one, the assembler generates the null character string. If e2 is negative, the assembler also issues error message ASMA095W.
\begin{tabular}{ll} 
Notation & \begin{tabular}{l} 
Value of Variable \\
Symbol
\end{tabular} \\
\begin{tabular}{ll} 
Character Value \\
of Substring
\end{tabular} \\
'ABCDE' \((4,0)\) & \\
null character string
\end{tabular}
- When e2 indexes past the end of the character expression (that is, e1+e2 is greater than \(n+1\) ), the assembler issues warning message ASMA094I, and generates a substring that includes only the characters up to the end of the character expression specified.
\begin{tabular}{lll} 
Notation & \begin{tabular}{l} 
Value of Variable \\
Symbol
\end{tabular} & \begin{tabular}{l} 
Character Value \\
of Substring
\end{tabular} \\
'ABCDE' \((3,5)\) & & CDE
\end{tabular}

Figure 104 shows the results of an assembly of SETC instructions with different substring notations.
\begin{tabular}{|c|c|c|c|c|c|}
\hline Loc Object Code & Addr1 Addr2 Stmt & Source State & ment & \multirow[t]{3}{*}{HLASM R5. 0} & 2004/06/11 17.48 \\
\hline & 8 & \&STRING SETC & 'STRING' & & 00008000 \\
\hline & 9 & \&SUBSTR1 SETC & '\&STRING' (0,4) & & 00009000 \\
\hline \multicolumn{6}{|l|}{** ASMA093E Substring expression 1 less than 1; default=null - OPENC} \\
\hline & & \&SUBSTR2 SETC & '\&STRING' \((7,4)\) & & 00010000 \\
\hline \multicolumn{6}{|l|}{** ASMA092E Substring expression 1 points past string end; default=null - OPENC} \\
\hline & & \&SUBSTR3 SETC & '\&STRING' \((3,0)\) & & 00011000 \\
\hline & & \&SUBSTR4 SETC & '\&STRING' \((3,-2)\) & & 00012000 \\
\hline \multicolumn{6}{|l|}{** ASMA095W Substring expression 2 less than 0; default=null - OPENC} \\
\hline & 13 & \&SUBSTR5 SETC & '\&STRING' \((3,4)\) & & 00013000 \\
\hline & & \&SUBSTR6 SETC & '\&STRING' \((3,5)\) & & 00014000 \\
\hline \multicolumn{6}{|l|}{** ASMA094I Substring goes past string end; default=remainder} \\
\hline & 15 & END & & & 00015000 \\
\hline
\end{tabular}

Figure 104. Sample Assembly Using Substring Notation

You can suppress the ASMA094l message by specifying the FLAG(NOSUBSTR) option or by setting the ACONTROL FLAG(NOSUBSTR) value. When this is done, the listing changes Figure 105 on page 374.


Figure 105. Sample Assembly Using Substring Notation With Messages Suppressed
Character (SETC) expressions can be used only in conditional assembly instructions. Figure 106 shows examples of using character expressions.

Figure 106. Use of Character Expressions
\begin{tabular}{lllll}
\hline Used in & Used as & \multicolumn{2}{l}{ Example } & \\
\hline SETC instruction & Operand & \&C & SETC & 'STRING0' \\
\hline AIF or SETB instruction & \begin{tabular}{l} 
Character string \\
in character \\
relation
\end{tabular} & AIF ('\&C' EQ 'STRING1').B & & \\
\hline Substring notation & \begin{tabular}{l} 
First part of \\
notation
\end{tabular} & 'SELECT' (2,5) returns 'ELECT' \\
& Operand & \&VAR & SETC & (LOWER '\&twenty.\&six') \\
\hline Built-in functions & & \&AB & SETA & A2B('10') \\
\hline
\end{tabular}

Character-Valued Built-in Functions: Character-valued built-in functions have arithmetic-only operands, character-only operands, or both arithmetic and character operands. Each type is described in a separate section. The maximum string length of any SETC variable is 1024 bytes. If this length is exceeded, the string value is truncated, and message ASMA091E is generated.

Here are the SETC built-in functions:
A2B
Format: Function-invocation
Operands: Arithmetic
Output: A2B (aexpr) converts the value of its arithmetic argument to a string of 32 zero ('0') and one ('1') characters. The value of aexpr must be representable as a 32-bit binary integer. If the aexpr argument is negative, the result contains 32 characters, the first of which is ' 1 '.

Examples
A2B(0) has value '000000000000000000000000000000000'
A2B(5) has value '000000000000000000000000000000101'
A2B(1022) has value '000000000000000000000001111111110'
A2B (-7) has value '111111111111111111111111111111001'
A2B(2345678901) indicates an error (value too large)
A2C
Format: Function-invocation
Operands: Arithmetic
Output: A2C (aexpr)
converts the value of its arithmetic argument to a string of four characters whose bit pattern is the same as the argument's.
```


## Examples

```
A2C(0) has value 'nnnn' (4 EBCDIC nulls)
A2C(241) has value 'nnn1'
A2C(20046) has value 'nn++'
A2C (-252645136) has value '0000'
```


## A2D

```
Format: Function-invocation
Operands: Arithmetic
Output: A2D (aexpr) converts the value of its arithmetic argument to a string of decimal digits preceded by a plus or minus sign.
Note: The A2D function is similar to the SIGNED function, except that A2D always provides an initial sign character.
```


## Examples

```
A2C(0) has value \({ }^{\prime}+0^{\prime}\)
A2C(241) has value '+241'
A2C(16448) has value ' +16448 '
A2C(-3) has value '-3'
A2X
Format: Function-invocation
Operands: Arithmetic
Output: A2X (aexpr) converts the value of its arithmetic argument to a string of eight hexadecimal characters.
```


## Examples

```
A2X(0) has value '000000000'
A2X(10) has value '00000000A'
A2X(257) has value '000000101'
A2X(1022) has value '000003FE'
A2X (-7) has value 'FFFFFFF9'
B2C
Format: Function-invocation
Operands: Character
Output: B2C('bitstring') converts the bit-string character argument to characters representing the same bit pattern. Null arguments return a null string.
If needed, the argument string is padded internally on the left with zeros so that its length is a multiple of eight.
The operand must contain only ones and zeros. Any other value causes the message ASMA214E to be generated.
Examples
```

B2C('11110011') has value '3'
B2C('101110011110001') has value '*1'
B2C('0') has value ' $n$ ' (EBCDIC null character)
B2C('00010010001') has value 'nj'
B2C('000000000') has value 'nn' (two EBCDIC nulls)
B2C('') has value '' (null string)
B2D

Format: Function-invocation
Operands: Character
Output: B2D('bitstring') converts a bit-string argument of at most 32 ' 0 ' and ' 1 ' characters to one to ten decimal characters preceded by a plus or minus sign, representing the value of the argument. Null arguments return '+0'.

## Examples

B2D('') has value '+0'
B2D('00010010001') has value ' +145 '
B2D('11110001') has value '+241'
B2D('01111111111111111111111111111111') has value '+2147483647'
B2D('11111111111111111111111111110001') has value '-15'
B2X
Format: Function-invocation
Operands: Character
Output: B2X('bitstring') converts the bit-string argument to hexadecimal characters representing the same bit pattern. Null arguments return a null string.

If needed, the argument string is padded internally on the left with zeros so that its length is a multiple of four.
The operand must contain only ones and zeros. Any other value causes the message ASMA214E to be generated.

## Examples

```
B2X('') has value '' (null string)
B2X('00000') has value '00'
B2X('0000010010001') has value '0091'
B2X('11110001') has value 'F1'
B2X('1111110001') has value '3F1'
```


## BYTE

Format: Logical-expression, function-invocation
Operands: Arithmetic
Output: BYTE (aexpr) or (BYTE aexpr) returns a one-character EBCDIC character expression in which the binary value of the character is specified by the arithmetic argument. The argument must have a value between 0 and 255, inclusive.

This function might be used to introduce characters which are not on the keyboard.

## Examples

```
    BYTE(0) has value 'n' (EBCDIC null character)
    BYTE(97) has value '/'
    BYTE(129) has value 'a'

\section*{C2B}
```

Format: Function-invocation
Operands: Character
Output: C 2 B ('charstring') converts the character argument to a string of '0' and ' 1 ' characters representing the same bit pattern. Null arguments return a null string.
If the result is not too long, the length of the result is eight times the length of the 'charstring' argument.
Examples
C2B('') has value ''
C2B('n') has value '000000000'
C2B(' ') has value '010000000'
C2B('1') has value '11110001'
C2B('1234') has value '11110001111100101111001111110100'
C2D
Format: Function-invocation
Operands: Character
Output: C2D('charstring') converts a character-string argument of at most four characters to one to ten decimal characters preceded by a plus or minus sign, representing the numeric value of the argument. Null arguments return '+0'.

```

\section*{Examples}
```

C2D('') has value '+0'
C2D('nj') has value '+145'
C2D('1') has value ' $+241^{\prime}$
C2D('0000') has value '-2526451
C2X
Format: Function-invocation
Operands: Character
Output: C2X('charstring') converts the character-string argument to hexadecimal characters representing the same bit pattern. Null arguments return a null string.
If the result is not too long, the length of the result is two times the length of the 'charstring' argument.

```

\section*{Examples}
```

C2X('') has value ''
C2X('n') has value '00'
C2X('1') has value 'F1'
C2X('a') has value '81'
C2X('1234567R') has value 'F1F2F3F4F5F6F7D9'
D2B
Format: Function-invocation
Operands: Character

```
Output: D2B('decstring') converts an argument string of optionally signed decimal characters to a string of 32 ' 0 ' and ' 1 ' characters representing a bit string with the same binary value. The value of decstring must be representable as a 32-bit binary integer. The argument string may not be null.
```


## Examples

```
\begin{tabular}{ll} 
D2B(' ' \()\) & indicates an error \\
D2B('0') & has value '00000000000000000000000000000000' \\
D2B('+5') & has value '0000000000000000000000000000101' \\
D2B('1022') & has value '0000000000000000000001111111110' \\
D2B('-7') & has value '11111111111111111111111111111001'
\end{tabular}
D2C
Format: Function-invocation Operands: Character
Output: D2C('decstring') converts an argument string of optionally signed decimal characters to a string of four characters whose byte values represent the same binary value. The value of decstring must be representable as a 32 -bit binary integer. The argument string may not be null.
```


## Examples

```
\begin{tabular}{|c|c|}
\hline D2C('') & indicates an error \\
\hline D2C('0') & has value 'nnnn' (4 EBCDIC null bytes) \\
\hline D2C('126') & has value 'nnn=' \\
\hline D2C('247') & has value 'nnn7' \\
\hline D2C('23793') & has value 'nn*1' \\
\hline D2C('-7') & has value 'fffg' ( \(f=\) byte of all 1-bits) \\
\hline
\end{tabular}
D2X
Format: Function-invocation
Operands: Character
Output: D2X('decstring') converts an argument string of optionally signed decimal characters to a string of eight hexadecimal characters whose digits represent the same hexadecimal value. The value of decstring must be representable as a 32 -bit binary integer. The argument string may not be null.
```


## Examples

```
D2X('') indicates an error
```

D2X('') indicates an error
D2X('0') has value '00000000'
D2X('0') has value '00000000'
D2X('+5') has value '00000005'
D2X('+5') has value '00000005'
D2X('255') has value '000000FF'
D2X('255') has value '000000FF'
D2X('01022') has value '000003FE'
D2X('01022') has value '000003FE'
D2X('-7') has value 'FFFFFFF9'
D2X('-7') has value 'FFFFFFF9'
DSX('2345678901') causes an error condition (value too large)

```
DSX('2345678901') causes an error condition (value too large)
```


## DCVAL

```
Format: Function-invocation
Operands: Character
Output: DCVAL('cexpr') performs a single scan of the argument string to find successive pairs of apostrophes and ampersands, and returns a string value in which each such pair has been replaced by a single occurrence. This pairing action occurs only once; that is, three successive occurrences of an apostrophe or ampersand will result in two occurrences, not one. A null argument is returned unchanged.
```

DCVAL is similar to DCLEN, except that DCLEN returns only the length of the result, not the paired string.

## Examples

```
DCVAL('') has value "" (null string)
DCVAL('''') has value "'" (single apostrophe)
DCVAL('&&') has value "&" (single ampersand)
DCVAL('a''''b') has value "a'b"
DCVAL('a''''b&&c') has value "a'b&c"
    .* Suppose &C has value "&&&''" (3 ampersands, 3 apostrophes)
&X SETC DCVAL('&C') &X has value "&&''" (2 of each)
```


## DEQUOTE

Format: Function-invocation
Operands: Character
Output: DEQUOTE ('cexpr') removes a single occurrence of an apostrophe from each end of the argument string, if any are present. A null argument is returned unchanged.

## Examples

| \&C | SETC | DEQUOTE('charstring') | \& ${ }^{\text {c has value "charstring" }}$ |
| :---: | :---: | :---: | :---: |
| \&C | SETC | DEQUOTE('') | $\& C$ is a null string |
| \& C | SETC | DEQUOTE('a') | \&C has value "a" |
| \&ARG | SETC | '''a''' | \&ARG has value "'a'" |
| \&C | SETC | DEQUOTE('\&ARG') | \&C has value "a" |
| \& | SETC | DEQUOTE('a''b') | \&C has value "a'b" |
| \&ARG | SETC | ' 1 | \&ARG has value "'r" |
| \&C | SETC | DEQUOTE('\&ARG') | \& has value "" (null string) |

Format: Logical-expression, function-invocation
Operands: Character
Output: DOUBLE ('cexpr') or (DOUBLE 'cexpr') converts each occurrence of an apostrophe or ampersand character in the argument string to a pair of apostrophes and ampersands. In this form, the string is suitable for substitution into statements such as DC and MNOTE. Null arguments return a null string. An error condition is detected if the resulting string is too long.

## Examples

Suppose the SETC variable \&C contains the characters "\&\&' \&" (two apostrophes, three ampersands):

```
DOUBLE('&C') has value "&&&&''''&&"
```


## LOWER

Format: Logical-expression, function-invocation
Operands: Character
Output: LOWER('cexpr') or (LOWER 'cexpr') converts the alphabetic characters $\mathrm{A}-\mathrm{Z}$ in the argument to lower case, a-z. Null arguments return a null string.

## Examples

LOWER('aBcDefG') has value 'abcdefg'

## SIGNED

Format: Logical-expression, function-invocation Operands:
Arithmetic
Output: SIGNED (aexpr) or (SIGNED aexpr) converts its arithmetic argument to a decimal character string representation of its value, with a leading minus sign if the argument is negative.

## Examples

```
SIGNED(10) has value '10'
SIGNED(-10) has value '-10'
```

Note: The SIGNED function creates properly signed values for display, whereas assigning a SETA value to a SETC variable produces only the magnitude of the SETA value. For example:

| $\& A$ | SETA | 10 | $\& A$ has value 10 |  |
| :--- | :--- | :--- | :--- | :--- |
| $\& C$ | SETC | '\&A' | \&C has value '10' |  |
| \&A | SETA | -10 | \&A has value -10 |  |
| \&C | SETC | '\&A' | \&C has value '10' |  |
| (unsigned) |  |  |  |  |

## SYSATTRA

Format: Function-invocation
Operands: Character
Output: SYSATTRA('symbol') returns the assembler-type value for the specified symbol.

- The 1 to 4 assembler type is returned, with trailing spaces removed.
- Symbols without an assigned assembler type return null.


## Examples

Given that symbol Sym1 has previously been assigned an assembler type of GR, and variable symbol \&SName has a value of SYM1, then:

```
SYSATTRA('Sym1') has value 'GR'
SYSATTRA('&Sname') has value 'GR'
```


## SYSATTRP

Format: Function-invocation
Operands: Character
Output: SYSATTRP('symbol') returns the program-type value for the specified symbol.

- The 4 byte program type is returned.
- Symbols without an assigned program type return null.


## Examples

Given that symbol Sym1 has previously been assigned a program type of "Box7," and variable symbol \&SName has a value of SYM1, then:
SYSATTRP('Sym1') has value 'Box7'
SYSATTRP('\&SName') has value 'Box7'
UPPER
Format: Logical-expression, function-invocation
Operands: Character

Output: UPPER('cexpr') or (UPPER 'cexpr') converts the alphabetic characters $a-z$ in the argument to upper case, A-Z. Null arguments return a null string.
Examples
UPPER('aBcDefG') has value 'ABCDEFG'

## X2B

Format: Function-invocation
Operands: Character
Output: X2B('hexstring') converts the value of its argument string of hexadecimal characters to a character string containing only zero (' 0 ') and one (' 1 ') characters representing the same bit pattern. Null arguments return a null string.

If the result is not too long, the length of the result is four times the length of the 'hexstring' argument.

The operand must contain only hexadecimal digits. Any other value causes the message ASMA214E to be generated.

## Examples

```
X2B('') has value '' (null string)
X2B('00') has value '00000000'
X2B('1') has value '0001'
X2B('F3') has value '11110011'
X2B('00F3') has value '0000000011110011'
```

X2C
Format: Function-invocation
Operands: Character
Output: X2C('hexstring') converts the hexstring argument to characters representing the same bit pattern. Null arguments return a null string.

If needed, the argument string is padded internally on the left with a zero character so that its length is a multiple of two.
The operand must contain only hexadecimal digits. Any other value causes the message ASMA214E to be generated.

## Examples

```
X2C('') has value '' (null string)
X2C('F3') has value '3'
X2C('0') has value 'n' (EBCDIC null character)
X2C('F1F2F3F4F5') has value '12345'
X2C('000F1') has value 'nn1'
```

X2D

Format: Function-invocation
Operands: Character
Output: X2D('hexstring') converts its argument string of at most eight hexadecimal characters to one to ten decimal characters preceded by a plus or minus sign, representing the value of the argument. Null arguments return '+0'. For example:

```
|
```

X2D('') has value '+0'

```
X2D('') has value '+0'
X2D('91') has value '+145'
X2D('91') has value '+145'
X2D('000F1') has value '+241
X2D('000F1') has value '+241
X2D('7FFFFFFF') has value '+2147483647'
X2D('7FFFFFFF') has value '+2147483647'
X2D('FFFFFFF1') has value '-15'
```

X2D('FFFFFFF1') has value '-15'

```


Figure 107. Defining Character (SETC) Expressions

\section*{Notes:}
1. The attribute reference term must not be preceded by a duplication factor.

Evaluation of Character Expressions: The value of a character expression is the character string within the enclosing single quotation marks, after the assembler carries out any substitution for variable symbols.

Character strings, including variable symbols, can be concatenated to each other within a character expression. The resultant string is the value of the expression.

Notes:
1. Use two single quotation marks to generate a single quotation mark as part of the value of a character expression.

The following statement assigns the character value L'SYMBOL to the SETC symbol \&LENGTH.
\&LENGTH SETC 'L'SYMBOL'
2. A double ampersand generates a double ampersand as part of the value of a character expression. To generate a single ampersand in a character expression, use the substring notation; for example:
\&AMP SETC '\&\&' \((1,1)\)
Note: A quoted single ampersand ' \(\&\) ' is not a valid character string.

The following statement assigns the character value HALF\&\& to the SETC symbol \&AND.
```

\&AND SETC 'HALF\&\&'

```

This is the only instance when the assembler does not pair ampersands to produce a single ampersand. However, if you substitute a SETC symbol with such a value into the nominal value in a DC instruction operand, or the operand of an MNOTE instruction, when the assembler processes the DC or MNOTE instruction, it pairs the ampersands and produces a single ampersand.
3. To generate a period, two periods must be specified after a variable symbol.

For example, if \&ALPHA has been assigned the character value \(A B \% 4\), the following statement can be used to assign the character value \(A B \% 4\). RST to the variable symbol \&GAMMA.
\&GAMMA SETC '\&ALPHA..RST'
4. To generate a period, the variable symbol may have a period as part of its value. For example:
```

\&DOT SETC '.'
\&DELTA SETC 'A\&DOT.\&DOT' \&DELTA has value 'A..'

```
5. Double-byte data can appear in the character string if the assembler is invoked with the DBCS option. The double-byte data must be bracketed by the SO and SI delimiters, and the double-byte data must be valid.
6. The DBCS ampersand and apostrophe are not recognized as delimiters.
7. A double-byte character that contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI .
8. Duplication (replication) factors are permitted before character built-in functions.
9. Releases of HLASM prior to Version 1 Release 4 permitted predefined absolute symbols in character expressions. To remove inconsistencies when handling character and arithmetic expressions such usage is no longer permitted and results in message ASMA137S if attempted. The built-in function BYTE can be used to convert a numeric value in a character expression as shown.
\begin{tabular}{llll} 
RPTDS & EQU & X'01' & \\
\&RPTC1 & SETC & 'SEND & '.(BYTE RPTDS)
\end{tabular}

Concatenation of Character String Values: Character expressions can be concatenated to each other or to substring notations in any order. The resulting value is a character string composed of the concatenated parts. This concatenated string can then be used in the operand field of a SETC instruction, or as a value for comparison in a logical expression.

You need the concatenation character (a period) to separate the single quotation mark that ends one character expression from the single quotation mark that begins the next.

For example, either of the following statements may be used to assign the character value ABCDEF to the SETC symbol \&BETA.
```

\&BETA SETC
\&BETA SETC
'ABCDEF'
'ABC'.'DEF'

```

Concatenation of strings containing double-byte data: If the assembler is invoked with the DBCS option, then the following additional considerations apply:
- When a variable symbol adjoins double-byte data, the SO delimiting the double-byte data is not a valid delimiter of the variable symbol. The variable symbol must be terminated by a period.
- The assembler checks for SI and SO at concatenation points. If the byte to the left of the join is SI and the byte to the right of the join is SO , then the \(\mathrm{SI} / \mathrm{SO}\) pair are considered redundant and are removed.
- To create redundant SI/SO pairs at concatenation points, use the substring notation and SETC expressions to create additional SI and SO characters. By controlling the order of concatenation, you can leave a redundant SI/SO pair at a concatenation point.
Instead of substring notation, you can use the BYTE function to create additional SI and SO characters:
```

\&SO SETC (BYTE 14)
\&SI SETC (BYTE 15)

```

\section*{Examples:}
\begin{tabular}{|c|c|c|}
\hline \&DBDA & SETC & '<Da>' \\
\hline \&SO & SETC & BYTE (X'0E') \\
\hline \&SI & SETC & BYTE (X'0F') \\
\hline \& DBCS1A & SETC & '\&DBDA.<Db>' \\
\hline \&DBCS1E & SETC & '\&DBDA<Db>' \\
\hline \&DBCS2 & SETC & '\&DBDA'. \({ }^{\text {< }}\) - \({ }^{\text {db }}\) ' \\
\hline \&DBCS2A & SETC &  \\
\hline \&DBCS3 & SETC & '\&DBDA'.'\&SI'.'\&SO'.'<Db>' \\
\hline \&DBCS3P & SETC & '\&DBDA'.'\&SI' \\
\hline \&DBCS3Q & SETC & '\&SO'. \({ }^{\prime}\) <Db>' \\
\hline \&DBCS3R & SETC & '\&DBCS3P'.'\&DBCS3Q' \\
\hline
\end{tabular}

These examples use the BYTE function to create variables \&SO and \&SI, which have the values of SO and SI, respectively. The variable \&DBCS1A is assigned the value <DaDb> with the SI/SO pair at the join removed. The assignment to variable \&DBCS1E fails with error ASMA035E Invalid delimiter, because the symbol \&DBDA is terminated by SO and not by a period. The variable \&DBCS2 is assigned the value <DaDb>. The variable \&DBCS2A is assigned the value <DaDbDa>. As with \&DBCS1A, redundant \(\mathrm{SI} / \mathrm{SO}\) pairs are removed at the joins. The variable \&DBCS3 is assigned the value <DaDb>. Although SI and SO have been added at the join, the concatenation operation removes two SI and two SO characters, since redundant \(\mathrm{SI} / \mathrm{SO}\) pairs are found at the second and third concatenations. However, by using intermediate variables \&DBCS3P and \&DBCS3Q to change the order of concatenation, the string <Da><Db> can be assigned to variable \&DBCS3R. Note that substituting the variable symbol \&DBCS3R in the nominal value of a G-type constant results in removal of the SI/SO pair at the join.

\section*{Using SETC Symbols}

The character value assigned to a SETC symbol is substituted for the SETC symbol when it is used in the name, operation, or operand field of a statement.

For example, consider the following macro definition, macro instruction, and generated statements:
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & & \\
\hline \multirow[t]{2}{*}{\&NAME} & MOVE & \&TO, \&FROM & \\
\hline & LCLC & \&PREFIX & \\
\hline \multirow[t]{6}{*}{\&PREFIX \&NAME} & SETC & 'FIELD' & Statement 1 \\
\hline & ST & 2,SAVEAREA & \\
\hline & L & 2,\&PREFIX\&FROM & Statement 2 \\
\hline & ST & 2,\&PREFIX\&T0 & Statement 3 \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & A, B & \\
\hline \multicolumn{2}{|l|}{+HERE ST} & 2,SAVEAREA & \\
\hline \(+\mathrm{L}\) & & 2,FIELDB & \\
\hline + ST & & 2,FIELDA & \\
\hline \(+\quad \mathrm{L}\) & & 2,SAVEAREA & \\
\hline
\end{tabular}

Statement 1 assigns the character value FIELD to the SETC symbol \&PREFIX. In statements 2 and 3 , \&PREFIX is replaced by FIELD.

The following example shows how the value assigned to a SETC symbol may be changed in a macro definition.
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & & \\
\hline \multirow[t]{2}{*}{\&NAME} & MOVE & \&TO, \&FROM & \\
\hline & LCLC & \&PREFIX & \\
\hline \&PREFIX & SETC & 'FIELD' & Statement 1 \\
\hline \multirow[t]{2}{*}{\&NAME} & ST & 2,SAVEAREA & \\
\hline & L & 2,\&PREFIX\&FROM & Statement 2 \\
\hline \multirow[t]{4}{*}{\&PREFIX} & SETC & 'AREA' & Statement 3 \\
\hline & ST & 2,\&PREFIX\&T0 & Statement 4 \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & A, B & \\
\hline +HERE ST & & 2,SAVEAREA & \\
\hline \(+\mathrm{L}\) & & 2,FIELDB & \\
\hline + ST & & 2,AREAA & \\
\hline \(+\quad \mathrm{L}\) & & 2,SAVEAREA & \\
\hline
\end{tabular}

Statement 1 assigns the character value FIELD to the SETC symbol \&PREFIX. Therefore, \&PREFIX is replaced by FIELD in statement 2. Statement 3 assigns the character value AREA to \&PREFIX. Therefore, \&PREFIX is replaced by AREA, instead of FIELD, in statement 4.

The following example uses the substring notation in the operand field of a SETC instruction.
\begin{tabular}{|c|c|c|c|}
\hline & MACRO & & \\
\hline \multirow[t]{2}{*}{\&NAME} & MOVE & \&TO, \&FROM & \\
\hline & LCLC & \&PREFIX & \\
\hline \multirow[t]{6}{*}{\&PREFIX \&NAME} & SETC & '\&TO' \((1,5)\) & Statement 1 \\
\hline & ST & 2,SAVEAREA & \\
\hline & L & 2,\&PREFIX\&FROM & Statement 2 \\
\hline & ST & 2,\&T0 & \\
\hline & L & 2,SAVEAREA & \\
\hline & MEND & & \\
\hline HERE & MOVE & FIELDA, B & \\
\hline \multicolumn{2}{|l|}{+HERE ST} & 2, SAVEAREA & \\
\hline + L & & 2,FIELDB & \\
\hline \multirow[t]{2}{*}{\(+\quad \mathrm{ST}\)
\(+\quad \mathrm{L}\)} & & 2,FIELDA & \\
\hline & & 2,SAVEAREA & \\
\hline
\end{tabular}

Statement 1 assigns the substring character value FIELD (the first five characters corresponding to symbolic parameter \&T0 to the SETC symbol \&PREFIX. Therefore, FIELD replaces \&PREFIX in statement 2.

Notes:
1. If the COMPAT(SYSLIST) assembler option is not specified, you can pass a sublist into a macro definition by assigning the sublist to a SETC symbol, and then specifying the SETC symbol as an operand in a macro instruction. However, if the COMPAT(SYSLIST) assembler option is specified, sublists assigned to SETC symbols are treated as a character string, not as a sublist.
2. Regardless of the setting of the COMPAT(SYSLIST) assembler option, you can not pass separate (as opposed to a sublist of) parameters into a macro definition, by specifying a string of values separated by commas as the operand of a SETC instruction and then using the SETC symbol as an operand in the macro instruction. If you attempt to do this, the operand of the SETC instruction is passed to the macro instruction as one parameter, not as a list of parameters.

Concatenating Substring Notations and Character Expressions: Substring notations (see "Substring Notation" on page 371) can be concatenated with character expressions in the operand field of a SETC instruction. If a substring notation follows a character expression, the two can be concatenated by placing a period between the terminating single quotation mark of the character expression and the opening single quotation mark of the substring notation.

For example, if \&ALPHA has been assigned the character value \(A B \% 4\), and \&BETA has been assigned the character value ABCDEF, the following statement assigns \&GAMMA the character value \(A B \% 4 B C D\) :
```

\&GAMMA SETC '\&ALPHA'.'\&BETA'(2,3)

```

If a substring notation precedes a character expression or another substring notation, the two can be concatenated by writing the opening single quotation mark of the second item immediately after the closing parenthesis of the substring notation.

\section*{Extended SET Statements}

Optionally, you can place a period between the closing parenthesis of a substring notation and the opening single quotation mark of the next item in the operand field.

If \&ALPHA has been assigned the character value \(A B \% 4\), and \&ABC has been assigned the character value 5RS, either of the following statements can be used to assign \&WORD the character value AB\%45RS.
```

\&WORD SETC '\&ALPHA' (1,4).'\&ABC'
\&WORD SETC '\&ALPHA' (1,4)'\&ABC' (1,3)

```

If a SETC symbol is used in the operand field of a SETA instruction, the character value assigned to the SETC symbol must be 1-to-10 decimal digits (not greater than 2147483647), or a valid self-defining term.

If a SETA symbol is used in the operand field of a SETC statement, the arithmetic value is converted to an unsigned integer with leading zeros removed. If the value is 0 , it is converted to a single 0 .

\section*{Extended SET Statements}

As well as assigning single values to SET symbols, you can assign values to multiple elements in an array of a subscripted SET symbol with one single SETx instruction. Such an instruction is called an extended SET statement.

variable_symbo((subscript)
is a variable symbol and a subscript that shows the position in the SET symbol array to which the first operand is to be assigned.

\section*{operand}
is the arithmetic value, binary value, or character value to be assigned to the corresponding SET symbol array element.

The first operand is assigned to the SET symbol denoted by variable_symbol(subscript). Successive operands are then assigned to successive positions in the SET symbol array. If an operand is omitted, the corresponding element of the array is unchanged. Consider the following example:
\begin{tabular}{ll} 
LCLA & \&LIST (50) \\
\&LIST(3) SETA & \(5,10,, 20,25,30\)
\end{tabular}

The first instruction declares \&LIST as a subscripted local SETA symbol. The second instruction assigns values to certain elements of the array \&LIST. Thus, the instruction does the same as the following sequence:
\begin{tabular}{ll} 
\&LIST(3) SETA & 5 \\
\&LIST(4) SETA & 10 \\
\&LIST(6) SETA & 20 \\
\&LIST(7) SETA & 25 \\
\&LIST(8) SETA & 30
\end{tabular}

Alternative Statement Format: You can use the alternative statement format for extended SETx statements. The above coding could then be written as follows:
\begin{tabular}{llll}
\(\& L I S T(3) ~ S E T A ~\) & 5, & THIS IS & \(X\) \\
& \(10,\), & AN ARRAY & X \\
& \(20,25,30\) & SPECIFICATION &
\end{tabular}

\section*{SETAF Instruction}

Use the SETAF instruction to call an external function to assign any number of arithmetic values to a SETA symbol. You can assign a large number of parameters-the exact number depending on factors such as the size of the program and of virtual storage-to pass to the external function routine.

The SETAF instruction can be used anywhere that a SETA instruction can be used.

variable symbol
is a variable symbol.
A global variable symbol in the name field must have been previously declared as a SETA symbol in a GBLA instruction. Local SETA symbols need not be declared in a LCLA instruction. The assembler considers any undeclared variable symbol found in the name field of a SETA instruction as a local SET symbol.
The variable symbol is assigned a type attribute value of N .
function_name
the name of an external function load module. The name must be specified as a character expression, and must evaluate to a valid module name no longer than 8 bytes.

Refer to Chapter 5, "Providing External Functions for Conditional Assembly" in the HLASM Programmer's Guide for information about external function load modules.
expression
is an arithmetic expression evaluated as a signed 32-bit arithmetic value. The minimum and maximum allowable values of the expression are \(-2^{31}\) and \(+2^{31}-1\), respectively.

See "SETA Instruction" on page 347 for further information about setting SETA symbols, and ways to specify arithmetic expressions.

The function name must be enclosed in single quotes. For example:
```

\&MAX_VAL SETAF 'MAX',7,4 Calls the external function X
MAX, passing values 7 and X
4 \mp@code { a s ~ o p e r a n d s . }

```

\section*{SETCF Instruction}

Use the SETCF instruction to call an external function to assign a character value to a SETC symbol. You can specify a large number of parameters-the exact number depending on factors such as the size of the program and of virtual storage-to pass to the external function routine.

The SETCF instruction can be used anywhere that a SETC instruction can be used.

variable symbol
is a variable symbol.
A global variable symbol in the name field must have been previously declared as a SETC symbol in a GBLC instruction. Local SETC symbols need not be declared in a LCLC instruction. The assembler considers any undeclared variable symbol found in the name field of a SETC instruction as a local SET symbol. The variable symbol is assigned a type attribute value of \(U\).
The character value assigned to the variable symbol can have a string length in the range 0 (for a null character string) through 1024.
function_name
the name of an external function load module. The name must be specified as a character expression, and must evaluate to a valid module name no longer than 8 bytes.
Refer to Chapter 5, "Providing External Functions for Conditional Assembly" in the HLASM Programmer's Guide for information about external function load modules.
character_value
is a character value that may be specified by one of the following:
- A type attribute reference
- An operation code attribute reference
- A character expression
- A substring notation
- A concatenation of one or more of the above

The character value can have a string length in the range 0 (for a null character string) through 1024.

When a SETA or SETB symbol is specified in a character expression, the unsigned decimal value of the symbol (with leading zeros removed) is the character value given to the symbol.

See "SETC Instruction" on page 369 for further information about setting SETC symbols, and ways to specify character expressions.

\section*{Branching}

You can control the sequence in which source program statements are processed by the assembler by using the conditional assembly branch instructions described in this section.

\section*{AIF Instruction \\ Use the AIF instruction to branch according to the results of a condition test. You can thus alter the sequence in which source program statements or macro definition statements are processed by the assembler. \\ The AIF instruction also provides loop control for conditional assembly processing, which lets you control the sequence of statements to be generated. \\ It also lets you check for error conditions and thereby to branch to the appropriate MNOTE instruction to issue an error message.}

sequence_symbol
is a sequence symbol

\section*{logical_expression}
is a logical expression (see "Logical (SETB) Expressions" on page 365) the assembler evaluates during conditional assembly time to determine if it is true or false. If the expression is true (logical value=1), the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is false (logical value \(=0\) ), the next sequential statement is processed by the assembler.

In the following example, the assembler branches to the label .OUT if \&C \(=\) YES:
\begin{tabular}{lll} 
& AIF & ('\&C' EQ 'YES').OUT \\
.ERROR & ANOP & \\
& \(\cdot\) & \\
& \(\cdot\) & \\
.OUT ANOP & \\
& &
\end{tabular}

The sequence symbol in the operand field is a conditional assembly label that represents a statement number during conditional assembly processing. It is the number of the statement that is branched to if the logical expression preceding the sequence symbol is true.

The statement identified by the sequence symbol referred to in the AIF instruction can appear before or after the AIF instruction. However, the statement must appear within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear:
- In open code, if the corresponding AIF instruction appears in open code
- In the same macro definition in which the corresponding AIF instruction appears.

\section*{AIF Instruction}

You cannot branch from open code into a macro definition or between macro definitions, regardless of nested calls to other macro definitions.

The following macro definition generates the statements needed to move a fullword fixed-point number from one storage area to another. The statements are generated only if the type attribute of both storage areas is the letter F.
\begin{tabular}{llll} 
& MACRO & & \\
\(\& N\) & MOVE & \(\& T, \& F\) & \\
& AIF & \(\left(T^{\prime} \& T\right.\) NE T'\&F).END & Statement 1 \\
& AIF & (T'\&T NE 'F').END & Statement 2 \\
\(\& N\) & ST & 2, SAVEAREA & Statement 3 \\
& L & \(2, \& F\) & \\
& ST & \(2, \& T\) & \\
& L & \(2, S A V E A R E A\) & \\
. END & MEND & & Statement 4
\end{tabular}

The logical expression in the operand field of Statement 1 has the value true if the type attributes of the two macro instruction operands are not equal. If the type attributes are equal, the expression has the logical value false.

Therefore, if the type attributes are not equal, Statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attributes are equal, Statement 2 (the next sequential statement) is processed.

The logical expression in the operand field of Statement 2 has the value true if the type attribute of the first macro instruction operand is not the letter F . If the type attribute is the letter F , the expression has the logical value false.

Therefore, if the type attribute is not the letter F, Statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attribute is the letter F, Statement 3 (the next sequential statement) is processed.

\section*{Extended AIF Instruction}

The extended AIF instruction combines several successive AIF statements into one statement.

sequence_symbol
is a sequence symbol
logical_expression
is a logical expression the assembler evaluates during conditional assembly time to determine if it is true or false. If the expression is true (logical value=1), the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is false (logical value=0), the next logical expression is evaluated.

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

\section*{Example:}
Cont.

AIF

Cont.

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

\section*{Alternative Format for AIF Instruction}

The alternative statement format is allowed for extended AIF instructions. This format is illustrated in the above example.

\section*{AIFB—Synonym of the AIF Instruction}

For compatibility with some earlier assemblers, High Level Assembler supports the AIFB symbolic operation code as a synonym of the AIF instruction. However, you should not use the AIFB instruction in new applications as support for it might be removed in the future.

\section*{AGO Instruction}

The AGO instruction branches unconditionally. You can thus alter the sequence in which your assembler language statements are processed. This provides you with final exits from conditional assembly loops.

sequence_symbol
is a sequence symbol.
The statement named by the sequence symbol in the operand field is the next statement processed by the assembler.

The statement identified by a sequence symbol referred to in the AGO instruction can appear before or after the AGO instruction. However, the statement must appear within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear:
- In open code, if the corresponding AGO instruction appears in open code
- In the same macro definition in which the corresponding AGO instruction appears.
\[
\begin{aligned}
& \text { ('\&L'(\&C,1) EQ '\$').DOLR, } \\
& \text { ('\&L'(\&C,1) EQ '\#').POUND, } \\
& \text { ('\&L'(\&C,1) EQ '@').AT, } \\
& \text { ('\&L'(\&C,1) EQ '=').EQUAL, } \\
& \text { ('\&L'(\&C,1) EQ '(').LEFTPAR, } \\
& \text { ('\&L'(\&C,1) EQ '+').PLUS, } \\
& \text { ('\&L'(\&C,1) EQ '-').MINUS } \\
& \text { X } \\
& \text { X }
\end{aligned}
\]

\section*{Example:}
\begin{tabular}{llll} 
& MACRO & & \\
\&NAME & MOVE & \(\& T, \& F\) & \\
& AIF & \(\left(T^{\prime} \& T\right.\) EQ ' \(\left.F^{\prime}\right)\). FIRST & Statement 1 \\
& AGO & (END & Statement 2 \\
. FIRST & AIF & (T'\&T NE T'\&F).END & Statement 3 \\
\&NAME & ST & 2, SAVEAREA & \\
& L & \(2, \& F\) & \\
& ST & \(2, \& T\) & \\
& L & 2, SAVEAREA & Statement 4
\end{tabular}

Statement 1 determines if the type attribute of the first macro instruction operand is the letter F. If the type attribute is the letter F, Statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, Statement 2 is the next statement processed by the assembler.

Statement 2 indicates to the assembler that the next statement to be processed is Statement 4 (the statement named by sequence symbol .END).

\section*{Computed AGO Instruction}

The computed AGO instruction makes branches according to the value of an arithmetic expression specified in the operand.

sequence_symbol
is a sequence symbol.
arithmetic_expression
is an arithmetic expression the assembler evaluates to \(k\), where \(k\) lies between 1 and \(n\) (the number of occurrences of sequence_symbol in the operand field) inclusive. The assembler branches to the \(k\)-th sequence symbol in the list. If \(k\) is outside that range, no branch is taken.

In the following example, control passes to the statement at .THIRD if \&I= 3 . Control passes through to the statement following the AGO if \&I is less than 1 or greater than 4.
(\&I). FIRST,. SECOND,
Cont.
\(x\) .THIRD, .FOURTH

\section*{Alternative Format for AGO Instruction}

The alternative statement format is allowed for computed AGO instructions. The above example could be coded as follows:
\begin{tabular}{llc} 
& (\&I).FIRST, & Cont. \\
& . SECOND, & \(X\) \\
& .THIRD, & \(X\) \\
&. FOURTH & \(X\)
\end{tabular}

\section*{ACTR Instruction}

\section*{AGOB—Synonym of the AGO Instruction}

For compatibility with some earlier assemblers, High Level Assembler supports the AGOB symbolic operation code as a synonym of the AGO instruction. However, you should not use the AGOB instruction in new applications as support for it might be removed in the future.

\section*{ACTR Instruction}

The ACTR instruction sets a conditional assembly branch counter either within a macro definition or in open code. The ACTR instruction can appear anywhere in open code or within a macro definition.

Each time the assembler processes an AIF or AGO branching instruction in a macro definition or in open code, the branch counter for that part of the program is decremented by one. When the number of conditional assembly branches reaches the value assigned to the branch counter by the ACTR instruction, the assembler exits from the macro definition or stops processing statements in open code.

By using the ACTR instruction, you avoid excessive looping during conditional assembly processing.

sequence_symbol
is a sequence symbol.

\section*{arithmetic_expression}
is an arithmetic expression used to set or reset a conditional assembly branch counter.

A conditional assembly branch counter has a local scope; its value is decremented by AGO and successful AIF instructions, and reassigned only by ACTR instructions that appear within the same scope. Thus, the nesting of macros has no effect on the setting of branch counters in other scopes. The assembler assigns a branch counter for open code and for each macro definition. In the absence of an ACTR instruction, a default value of 4096 is assigned.

\section*{Branch Counter Operations}

Within the scope of a branch counter, the following occurs:
1. Each time an AGO or AIF branch is executed, the assembler checks the branch counter for zero or a negative value.
2. If the count is not zero or negative, it is decremented by one.
3. If the count is zero or negative, the assembler takes one of two actions:
a. If it is processing instructions in open code, the assembler processes the remainder of the instructions in the source module as comments. Errors discovered in these instructions during previous passes are flagged.
b. If it is processing instructions inside a macro definition, the assembler terminates the expansion of that macro definition and processes the next sequential instruction after the calling macro instruction. If the macro definition is called by an outer macro instruction, the assembler processes
the next sequential prototype instruction after the call; that is, it continues processing at the next outer level of nested macros.

The assembler halves the ACTR counter value when it encounters serious syntax errors in conditional assembly instructions.

\section*{ANOP Instruction}

You can specify a sequence symbol in the name field of an ANOP instruction, and use the symbol as a label for branching purposes.

The ANOP instruction carries out no operation itself, but you can use it to allow conditional assembly to resume assembly or conditional generation at an instruction that does not have a sequence symbol in its name field. For example, if you wanted to branch to a SETA, SETB, or SETC assignment instruction, which requires a variable symbol in the name field, you could insert a labeled ANOP instruction immediately before the assignment instruction. By branching to the ANOP instruction with an AIF or AGO instruction, you would, in effect, be branching to the assignment instruction.

sequence_symbol
is a sequence symbol.
No operation is carried out by an ANOP instruction. Instead, if a branch is taken to the ANOP instruction, the assembler processes the next sequential instruction.

\section*{Example:}
\begin{tabular}{llll} 
& MACRO & & \\
\&NAME & MOVE & \&T,\&F & \\
& LCLC & \&TYPE & \\
& AIF & (T'\&T EQ 'F').FTYPE & Statement 1 \\
\&TYPE & SETC & 'E' \(^{\prime}\) & Statement 2 \\
.FTYPE & ANOP & & Statement 3 \\
\&NAME & ST\&TYPE & 2, SAVEAREA & Statement 4 \\
& L\&TYPE & \(2, \& F\) & \\
& ST\&TYPE & \(2, \& T\) & \\
& L\&TYPE & 2, SAVEAREA & \\
& MEND & &
\end{tabular}

Statement 1 determines if the type attribute of the first macro instruction operand is the letter \(F\). If the type attribute is not the letter \(F\), Statement 2 is the next statement processed by the assembler. If the type attribute is the letter \(F\), Statement 4 should be processed next. However, because there is a variable symbol (\&NAME) in the name field of Statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (Statement 3) must be placed before Statement 4.

Then, if the type attribute of the first operand is the letter F , the next statement processed by the assembler is the statement named by sequence symbol. FTYPE. The value of \&TYPE retains its initial null character value because the SETC instruction is not processed. Because .FTYPE names an ANOP instruction, the next
statement processed by the assembler is Statement 4, the statement following the ANOP instruction.

\section*{Chapter 10. MHELP Instruction}

The MHELP instruction controls a set of trace and dump facilities. MHELP statements can occur anywhere in open code or in macro definitions. MHELP options remain in effect until superseded by another MHELP statement.

\section*{MHELP Options}

Options are selected by an absolute expression in the MHELP operand field.

sequence_symbol
is a sequence symbol.
options
is the sum of the binary or decimal options described below.
MHELP B'1' or MHELP 1, Macro Call Trace: This option provides a one-line trace listing for each macro call, giving the name of the called macro, its nested depth, and its \&SYSNDX value. The trace is provided only upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

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

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

MHELP B' \(\mathbf{1 0 0 0}^{\prime}\) or MHELP 8, Macro Exit Dump: This option dumps undimensioned SET symbols from the macro dictionary whenever an MEND or MEXIT statement is encountered.

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

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

MHELP B' 1000000 ' or MHELP 64, Macro Hex Dump: This option, when used with the Macro AIF dump, the Macro Exit dump, or the Macro Entry dump, dumps the parameter and SETC symbol values in EBCDIC and hexadecimal formats. Only positional and keyword parameters are dumped in hexadecimal; system parameters are dumped in EBCDIC. The full value of SETC variables or parameters is dumped in hexadecimal.

\section*{MHELP Instruction}

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

MHELP Control on \&SYSNDX: The maximum value of the \&SYSNDX system variable can be controlled by the MHELP instruction. The limit is set by specifying a number in the operand of the MHELP instruction that is not one of the MHELP codes defined above, and is in the following number ranges:
- 256 to 65535
- Most numbers in the range 65792 to 9999999 . Refer to MHELP Operand Mapping below for details.

When the \&SYSNDX limit is reached, message ASMA013S ACTR counter exceeded is issued, and the assembler in effect ignores all further macro calls.

\section*{MHELP Operand Mapping}

The MHELP operand field is actually mapped into a fullword. The predefined MHELP codes correspond to the fourth byte of this fullword, while the \&SYSNDX limit is controlled by setting any bit in the third byte to 1 . If all bits in the third byte are 0 , then the \&SYSNDX limit is not set.

The bit settings for bytes 3 and 4 are shown in Figure 108.

Figure 108. \&SYSNDX Control Bits
\begin{tabular}{|c|c|}
\hline Byte & Description \\
\hline Byte 3-\&SYSNDX control & \(1 \ldots . \ldots\) Bit \(0=1\). Value \(=32768\). Limit \(\& S Y S N D X\) to .1.. .... Bit \(1=1\). Value \(=16384\). Limit \&SYSNDX to \(\ldots 1 . \ldots\) Bit \(2=1\). Value \(=8192\). Limit \&SYSNDX to 8 \(\ldots 1 \ldots\) Bit \(3=1\). Value \(=4096\). Limit \(\& S Y S N D X\) to 4 \(\ldots . .1 \ldots\) Bit \(4=1\). Value=2048. Limit \&SYSNDX to 20 .... .1.. Bit \(5=1\). Value=1024. Limit \&SYSNDX to 1 \(\ldots . . .1\). Bit \(6=1\). Value \(=512\). Limit \&SYSNDX to 5 \(\ldots . . . .1\) Bit \(7=1\). Value=256. Limit \&SYSNDX to 2 \\
\hline Byte 4 & \begin{tabular}{l}
\(1 \ldots \ldots\) Bit \(0=1\). Value=128. MHELP Suppression. \\
.1.. .... Bit \(1=1\). Value=64. Macro Hex Dump. \\
\(\ldots 1 . \ldots\) Bit 2 = 1. Value=32. Global Suppression. \\
\(\ldots 1 \ldots\). . . Bit \(3=1\). Value=16. Macro Entry Dump. \\
\(\ldots\).... 1... Bit \(4=1\). Value=8. Macro Exit Dump. \\
.... .1.. Bit \(5=1\). Value=4. Macro AIF Dump. \\
\(\ldots . . .1\). Bit \(6=1\). Value=2. Macro Branch Trace. \\
\(\ldots . . .1\) Bit \(7=1\). Value=1. Macro Call Trace.
\end{tabular} \\
\hline
\end{tabular}

Note: You can use any combination of bit settings in any byte of the MHELP fullword to set the limit, provided at least one bit in byte 3 is set. This explains why not all values between 65792 and 9999999 can be used to set the limit. For example, the number 131123 does not set the \&SYSNDX limit because none of the bits in byte 3 are set to 1 .

\section*{Examples:}

MHELP 256 Limit \&SYSNDX to 256
MHELP 1 Trace macro calls
MHELP 65536 No effect. No bits in bytes 3,4
MHELP 65792 Limit \&SYSNDX to 65792

See Figure 109 on page 399 for more examples.

\section*{Combining Options}

More than one MHELP option, including the limit for \&SYSNDX, can be specified at the same time by combining the option codes in one MHELP operand. For example, call and branch traces can be invoked by:
MHELP B'11'
MHELP 2+1
MHELP 3
Substitution by variable symbols may also be used.
\begin{tabular}{|c|c|c|c|c|c|}
\hline MHELP Instruction & \multicolumn{4}{|c|}{MHELP Operand} & \multirow[t]{3}{*}{MHELP Effect} \\
\hline & \multirow[t]{2}{*}{Decimal} & \multicolumn{3}{|c|}{Hexadecimal} & \\
\hline & & & \&SYSNDX & MHELP & \\
\hline MHELP 4869 & 4869 & 0000 & 13 & 05 & Macro call trace and AIF dump; \&SYSNDX limited to 4869 \\
\hline MHELP 65536 & 65536 & 0001 & 00 & 00 & No effect \\
\hline MHELP 16777232 & 16777232 & 0010 & 00 & 10 & Macro entry dump \\
\hline MHELP 28678 & 28678 & 0000 & 70 & 06 & Macro branch trace and AIF dump; \&SYSNDX limited to 28678 \\
\hline MHELP 256+1 & 257 & 0000 & 01 & 01 & \begin{tabular}{l}
Macro call trace; \\
\&SYSNDX limited to 257
\end{tabular} \\
\hline MHELP B'11' & 3 & 0000 & 00 & 03 & Macro call trace, and macro branch trace \\
\hline
\end{tabular}

Figure 109. MHELP Control on \&SYSNDX

\section*{Part 4. Appendixes}
Appendix A. Assembler Instructions ..... 402
Appendix B. Summary of Constants ..... 407
Appendix C. Macro and Conditional Assembly Language Summary ..... 409
Appendix D. Standard Character Set Code Table ..... 421

\section*{Appendix A. Assembler Instructions}

Figure 110 summarizes the basic formats of assembler instructions, and Figure 111 on page 405 summarizes assembler statements.

Figure 110 (Page 1 of 5). Assembler Instructions
\begin{tabular}{|c|c|c|}
\hline Operation Entry & Name Entry & Operand Entry \\
\hline ACONTROL5 & A sequence symbol or space & One or more operands, separated by commas \\
\hline ACTR & A sequence symbol or space & An arithmetic SETA expression \\
\hline ADATA \({ }^{5}\) & A sequence symbol or space & One-to-four decimal, self-defining terms, and one character string, separated by commas. \\
\hline AEJECT \(^{2}\) & A sequence symbol or space & Taken as a remark \\
\hline AGO & A sequence symbol or space & A sequence symbol \\
\hline AIF & A sequence symbol or space & A logical expression enclosed in parentheses, immediately followed by a sequence symbol \\
\hline AINSERT \({ }^{5}\) & A sequence symbol or space & A character string, followed by FRONT or BACK \\
\hline AMODE & Any symbol or space & 24, 31, 64, ANY or ANY31 \\
\hline ALIAS \({ }^{5}\) & A symbol & A character string or a hexadecimal string \\
\hline ANOP & A sequence symbol or space & Taken as a remark \\
\hline AREAD \({ }^{2}\) & Any SETC symbol & NOPRINT, NOSTMT, CLOCKB, CLOCKD, or spaces \\
\hline ASPACE & A sequence symbol or space & An absolute expression \\
\hline CATTR (MVS and CMS) & A valid program object external class name & One or more attributes \\
\hline \(\mathrm{CCW}^{4}\) & Any symbol or space & Four operands, separated by commas \\
\hline CCW0 \({ }^{4}\) & Any symbol or space & Four operands, separated by commas \\
\hline CCW14 & Any symbol or space & Four operands, separated by commas \\
\hline CEJECT \({ }^{5}\) & A sequence symbol or space & An absolute expression or space \\
\hline CNOP4 & Any symbol or space & Two absolute expressions, separated by a comma \\
\hline COM & Any symbol or space & Taken as a remark \\
\hline COPY \({ }^{5}\) & A sequence symbol or space & An ordinary symbol, or, for open code statements, a variable symbol \\
\hline CSECT & Any symbol or space & Taken as a remark \\
\hline CXD \({ }^{4}\) & Any symbol or space & Taken as a remark \\
\hline DC4 & Any symbol or space & One or more operands, separated by commas \\
\hline DROP & A sequence symbol or space & One or more absolute expressions and symbols, separated by commas, or space \\
\hline
\end{tabular}

Figure 110 (Page 2 of 5). Assembler Instructions
Operation
\begin{tabular}{|c|c|c|}
\hline Entry & Name Entry & Operand Entry \\
\hline DS \({ }^{4}\) & Any symbol or space & One or more operands, separated by commas \\
\hline DSECT & A symbol or space & Taken as a remark \\
\hline DXD4,5 & A symbol & One or more operands, separated by commas \\
\hline EJECT \({ }^{5}\) & A sequence symbol or space & Taken as a remark \\
\hline END & A sequence symbol or space & A relocatable expression or space \\
\hline ENTRY5 & A sequence symbol or space & One or more relocatable symbols, separated by commas \\
\hline EQU \({ }^{4}\) & A variable symbol or an ordinary symbol & One to five operands, separated by commas \\
\hline EXITCTL \({ }^{5}\) & A sequence symbol or space & A character-string operand followed by one to four decimal self-defining terms, separated by commas \\
\hline EXTRN \({ }^{5}\) & A sequence symbol or space & One or more relocatable symbols, separated by commas \\
\hline GBLA & A sequence symbol or space & One or more variable symbols that are to be used as SET symbols, separated by commas \({ }^{1}\) \\
\hline GBLB & A sequence symbol or space & One or more variable symbols that are to be used as SET symbols, separated by commas \({ }^{1}\) \\
\hline GBLC & A sequence symbol or space & One or more variable symbols that are to be used as SET symbols, separated by commas \({ }^{1}\) \\
\hline ICTL & Space & One to three decimal self-defining terms, separated by commas \\
\hline ISEQ \({ }^{5}\) & A sequence symbol or space & Two decimal self-defining terms, separated by a comma, or space \\
\hline LCLA & A sequence symbol or space & One or more variable symbols that are to be used as SET symbols, separated by commas \({ }^{1}\) \\
\hline LCLB & A sequence symbol or space & One or more variable symbols that are to be used as SET symbols, separated by commas \({ }^{1}\) \\
\hline LCLC & A sequence symbol or space & One or more variable symbols separated by commas \({ }^{1}\) \\
\hline LOCTR & A variable or ordinary symbol & Space \\
\hline LTORG & Any symbol or space & Taken as a remark \\
\hline MACRO2,5 & Space & Taken as a remark \\
\hline MEND2,5 & A sequence symbol or space & Taken as a remark \\
\hline MEXIT 2,5 & A sequence symbol or space & Taken as a remark \\
\hline MHELP & A sequence symbol or space & Absolute expression, binary or decimal options \\
\hline
\end{tabular}

Figure 110 (Page 3 of 5). Assembler Instructions
Operation
\begin{tabular}{lll} 
Entry & Name Entry & Operand Entry \\
\hline MNOTE & A sequence symbol or space & \begin{tabular}{l} 
A severity code, followed by a comma, \\
followed by a 1-to-256-character string \\
enclosed in single quotation marks. \\
Double-byte characters are permitted if the \\
DBCS assembler option is specified.
\end{tabular} \\
\hline OPSYN & An ordinary symbol & \begin{tabular}{l} 
An machine instruction mnemonic \\
or an operation code defined by \\
a previous macro definition or \\
OPSYN instruction
\end{tabular} \\
& An operation code mnemonic & Space \\
\hline ORG & A sequence symbol or space & A relocatable expression or space \\
\hline POP5 & A sequence symbol or space & \begin{tabular}{l} 
One or more operands, separated by \\
commas
\end{tabular} \\
\hline PRINT5 & A sequence symbol or space & \begin{tabular}{l} 
One or more operands, separated by \\
commas
\end{tabular} \\
\hline PUNCH5 & A sequence symbol or space & \begin{tabular}{l} 
A 1-to-80-character string enclosed in \\
single quotation marks. Double-byte \\
characters are permitted if the DBCS \\
assembler option is specified.
\end{tabular} \\
\hline
\end{tabular}
\begin{tabular}{lll}
\hline PUSH \(^{5}\) & A sequence symbol or space & \begin{tabular}{l} 
One or more operands, separated by \\
commas
\end{tabular} \\
\hline REPRO \(^{5}\) & A sequence symbol or space & Taken as a remark \\
\hline RMODE & Any symbol or space & \(24,31,64\) or ANY \\
\hline RSECT & Any symbol or space & Taken as a remark \\
\hline SETA & A SETA symbol & An arithmetic expression \\
\hline SETAF & A SETA symbol & \begin{tabular}{l} 
An external function module, and the \\
arithmetic expressions it requires, \\
separated by commas
\end{tabular} \\
\hline SETB & A SETB symbol & \begin{tabular}{l} 
A 0 or a 1, or a logical expression \\
enclosed in parentheses
\end{tabular} \\
\hline SETC & A SETC symbol & \begin{tabular}{l} 
A type attribute, a character expression, a \\
substring notation, or a concatenation of \\
character expressions and substring
\end{tabular} \\
& & \begin{tabular}{l} 
notations. Double-byte characters are \\
permitted if the DBCS assembler option is \\
specified.
\end{tabular} \\
& & \begin{tabular}{l} 
An exter
\end{tabular} \\
& &
\end{tabular}
\begin{tabular}{lll}
\hline SETCF & A SETC symbol & \begin{tabular}{l} 
An external function module, and the \\
character expressions it requires, \\
separated by commas
\end{tabular} \\
\hline SPACE \(^{5}\) & A sequence symbol or space & An absolute expression \\
\hline START & Any symbol or space & An absolute expression or space \\
\hline TITLE3,5 & \begin{tabular}{l} 
A 1-to-8-character string, a \\
variable symbol, a \\
combination of character \\
string or variable symbol, a \\
sequence symbol, or space
\end{tabular} & \begin{tabular}{l} 
A 1-to-100-character string enclosed in \\
single quotation marks. Double-byte \\
characters are permitted if the DBCS \\
assembler option is specified.
\end{tabular} \\
\hline
\end{tabular}

Figure 110 (Page 4 of 5). Assembler Instructions
\begin{tabular}{lll}
\begin{tabular}{l} 
Operation \\
Entry
\end{tabular} & Name Entry & Operand Entry \\
\hline USING & A symbol or space & \begin{tabular}{l} 
Either a single absolute or relocatable \\
expression or a pair of absolute or \\
relocatable expressions enclosed in \\
parentheses and followed by 1 to 16 \\
absolute expressions, separated by \\
commas, or followed by a relocatable \\
expression
\end{tabular} \\
\hline WXTRN \(^{5}\) & A sequence symbol or space & \begin{tabular}{l} 
One or more relocatable symbols, \\
separated by commas
\end{tabular} \\
\hline XATTR5 & An external symbol & \begin{tabular}{l} 
One or more operands, separated by \\
commas
\end{tabular} \\
(MVS and & & \\
\hline
\end{tabular}

\section*{Notes:}
1. SET symbols may be defined as subscripted SET symbols.
2. May only be used as part of a macro definition.
3. See "TITLE Instruction" on page 215 for a description of the name entry.
4. These instructions start a private section.
5. These instructions can be specified before the first executable control section.

Figure 111 (Page 1 of 2). Assembler Statements
\begin{tabular}{|c|c|c|}
\hline Instruction Entry & Name Entry & Operand Entry \\
\hline Model Statements 1 and 2 & An ordinary symbol, variable symbol, sequence symbol, or a combination of variable symbols and other characters that is equivalent to a symbol, or space & Any combination of characters (including variable symbols) \\
\hline Prototype Statement \({ }^{3}\) & A symbolic parameter or space & Zero or more operands that are symbolic parameters (separated by commas), and zero or more operands (separated by commas) of the form symbolic parameter, equal sign, optional standard value \\
\hline Macro Instruction Statement \({ }^{3}\) & An ordinary symbol, a variable symbol, or a combination of variable symbols and other characters that is equivalent to a symbol, any character string, a sequence symbol \({ }^{4}\) or space & Zero or more positional operands (separated by commas), and zero or more keyword operands (separated by commas) of the form keyword, equal sign, value \({ }^{5}\) \\
\hline
\end{tabular}

\section*{Assembler Instructions and Statements}

Figure 111 (Page 2 of 2). Assembler Statements
\begin{tabular}{lll}
\hline Instruction Entry & Name Entry & Operand Entry \\
\hline Assembler & \begin{tabular}{l} 
An ordinary symbol, a variable \\
symbol, a sequence symbol, or
\end{tabular} & \begin{tabular}{l} 
Any combination of characters \\
(including variable symbols)
\end{tabular} \\
Stanguage & \begin{tabular}{l} 
a combination of variable \\
symbols and other characters \\
that is equivalent to a symbol, \\
or space
\end{tabular} & \\
\hline
\end{tabular}

\section*{Notes:}
1. Variable symbols may be used to generate assembler language mnemonic operation codes (listed in Chapter 5, "Assembler Instruction Statements" on page 100), except COPY, ICTL, ISEQ, and REPRO. Variable symbols may not be used in the name and operand entries of COPY, ICTL, and ISEQ instructions, except for the COPY instruction in open code, where a variable symbol is allowed for the operand entry.
2. No substitution is done for variables in the line following a REPRO statement.
3. May only be used as part of a macro definition.
4. When the name field of a macro instruction contains a sequence symbol, the sequence symbol is not passed as a name field parameter. It only has meaning as a possible branch target for conditional assembly.
5. Variable symbols appearing in a macro instruction are replaced by their values before the macro instruction is processed.

\section*{Appendix B. Summary of Constants}

Figure 112 and Figure 113 on page 408 summarize the types of assembler constants.

Figure 112. Summary of Constants (Part 1 of 2)
\begin{tabular}{|c|c|c|c|c|c|}
\hline Constant & Type & Implicit Length (Bytes) & Alignment & Length Modifier Range & Specified By \\
\hline Address & A & 4 & Fullword & . 1 to \(4^{1}\) & Any expression \\
\hline Doubleword Address & AD & 8 & Doubleword & 2 to 8 & Any expression \\
\hline Binary & B & As needed & Byte & . 1 to 256 & Binary digits \\
\hline Character & C & As needed & Byte & . 1 to \(256^{2}\) & Characters \\
\hline ASCII Character & CA & As needed & Byte & . 1 to \(256{ }^{2}\) & Characters \\
\hline Unicode Character & CU & As needed & Byte & 2 to \(256^{3}\) & Characters \\
\hline Floating Point Hex & D & 8 & Doubleword & . 1 to 8 & Decimal digits \\
\hline Floating Point Hex & DH & 8 & Doubleword & . 12 to 8 & Decimal digits \\
\hline Floating Point Binary & DB & 8 & Doubleword & . 12 to 8 & Decimal digits \\
\hline Floating Point Hex & E & 4 & Fullword & . 1 to 8 & Decimal digits \\
\hline Floating Point Hex & EH & 4 & Fullword & . 12 to 8 & Decimal digits \\
\hline Floating Point Binary & EB & 4 & Fullword & . 9 to 8 & Decimal digits \\
\hline Fixed Point & F & 4 & Fullword & . 1 to 8 & Decimal digits \\
\hline Doubleword Fixed Point & FD & 8 & Doubleword & . 1 to 8 & Decimal digits \\
\hline Graphic (DBCS) & G & As needed & Byte & 2 to \(256^{3}\) & DBCS characters \\
\hline Fixed Point & H & 2 & Halfword & . 1 to 8 & Decimal digits \\
\hline Length & J & 4 & Fullword & 1 to 4 & Class name or external DSECT name \({ }^{4}\) \\
\hline Floating Point Hex & L & 16 & Doubleword & . 1 to 16 & Decimal digits \\
\hline Floating Point Hex & LH & 16 & Doubleword & . 12 to 16 & Decimal digits \\
\hline Floating Point Binary & LB & 16 & Doubleword & . 16 to 16 & Decimal digits \\
\hline Floating Point Hex & LQ & 16 & Quadword & . 1 to 16 & Decimal digits \\
\hline Decimal & P & As needed & Byte & . 1 to 16 & Decimal digits \\
\hline Offset & Q & 4 & Fullword & 1 to 4 & Symbol naming a DXD or DSECT \\
\hline Address & \(\mathrm{R}^{4}\) & 4 & Fullword & 3, 4 & Symbol \\
\hline Address & S & 2 & Halfword & 2 only & One absolute or relocatable expression, or two absolute expressions: \(\exp (\exp )\) \\
\hline Address & V & 4 & Fullword & 3, 4 & Relocatable symbol \\
\hline Hexadecimal & X & As needed & Byte & . 1 to \(256^{2}\) & Hex digits \\
\hline Address & Y & 2 & Halfword & . 1 to \(2^{1}\) & Any expression \\
\hline Decimal & Z & As needed & Byte & . 1 to 16 & Decimal digits \\
\hline
\end{tabular}

\section*{Notes:}
1. Bit length specification permitted with absolute expressions only; relocatable A-type constants, 2 , 3 , or 4 bytes only; relocatable Y-type constants, 2 bytes only.
2. In a DS assembler instruction, C-and-X type constants can have length specification to 65535.
3. The length modifier must be a multiple of 2 , and may be up to 65534 in a DS assembler instruction.
4. GOFF only.

\section*{Summary of Constants}

Figure 113. Summary of Constants (Part 2 of 2)
\begin{tabular}{|c|c|c|c|c|c|}
\hline Constant & Type & No. of Constants per Operand & Range for Exponents & Range for Scale & Truncation or Padding Side \\
\hline Address & A & Multiple & & & Left \\
\hline Binary & B & Multiple & & & Left \\
\hline Character & C & One & & & Right \\
\hline ASCII Character & CA & One & & & Right \\
\hline Unicode Character & CU & One & & & Right \\
\hline Floating Point Hex & D & Multiple & -85 to +75 & 0 to 14 & Right \({ }^{1}\) \\
\hline Floating Point Hex & DH & Multiple & \(-2^{31}\) to \(2^{31}-1\) & 0 to 14 & Right \({ }^{1}\) \\
\hline Floating Point Binary & DB & Multiple & \(-2^{31}\) to \(2^{31}-1\) & N/A & Right \({ }^{1}\) \\
\hline Floating Point Hex & E & Multiple & -85 to +75 & 0 to 14 & Right \({ }^{1}\) \\
\hline Floating Point Hex & EH & Multiple & \(-2^{31}\) to \(2^{31}-1\) & 0 to 14 & Right \({ }^{1}\) \\
\hline Floating Point Binary & EB & Multiple & \(-2^{31}\) to \(2^{31}-1\) & N/A & Right \({ }^{1}\) \\
\hline Fixed Point & F & Multiple & -85 to +75 & -187 to +346 & Left \({ }^{1}\) \\
\hline Graphic (DBCS) & G & One & & & Right \\
\hline Fixed Point & H & Multiple & -85 to +15 & -187 to +346 & Left \({ }^{1}\) \\
\hline Length & J & Multiple & & & Left \({ }^{1}\) \\
\hline Floating Point Hex & L & Multiple & -85 to +75 & 0 to 28 & Right \({ }^{1}\) \\
\hline Floating Point Hex & LH & Multiple & \(-2^{31}\) to \(2^{31}-1\) & 0 to 28 & Right \({ }^{1}\) \\
\hline Floating Point Binary & LB & Multiple & \(-2^{31}\) to \(2^{31}-1\) & N/A & Right \({ }^{1}\) \\
\hline Floating Point Hex & LQ & Multiple & -231 to \(2^{31}-1\) & 0 to 28 & Right \({ }^{1}\) \\
\hline Decimal & \(P\) & Multiple & & & Left \\
\hline Offset & Q & Multiple & & & Left \\
\hline Address & R & Multiple & & & Left \\
\hline Address & S & Multiple & & & \\
\hline Address & V & Multiple & & & Left \\
\hline Hexadecimal & X & Multiple & & & Left \\
\hline Address & Y & Multiple & & & Left \\
\hline Decimal & Z & Multiple & & & Left \\
\hline
\end{tabular}

\section*{Notes:}
1. Errors are flagged if significant bits are truncated or if the value specified cannot be contained in the implicit length of the constant.

\section*{Appendix C. Macro and Conditional Assembly Language Summary}

This appendix summarizes the macro and conditional assembly language described in Part 3 of this publication. Figure 114 indicates which macro and conditional assembly language elements may be used in the name and operand entries of each statement. Figure 116 on page 413 summarizes the expressions that may be used in macro instruction statements Figure 118 on page 414 summarizes the attributes that may be used in each expression Figure 119 on page 415 summarizes the variable symbols that may be used in each expression.
Figure 120 on page 416 summarizes the system variable symbols that may be used in each expression.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{9}{|l|}{Figure 114 (Page 1 of 2). Macro Language Elements (Part 1)} \\
\hline \multirow[t]{3}{*}{Statement} & \multirow[t]{3}{*}{Symbolic Parameter} & \multicolumn{6}{|l|}{Variable Symbols} & \multirow[t]{3}{*}{Sequence Symbol} \\
\hline & & \multicolumn{3}{|l|}{Global-scope SET Symbols \({ }^{2}\)} & \multicolumn{3}{|l|}{Local SET Symbols \({ }^{2}\)} & \\
\hline & & SETA & SETB & SETC & SETA & SETB & SETC & \\
\hline \multicolumn{9}{|l|}{MACRO} \\
\hline Prototype Statement & Name Operand & & & & & & & \\
\hline GBLA & Operand \({ }^{11}\) & Operand & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Name \\
\hline GBLB & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Name \\
\hline GBLC & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Name \\
\hline LCLA & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Name \\
\hline LCLB & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand & Operand \({ }^{11}\) & Name \\
\hline LCLC & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand \({ }^{11}\) & Operand & Name \\
\hline Model Statement & Name Operation Operand & Name Operation Operand & Name Operation Operand & Name Operation Operand & Name Operation Operand & \begin{tabular}{l}
Name \\
Operation \\
Operand
\end{tabular} & Name Operation Operand & Name \\
\hline SETA & Name \({ }^{12}\) Operand \({ }^{3}\) & Name Operand & Name \({ }^{12}\) Operand \({ }^{4}\) & Name \({ }^{12}\) Operand \({ }^{10}\) & Name Operand & Name \({ }^{12}\) Operand \({ }^{4}\) & Name \({ }^{12}\) Operand \({ }^{10}\) & \\
\hline SETAF & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{3},{ }^{13}\)
\end{tabular} & \begin{tabular}{l}
Name \\
Operand \({ }^{13}\)
\end{tabular} & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{4},^{13}\)
\end{tabular} & Name \({ }^{10,12}\) Operand \({ }^{13}\) & Name Operand \({ }^{13}\) & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{4},^{13}\)
\end{tabular} & Name \({ }^{10,12}\) Operand \({ }^{13}\) & \\
\hline SETB & Name \({ }^{12}\) Operand \({ }^{7}\) & Name \({ }^{12}\) Operand \({ }^{7}\) & Name Operand & Name \({ }^{12}\) Operand \({ }^{7}\) & Name \({ }^{12}\) Operand \({ }^{7}\) & Name Operand & Name \({ }^{12}\) Operand \({ }^{7}\) & \\
\hline SETC & Name \({ }^{12}\) Operand & Name \({ }^{12}\) Operand \({ }^{8}\) & Name \({ }^{12}\) Operand \({ }^{9}\) & Name Operand & Name \({ }^{12}\) Operand \({ }^{8}\) & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{9}\)
\end{tabular} & Name Operand & Operand \\
\hline SETCF & Name \({ }^{12}\) Operand \({ }^{13}\) & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{8},{ }^{13}\)
\end{tabular} & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{9},^{13}\)
\end{tabular} & Name Operand \({ }^{13}\) & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{8},{ }^{13}\)
\end{tabular} & \begin{tabular}{l}
Name \({ }^{12}\) \\
Operand \({ }^{9},^{13}\)
\end{tabular} & Name Operand \({ }^{13}\) & \\
\hline ACTR & Operand \({ }^{3}\) & Operand & Operand \({ }^{4}\) & Operand \({ }^{3}\) & Operand & Operand \({ }^{4}\) & Operand \({ }^{3}\) & Name \\
\hline AEJECT & & & & & & & & Name \\
\hline AGO & & & & & & & & Name Operand \\
\hline AIF & Operand \({ }^{7}\) & Operand \({ }^{7}\) & Operand & Operand \({ }^{7}\) & Operand \({ }^{7}\) & Operand & Operand \({ }^{7}\) & Name Operand \\
\hline ANOP & & & & & & & & Name \\
\hline AREAD & Name \({ }^{12}\) & Name \({ }^{12}\) & Name \({ }^{12}\) & Name & Name \({ }^{12}\) & Name \({ }^{12}\) & Name & \\
\hline ASPACE & Operand \({ }^{3}\) & Operand & Operand \({ }^{4}\) & Operand \({ }^{3}\) & Operand & Operand \({ }^{4}\) & Operand \({ }^{3}\) & Name \\
\hline MEXIT & & & & & & & & Name \\
\hline MNOTE & Operand & Operand & Operand & Operand & Operand & Operand & Operand & Name \\
\hline MEND & & & & & & & & Name \\
\hline Outer Macro & & Name Operand & Name Operand & Name Operand & Name Operand & Name Operand & Name Operand & Name \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{9}{|l|}{Figure 115. Macro Language Elements (Part 2)} \\
\hline \multirow[t]{2}{*}{Statement} & \multicolumn{7}{|l|}{Attributes} & \multirow[t]{2}{*}{Type} \\
\hline & Length & Scale & Integer & Count & Number & Defined & Operation Code & \\
\hline \multicolumn{9}{|l|}{MACRO} \\
\hline \multicolumn{9}{|l|}{Prototype Statement} \\
\hline \multicolumn{9}{|l|}{GBLA} \\
\hline \multicolumn{9}{|l|}{GBLB} \\
\hline \multicolumn{9}{|l|}{GBLC} \\
\hline \multicolumn{9}{|l|}{LCLA} \\
\hline \multicolumn{9}{|l|}{LCLB} \\
\hline \multicolumn{9}{|l|}{LCLC} \\
\hline \multicolumn{9}{|l|}{Model Statement} \\
\hline SETA & & Operand & Operand & Operand & Operand & Operand & Operand & \\
\hline SETAF & & Operand \({ }^{13}\) & Operand \({ }^{13}\) & Operand \({ }^{13}\) & Operand \({ }^{13}\) & Operand \({ }^{13}\) & & \\
\hline SETB & Operand \({ }^{5}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{5}\) \\
\hline SETC & Operand & & & & & & & Operand \\
\hline SETCF & Operand \({ }^{13}\) & & & & & & & \\
\hline ACTR & & Operand & Operand & Operand & Operand & Operand & & \\
\hline \multicolumn{9}{|l|}{AEJECT} \\
\hline \multicolumn{9}{|l|}{AGO} \\
\hline AIF & Operand \({ }^{5}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand \({ }^{6}\) & Operand & Operand \\
\hline \multicolumn{9}{|l|}{ANOP} \\
\hline \multicolumn{9}{|l|}{AREAD} \\
\hline ASPACE & & Operand & Operand & Operand & Operand & Operand & & \\
\hline \multicolumn{9}{|l|}{MEXIT} \\
\hline \multicolumn{9}{|l|}{MNOTE} \\
\hline \multicolumn{9}{|l|}{MEND} \\
\hline \multicolumn{9}{|l|}{\begin{tabular}{l}
Outer \\
Macro
\end{tabular}} \\
\hline \multicolumn{9}{|l|}{\begin{tabular}{l}
1. Variable symbols in macro instructions are replaced by their values before processing. \\
 \\
3. Only if value is self-defining term. \\
4. Converted to arithmetic 0 or 1. \\
5. Only in character relations. \\
6. Only in arithmetic relations. \\
7. Only in arithmetic or character relations. \\
8. Converted to an unsigned number. \\
9. Converted to character 0 or 1. \\
10. Only if one to ten decimal digits, not greater than 2147483647. \\
11. Only in created SET symbols if value of parenthesized expression is an alphabetic character followed by 0 to 61 alphanumeric characters. \\
12. Only in created SET symbols (as described above) and in subscripts (see SETA statement). \\
13. The first operand of a SETAF or SETCF instruction must be a character (SETC) expression containing or evaluating to an eight byte module name.
\end{tabular}} \\
\hline
\end{tabular}

Figure 116 (Page 1 of 2). Conditional Assembly Expressions
\begin{tabular}{|c|c|c|c|}
\hline Expression & Arithmetic Expressions & Character Expressions & Logical Expressions \\
\hline \multirow[t]{7}{*}{Can contain} & Self-defining terms & Any combination of characters (including double-byte & A 0 or a 1 \\
\hline & Absolute, predefined ordinary symbols & characters, if the DBCS assembler option is specified) enclosed in single quotation & Absolute, predefined ordinary symbols \\
\hline & Length, scale, integer, count, & marks & SETB symbols \\
\hline & defined, and number attributes & Any variable symbol enclosed in single quotation marks & Arithmetic relations \\
\hline & SETA and SETB symbols & A concatenation of variable symbols and other characters enclosed in single quotation & \begin{tabular}{l}
Character relations \\
Arithmetic value
\end{tabular} \\
\hline & SETC symbols whose values are a self-defining term & \begin{tabular}{l}
marks \\
Built-in Functions
\end{tabular} & \\
\hline & Symbolic parameters if the corresponding operand is a decimal self-defining term & \begin{tabular}{l}
A type or operation code attribute reference \\
Substrings
\end{tabular} & \\
\hline
\end{tabular}
\&SYSDATC
\&SYSLIST( \(n\) ) if the corresponding operand is a decimal self-defining term
\&SYSLIST \((n, m)\) if the corresponding operand is a decimal self-defining term
\&SYSOPT_DBCS,
\&SYSOPT_RENT,
and
\&SYSOPT_XOBJECT
\&SYSM_HSEV and \&SYSM_SEV
\&SYSNDX,
\&SYSNEST, and
\&SYSSTMT
\begin{tabular}{|c|c|c|c|}
\hline Operations & \begin{tabular}{l}
+, - (unary and binary), *, and /; \\
Parentheses permitted
\end{tabular} & Concatenation, with a period (.), or by juxtaposition; substrings & \begin{tabular}{l}
AND, OR, NOT, XOR \\
Parentheses permitted
\end{tabular} \\
\hline Range of values & \(-2^{31}\) to \(+2^{31}-1\) & 0 through 1024 characters & 0 (false) or 1 (true) \\
\hline
\end{tabular}

\section*{Macro and Conditional Assembly Language Summary}

Figure 116 (Page 2 of 2). Conditional Assembly Expressions
\begin{tabular}{llll}
\hline Expression & \begin{tabular}{l} 
Arithmetic \\
Expressions
\end{tabular} & \begin{tabular}{l} 
Character \\
Expressions
\end{tabular} & \begin{tabular}{l} 
Logical \\
Expressions
\end{tabular} \\
\hline Used in & SETA operands & SETC operands & SETB operands \\
& \begin{tabular}{l} 
Created SET \\
symbols
\end{tabular} & Character relations & AIF operands \\
& \begin{tabular}{l} 
Subscripted SET \\
symbols
\end{tabular} & Created SET symbols \\
\&SYSLIST \\
subscript(s) & & & \\
Substring notation & & & \\
\hline
\end{tabular}

Built-in functions fall into the following categories:

Figure 117. Built-in functions
\begin{tabular}{ll} 
Value Type & Functions \\
\hline Arithmetic & \begin{tabular}{l} 
AND, B2A, C2A, D2A, DCLEN, FIND, INDEX, NOT, OR, SLA, SLL, \\
SRA, SRL, X2A, XOR
\end{tabular} \\
\hline Logical & \begin{tabular}{ll} 
AND, AND NOT, ISBIN, ISDEC, ISHEX, ISSYM, NOT, OR, OR \\
& NOT, XOR, XOR NOT,
\end{tabular} \\
\hline Character & \begin{tabular}{l} 
A2B, A2C, A2D, A2X, B2C, B2D, B2X, BYTE, C2B, C2D, C2X, \\
\\
\\
\\
\\
\\
\\
\end{tabular} UPPB, D2C, D22, DCVAL, DEQUOTE, DOUBLE, LOWER, SIGNED, \\
\hline
\end{tabular}

Figure 118 (Page 1 of 2). Attributes
\begin{tabular}{|c|c|c|c|c|}
\hline Attribute & Notation & Can be used with: & Can be used only if Type Attribute is: & Can be used in: \\
\hline Type & T' & Ordinary symbols defined in open code; symbolic parameters inside macro definitions; \&SYSLIST(n), \&SYSLIST( \(n, m\) ) inside macro definitions; SET symbols; all system variable symbols & Any value & \begin{tabular}{l}
SETC operand fields \\
Character relations
\end{tabular} \\
\hline Length & L' & Ordinary symbols defined in open code; symbolic parameters inside macro definitions; \&SYSLIST( \(n\) ), and \&SYSLIST \((n, m)\) inside macro definitions & Any value except M , \(\mathrm{N}, \mathrm{O}, \mathrm{T}, \mathrm{U}\) & Arithmetic expressions \\
\hline
\end{tabular}

Figure 118 (Page 2 of 2). Attributes
\begin{tabular}{|c|c|c|c|c|}
\hline Attribute & Notation & Can be used with: & Can be used only if Type Attribute is: & Can be used in: \\
\hline Scale & S' & Ordinary symbols defined in open code; symbolic parameters inside macro definitions; \&SYSLIST( \(n\) ), and \&SYSLIST( \(n, m\) ) inside macro definitions & H,F,G,D,E,L,K,P, and Z & Arithmetic expressions \\
\hline Integer & 11 & Ordinary symbols defined in open code; symbolic parameters inside macro definitions; \&SYSLIST( \(n\) ), and \&SYSLIST( \(n, m\) ) inside macro definitions & H,F,G,D,E,L,K,P, and Z & Arithmetic expressions \\
\hline Count & K' & Symbolic parameters inside macro definitions; \&SYSLIST( \(n\) ), and \&SYSLIST( \(n, m\) ) inside macro definitions; SET symbols; all system variable symbols & Any letter or @ & Arithmetic expressions \\
\hline Number & N' & Symbolic parameters, \&SYSLIST and \&SYSLIST( \(n\) ) inside macro definitions, with dimensioned SET symbols & Any letter & Arithmetic expressions \\
\hline Defined & D' & Ordinary symbols defined in open code; symbolic parameters inside macro definitions; \&SYSLIST and \&SYSLIST( \(n\) ) inside macro definitions; SETC symbols whose value is an ordinary symbol & Any value except \(M\), \(\mathrm{N}, \mathrm{O}, \mathrm{T}, \mathrm{U}\) & Arithmetic expressions \\
\hline Operation Code & \(\mathrm{O}^{\prime}\) & A character string, or variable symbol containing a character string. & The @ and any letter except N, O and (only sometimes) U & SETC operand fields Character relations \\
\hline
\end{tabular}

Refer to Chapter 9, "How to Write Conditional Assembly Instructions" on page 318 for usage restrictions of the attributes in Figure 118.

Figure 119 (Page 1 of 2). Variable Symbols
\begin{tabular}{lllll}
\hline Variable & \begin{tabular}{lll} 
Declared \\
by:
\end{tabular} & \begin{tabular}{l} 
Initialized \\
or set to:
\end{tabular} & \begin{tabular}{l} 
Value \\
changed \\
by:
\end{tabular} & \begin{tabular}{l} 
May be used \\
in:
\end{tabular} \\
\hline Symbolic \({ }^{1}\) parameter & \begin{tabular}{lll} 
Prototype \\
statement
\end{tabular} & \begin{tabular}{l} 
Corresponding \\
macro \\
instruction \\
operand
\end{tabular} & \begin{tabular}{l} 
Constant \\
throughout \\
definition
\end{tabular} & \begin{tabular}{l} 
Arithmetic expressions \\
if operand is \\
self-defining term
\end{tabular} \\
& & & & Character expressions \\
\hline
\end{tabular}

Figure 119 (Page 2 of 2). Variable Symbols
\begin{tabular}{lllll}
\hline \begin{tabular}{l} 
Variable \\
Symbol
\end{tabular} & \begin{tabular}{l} 
Declared \\
by:
\end{tabular} & \begin{tabular}{l} 
Initialized \\
or set to:
\end{tabular} & \begin{tabular}{l} 
Value \\
changed \\
by:
\end{tabular} & \begin{tabular}{l} 
May be used \\
in:
\end{tabular} \\
\hline SETA & \begin{tabular}{l} 
LCLA or \\
GBLA \\
instruction
\end{tabular} & 0 & \begin{tabular}{l} 
SETA \\
instruction
\end{tabular} & \begin{tabular}{l} 
Arithmetic expressions \\
Character expressions \\
Logical expressions
\end{tabular} \\
\hline SETB & \begin{tabular}{l} 
LCLB or \\
GBLB \\
instruction
\end{tabular} & 0 & \begin{tabular}{l} 
SETB \\
instruction
\end{tabular} & \begin{tabular}{l} 
Arithmetic expressions \\
Character expressions
\end{tabular} \\
& & \begin{tabular}{l} 
LCLC or \\
GBLC \\
instruction
\end{tabular} & \begin{tabular}{l} 
String of \\
length 0 (null)
\end{tabular} & \begin{tabular}{l} 
SETC \\
instruction
\end{tabular} \\
\hline SETC & & \begin{tabular}{l} 
Arithmetic expressions \\
if value is self-defining \\
term
\end{tabular} \\
& & & & \begin{tabular}{l} 
Character expressions
\end{tabular} \\
& & & & \begin{tabular}{l} 
Logical expressions if \\
value is self-defining \\
term
\end{tabular} \\
\hline
\end{tabular}

Notes:
1. Can be used only in macro definitions.

Figure 120 (Page 1 of 5). System Variable Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline System Variable Symbol & Availability \({ }^{1}\) & Type \({ }^{2}\) & Type Attr. \({ }^{3}\) & Scope & Initialized or set to & Value changed by & May be used in \\
\hline \&SYSADATA_DSN & HLA2 & C & U,O & L & Current associated data file & Constant throughout assembly & Character expressions \\
\hline \&SYSADATA_MEMBER & HLA2 & C & U,O & L & Current associated data file member name & Constant throughout assembly & Character expressions \\
\hline \&SYSADATA_VOLUME & HLA2 & C & U,O & L & Current associated data file volume identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSASM & HLA1 & C & U & G & Assembler name & Constant throughout assembly & Character expression \\
\hline \&SYSCLOCK & HLA3 & C & U & L & Current date and time & Constant throughout macro expansion & Character expressions \\
\hline \&SYSDATC & HLA1 & C,A & N & G & Assembly date (with century) & Constant throughout assembly & Arithmetic expressions Character expressions \\
\hline \&SYSDATE & AsmH & C & U & G & Assembly date & Constant throughout assembly & Character expressions \\
\hline \&SYSECT & All & C & U & L & Name of control section in effect where macro instruction appears & Constant throughout definition; set by START, CSECT, RSECT, DSECT, or COM & Character expressions \\
\hline
\end{tabular}

Figure 120 (Page 2 of 5). System Variable Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline System Variable Symbol & Availability \({ }^{1}\) & Type \({ }^{2}\) & Type Attr. \({ }^{3}\) & Scope & Initialized or set to & Value changed by & May be used in \\
\hline \&SYSIN_DSN & HLA1 & C & U & L & Current primary input data set name & Constant throughout definition & Character expressions \\
\hline \&SYSIN_MEMBER & HLA1 & C & U,O & L & Current primary input member name & Constant throughout definition & Character expressions \\
\hline \&SYSIN_VOLUME & HLA1 & C & U,O & L & Current primary input volume identifier & Constant throughout definition & Character expressions \\
\hline \&SYSJOB & HLA1 & C & U & G & Source module assembly jobname & Constant throughout assembly & Character expressions \\
\hline \&SYSLIB_DSN & HLA1 & C & U & L & Current macro library filename & Constant throughout definition & Character expressions \\
\hline \&SYSLIB_MEMBER & HLA1 & C & U,O & L & Current macro library member name & Constant throughout definition & Character expressions \\
\hline \&SYSLIB_VOLUME & HLA1 & C & U,O & L & Current macro library volume identifier & Constant throughout definition & Character expressions \\
\hline \&SYSLIN_DSN & HLA2 & C & U & L & Current object data set name & Constant throughout assembly & Character expressions \\
\hline \&SYSLIN_MEMBER & HLA2 & C & U,O & L & Current object data set member name & Constant throughout assembly & Character expressions \\
\hline \&SYSLIN_VOLUME & HLA2 & C & U,O & L & Current object data set volume identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSLIST & All & C & any & L & Not applicable & Not applicable & N'\&SYSLIST in arithmetic expressions \\
\hline \&SYSLIST( \(n\) ) \&SYSLIST(n,m) & All & C & any & L & Corresponding macro instruction operand & Constant throughout definition & \begin{tabular}{l}
Arithmetic expressions if operand is self-defining term \\
Character expressions
\end{tabular} \\
\hline \&SYSLOC & AsmH & C & U & L & Location counter in effect where macro instruction appears & \begin{tabular}{l}
Constant \\
throughout \\
definition; \\
set by \\
START, \\
CSECT, \\
RSECT, \\
DSECT, \\
COM, \\
and \\
LOCTR
\end{tabular} & Character expressions \\
\hline \&SYSMAC & HLA3 & C & U, O & L & Macro name & Constant throughout definition & Arithmetic expressions \\
\hline \&SYSMAC \((n)_{1}\) & HLA3 & C & U,O & L & Ancestor macro name & Constant throughout definition & Arithmetic expressions \\
\hline
\end{tabular}

Figure 120 (Page 3 of 5). System Variable Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline System Variable Symbol & Availability \({ }^{1}\) & Type \({ }^{2}\) & Type Attr. \({ }^{3}\) & Scope & Initialized or set to & Value changed by & May be used in \\
\hline \&SYSM_HSEV & HLA3 & A & N & G & 0 & Mnote & Arithmetic expressions \\
\hline \&SYSM_SEV & HLA3 & A & N & G & 0 & \begin{tabular}{l}
At \\
nesting \\
and \\
unnesting \\
of \\
macros, \\
from \\
MNOTE
\end{tabular} & Arithmetic expressions \\
\hline \&SYSNDX & All & C & N & L & Macro instruction index & Constant throughout definition; unique for each macro instruction & Arithmetic expressions Character expressions \\
\hline \&SYSNEST & HLA1 & A & N & L & Macro instruction nesting level & Constant throughout definition; unique for each macro nesting level & Arithmetic expressions Character expressions \\
\hline \&SYSOPT_DBCS & HLA1 & B & N & G & DBCS assembler option indicator & Constant throughout assembly & Arithmetic expressions Character expressions Logical expressions \\
\hline \&SYSOPT_OPTABLE & HLA3 & C & U & G & OPTABLE assembler option value & Constant throughout assembly & Character expressions \\
\hline \&SYSOPT_RENT & HLA1 & B & N & G & RENT assembler option indicator & Constant throughout assembly & Arithmetic expressions Character expressions Logical expressions \\
\hline \&SYSOPT_XOBJECT & HLA3 & B & N & G & XOBJECT assembler option indicator & Constant throughout assembly & Arithmetic expressions Character expressions Logical expressions \\
\hline \&SYSPARM & All & C & U, O & G & User defined or null & Constant throughout assembly & Arithmetic expressions if value is self-defining term Character expressions \\
\hline \&SYSPRINT_DSN & HLA2 & C & U & L & Current assembler listing data set name & Constant throughout assembly & Character expressions \\
\hline \&SYSPRINT_MEMBER & HLA2 & C & U,O & L & Current assembler listing data set member name & Constant throughout assembly & Character expressions \\
\hline \&SYSPRINT_VOLUME & HLA2 & C & U,O & L & Current assembler listing data set volume identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSPUNCH_DSN & HLA2 & C & U & L & Current object data set name & Constant throughout assembly & Character expressions \\
\hline
\end{tabular}

Figure 120 (Page 4 of 5). System Variable Symbols
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline System Variable Symbol & Availability \({ }^{1}\) & Type \({ }^{2}\) & Type Attr. \({ }^{3}\) & Scope & Initialized or set to & Value changed by & May be used in \\
\hline \&SYSPUNCH_MEMBER & HLA2 & C & U,O & L & Current object data set member name & Constant throughout assembly & Character expressions \\
\hline \&SYSPUNCH_VOLUME & HLA2 & C & U,O & L & Current object data set volume identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSSEQF & HLA1 & C & U,O & L & Outer-most macro instruction identification- sequence field & Constant throughout definition & Character expressions \\
\hline \&SYSSTEP & HLA1 & C & U & G & Source module assembly job name & Constant throughout assembly & Character expressions \\
\hline \&SYSSTMT & HLA1 & C,A & N & G & Next statement number & Assembler increments each time a statement is processed & Arithmetic expressions Character expressions \\
\hline \&SYSSTYP & HLA1 & C & U,O & L & Type of control section in effect where macro instruction appears & \begin{tabular}{l}
Constant \\
throughout \\
definition; \\
set by \\
START, \\
CSECT, \\
RSECT, \\
DSECT, \\
or COM
\end{tabular} & Character expressions \\
\hline \&SYSTEM_ID & HLA1 & C & U & G & Assembly operating system environment identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSTERM_DSN & HLA2 & C & U & L & Current terminal data set name & Constant throughout assembly & Character expressions \\
\hline \&SYSTERM_MEMBER & HLA2 & C & U,O & L & Current terminal data set member name & Constant throughout assembly & Character expressions \\
\hline \&SYSTERM_VOLUME & HLA2 & C & U,O & L & Current terminal data set volume identifier & Constant throughout assembly & Character expressions \\
\hline \&SYSTIME & AsmH & C & U & G & Source module assembly time & Constant throughout assembly & Character expressions \\
\hline
\end{tabular}

\section*{Macro and Conditional Assembly Language Summary}

Figure 120 (Page 5 of 5). System Variable Symbols
\begin{tabular}{llllllll}
\hline System Variable & \begin{tabular}{l} 
Avail- \\
ability \(^{1}\)
\end{tabular} & Type \(^{2}\) & Type & Attr. \({ }^{3}\) & Scope & & \begin{tabular}{l} 
Value \\
changed
\end{tabular} \\
\begin{tabular}{llllll} 
Symbol
\end{tabular} & HLA1 & C & U & G & \begin{tabular}{l} 
Assembler release \\
level
\end{tabular} & \begin{tabular}{l} 
Constant \\
throughout
\end{tabular} & Character expressions \\
asySVER & & & & & & assembly
\end{tabular}

\section*{Notes:}
1. Availability:

All All assemblers, including the DOS/VSE Assembler
AsmH Assembler H Version 2 and High Level Assembler
HLA1 High Level Assembler Release 1
HLA2 High Level Assembler Release 2
HLA3 High Level Assembler Release 3
HLA4 High Level Assembler Release 4
HLA5 High Level Assembler Release 5
2. Type:

A Arithmetic
B Boolean
C Character
3. Type Attr:
\(\mathrm{N} \quad\) Numeric (self-defining term)
O Omitted
U Undefined, unknown, deleted or unassigned
4. Scope:

L Local - only in macro
G Global - in entire program

\section*{Appendix D. Standard Character Set Code Table}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline \multicolumn{8}{|l|}{Figure 121. Standard Character Set Code Table - From Code Page 00037} \\
\hline Hex. & Dec. & EBCDIC & Binary & Hex. & Dec. & EBCDIC & Binary \\
\hline 00 & 0 & & 00000000 & 20 & 32 & & 00100000 \\
\hline 01 & 1 & & 00000001 & 21 & 33 & & 00100001 \\
\hline 02 & 2 & & 00000010 & 22 & 34 & & 00100010 \\
\hline 03 & 3 & & 00000011 & 23 & 35 & & 00100011 \\
\hline 04 & 4 & & 00000100 & 24 & 36 & & 00100100 \\
\hline 05 & 5 & & 00000101 & 25 & 37 & & 00100101 \\
\hline 06 & 6 & & 00000110 & 26 & 38 & & 00100110 \\
\hline 07 & 7 & & 00000111 & 27 & 39 & & 00100111 \\
\hline 08 & 8 & & 00001000 & 28 & 40 & & 00101000 \\
\hline 09 & 9 & & 00001001 & 29 & 41 & & 00101001 \\
\hline 0A & 10 & & 00001010 & 2A & 42 & & 00101010 \\
\hline 0B & 11 & & 00001011 & 2 B & 43 & & 00101011 \\
\hline 0 C & 12 & & 00001100 & 2 C & 44 & & 00101100 \\
\hline 0D & 13 & & 00001101 & 2D & 45 & & 00101101 \\
\hline 0E & 14 & & 00001110 & 2 E & 46 & & 00101110 \\
\hline 0F & 15 & & 00001111 & 2 F & 47 & & 00101111 \\
\hline 10 & 16 & & 00010000 & 30 & 48 & & 00110000 \\
\hline 11 & 17 & & 00010001 & 31 & 49 & & 00110001 \\
\hline 12 & 18 & & 00010010 & 32 & 50 & & 00110010 \\
\hline 13 & 19 & & 00010011 & 33 & 51 & & 00110011 \\
\hline 14 & 20 & & 00010100 & 34 & 52 & & 00110100 \\
\hline 15 & 21 & & 00010101 & 35 & 53 & & 00110101 \\
\hline 16 & 22 & & 00010110 & 36 & 54 & & 00110110 \\
\hline 17 & 23 & & 00010111 & 37 & 55 & & 00110111 \\
\hline 18 & 24 & & 00011000 & 38 & 56 & & 00111000 \\
\hline 19 & 25 & & 00011001 & 39 & 57 & & 00111001 \\
\hline 1A & 26 & & 00011010 & 3A & 58 & & 00111010 \\
\hline 1B & 27 & & 00011011 & 3B & 59 & & 00111011 \\
\hline 1C & 28 & & 00011100 & 3C & 60 & & 00111100 \\
\hline 1D & 29 & & 00011101 & 3D & 61 & & 00111101 \\
\hline 1E & 30 & & 00011110 & 3E & 62 & & 00111110 \\
\hline 1F & 31 & & 00011111 & 3 F & 63 & & 00111111 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline Hex. & Dec. & EBCDIC & Binary & Hex. & Dec. & EBCDIC & Binary \\
\hline 40 & 64 & SPACE & 01000000 & 60 & 96 & - & 01100000 \\
\hline 41 & 65 & & 01000001 & 61 & 97 & / & 01100001 \\
\hline 42 & 66 & & 01000010 & 62 & 98 & & 01100010 \\
\hline 43 & 67 & & 01000011 & 63 & 99 & & 01100011 \\
\hline 44 & 68 & & 01000100 & 64 & 100 & & 01100100 \\
\hline 45 & 69 & & 01000101 & 65 & 101 & & 01100101 \\
\hline 46 & 70 & & 01000110 & 66 & 102 & & 01100110 \\
\hline 47 & 71 & & 01000111 & 67 & 103 & & 01100111 \\
\hline 48 & 72 & & 01001000 & 68 & 104 & & 01101000 \\
\hline 49 & 73 & & 01001001 & 69 & 105 & & 01101001 \\
\hline 4A & 74 & & 01001010 & 6A & 106 & & 01101010 \\
\hline 4B & 75 & . & 01001011 & 6B & 107 & , & 01101011 \\
\hline 4C & 76 & & 01001100 & 6C & 108 & & 01101100 \\
\hline 4D & 77 & ( & 01001101 & 6D & 109 & - & 01101101 \\
\hline 4E & 78 & + & 01001110 & 6E & 110 & & 01101110 \\
\hline 4F & 79 & & 01001111 & 6F & 111 & & 01101111 \\
\hline 50 & 80 & \& & 01010000 & 70 & 112 & & 01110000 \\
\hline 51 & 81 & & 01010001 & 71 & 113 & & 01110001 \\
\hline 52 & 82 & & 01010010 & 72 & 114 & & 01110010 \\
\hline 53 & 83 & & 01010011 & 73 & 115 & & 01110011 \\
\hline 54 & 84 & & 01010100 & 74 & 116 & & 01110100 \\
\hline 55 & 85 & & 01010101 & 75 & 117 & & 01110101 \\
\hline 56 & 86 & & 01010110 & 76 & 118 & & 01110110 \\
\hline 57 & 87 & & 01010111 & 77 & 119 & & 01110111 \\
\hline 58 & 88 & & 01011000 & 78 & 120 & & 01111000 \\
\hline 59 & 89 & & 01011001 & 79 & 121 & & 01111001 \\
\hline 5A & 90 & & 01011010 & 7A & 122 & & 01111010 \\
\hline 5B & 91 & \$ & 01011011 & 7B & 123 & \# & 01111011 \\
\hline 5C & 92 & * & 01011100 & 7 C & 124 & @ & 01111100 \\
\hline 5D & 93 & ) & 01011101 & 7D & 125 & ' & 01111101 \\
\hline 5E & 94 & & 01011110 & 7E & 126 & = & 01111110 \\
\hline 5F & 95 & & 01011111 & 7F & 127 & & 01111111 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline Hex. & Dec. & EBCDIC & Binary & Hex. & Dec. & EBCDIC & Binary \\
\hline 80 & 128 & & 10000000 & A0 & 160 & & 10100000 \\
\hline 81 & 129 & a & 10000001 & A1 & 161 & & 10100001 \\
\hline 82 & 130 & b & 10000010 & A2 & 162 & s & 10100010 \\
\hline 83 & 131 & c & 10000011 & A3 & 163 & t & 10100011 \\
\hline 84 & 132 & d & 10000100 & A4 & 164 & \(u\) & 10100100 \\
\hline 85 & 133 & e & 10000101 & A5 & 165 & v & 10100101 \\
\hline 86 & 134 & f & 10000110 & A6 & 166 & W & 10100110 \\
\hline 87 & 135 & g & 10000111 & A7 & 167 & x & 10100111 \\
\hline 88 & 136 & h & 10001000 & A8 & 168 & y & 10101000 \\
\hline 89 & 137 & i & 10001001 & A9 & 169 & z & 10101001 \\
\hline 8A & 138 & & 10001010 & AA & 170 & & 10101010 \\
\hline 8B & 139 & & 10001011 & AB & 171 & & 10101011 \\
\hline 8C & 140 & & 10001100 & AC & 172 & & 10101100 \\
\hline 8D & 141 & & 10001101 & AD & 173 & & 10101101 \\
\hline 8E & 142 & & 10001110 & AE & 174 & & 10101110 \\
\hline 8F & 143 & & 10001111 & AF & 175 & & 10101111 \\
\hline 90 & 144 & & 10010000 & B0 & 176 & & 10110000 \\
\hline 91 & 145 & j & 10010001 & B1 & 177 & & 10110001 \\
\hline 92 & 146 & k & 10010010 & B2 & 178 & & 10110010 \\
\hline 93 & 147 & 1 & 10010011 & B3 & 179 & & 10110011 \\
\hline 94 & 148 & m & 10010100 & B4 & 180 & & 10110100 \\
\hline 95 & 149 & n & 10010101 & B5 & 181 & & 10110101 \\
\hline 96 & 150 & 0 & 10010110 & B6 & 182 & & 10110110 \\
\hline 97 & 151 & p & 10010111 & B7 & 183 & & 10110111 \\
\hline 98 & 152 & q & 10011000 & B8 & 184 & & 10111000 \\
\hline 99 & 153 & r & 10011001 & B9 & 185 & & 10111001 \\
\hline 9A & 154 & & 10011010 & BA & 186 & & 10111010 \\
\hline 9B & 155 & & 10011011 & BB & 187 & & 10111011 \\
\hline 9 C & 156 & & 10011100 & BC & 188 & & 10111100 \\
\hline 9 D & 157 & & 10011101 & BD & 189 & & 10111101 \\
\hline 9E & 158 & & 10011110 & BE & 190 & & 10111110 \\
\hline 9F & 159 & & 10011111 & BF & 191 & & 10111111 \\
\hline
\end{tabular}
\begin{tabular}{|l|l|l|l|l|l|l|l|}
\hline Hex. & Dec. & EBCDIC & Binary & Hex. & Dec. & EBCDIC & Binary \\
\hline C0 & 192 & & 11000000 & E0 & 224 & & 11100000 \\
\hline C1 & 193 & A & 11000001 & E1 & 225 & & 11100001 \\
\hline C2 & 194 & B & 11000010 & E2 & 226 & S & 11100010 \\
\hline C3 & 195 & C & 11000011 & E3 & 227 & T & 11100011 \\
\hline C4 & 196 & D & 11000100 & E4 & 228 & U & 11100100 \\
\hline C5 & 197 & E & 11000101 & E5 & 229 & V & 11100101 \\
\hline C6 & 198 & F & 11000110 & E6 & 230 & W & 11100110 \\
\hline C7 & 199 & G & 11000111 & E7 & 231 & X & 11100111 \\
\hline C8 & 200 & H & 11001000 & E8 & 232 & Y & 11101000 \\
\hline C9 & 201 & I & 11001001 & E9 & 233 & Z & 11101001 \\
\hline CA & 202 & & 11001010 & EA & 234 & & 11101010 \\
\hline CB & 203 & & 11001011 & EB & 235 & & 11101011 \\
\hline CC & 204 & & 11001100 & EC & 236 & & 11101100 \\
\hline CD & 205 & & 11001101 & ED & 237 & & 11101101 \\
\hline CE & 206 & & 11001110 & EE & 238 & & 11101110 \\
\hline CF & 207 & & 11001111 & EF & 239 & & 11101111 \\
\hline D0 & 208 & & 11010000 & F0 & 240 & 0 & 11110000 \\
\hline D1 & 209 & J & 11010001 & F1 & 241 & 1 & 11110001 \\
\hline D2 & 210 & K & 11010010 & F2 & 242 & 2 & 11110010 \\
\hline D3 & 211 & L & 11010011 & F3 & 243 & 3 & 11110011 \\
\hline D4 & 212 & M & 11010100 & F4 & 244 & 4 & 11110100 \\
\hline D5 & 213 & N & 11010101 & F5 & 245 & 5 & 11110101 \\
\hline D6 & 214 & 0 & 11010110 & F6 & 246 & 6 & 11110110 \\
\hline D7 & 215 & P & 11010111 & F7 & 247 & 7 & 11110111 \\
\hline D8 & 216 & Q & 11011000 & F8 & 248 & 8 & 11111000 \\
\hline D9 & 217 & R & 11011001 & F9 & 249 & 9 & 11111001 \\
\hline DA & 218 & & 11011010 & FA & 250 & & 11111010 \\
\hline DB & 219 & & 11011011 & FB & 251 & & 11111011 \\
\hline DC & 220 & & 11011100 & FC & 252 & & 11111100 \\
\hline DD & 221 & & & FD & 253 & & 11111101 \\
\hline DE & 222 & & FE & 254 & & 11111110 \\
\hline FF & 255 & & 11111111 \\
\hline
\end{tabular}

\section*{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:

\section*{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
2455 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:

\author{
IBM World Trade Asia Corporation Licensing \\ 2-31 Roppongi 3-chome, Minato-ku \\ Tokyo 106-0032, 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.

\section*{Trademarks}

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

\section*{Trademarks}
AIX
BookMaster
CICS
DFSMS
DFSMS/MVS
DFSMSdfp
Enterprise Systems Architecture/370
Enterprise Systems Architecture/390
IBM
Language Environment
MVS
MVS/DFP
MVS/ESA
OS/390
SP
System/370
System/390
VM/ESA
VSE/ESA
Z/OS

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

\section*{Bibliography}

\section*{High Level Assembler Publications}
HLASM General Information GC26-4943
HLASM Installation and Customization Guide
SC26-3494
HLASM Language Reference sC26-4940
HLASM Licensed Program Specifications,
GC26-4944
HLASM Programmer's Guide, SC26-4941

\section*{Toolkit Feature Publications}

HLASM Toolkit Feature User's Guide GC26-8710
HLASM Toolkit Feature Debug Reference Summary GC26-8712

HLASM Toolkit Feature Interactive Debug Facility User's Guide, GC26-8709
HLASM Toolkit Feature Installation and Customization Guide, GC26-8711

\section*{Related Publications (Architecture)}

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

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

\section*{Related Publications for MVS}
z/OS:
z/OS MVS JCL Reference SA22-7597
z/OS MVS JCL User's Guide SA22-7598
z/OS MVS Programming: Assembler Services Guide SA22-7605
z/OS MVS Programming: Assembler Services Reference, Volumes 1 and 2, SA22-7606, SA22-7607
z/OS MVS Programming: Authorized Assembler Services Guide SA22-7608
z/OS MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, SA22-7609-SA22-7612
z/OS MVS Program Management: User's Guide and Reference SA22-7643
z/OS MVS System Codes SA22-7626
z/OS MVS System Commands SA22-7627
z/OS MVS System Messages, Volumes 1-10,
SA22-7631-SA22-7640
OS/390:
OS/390 MVS JCL Reference, GC28-1757
OS/390 MVS JCL User's Guide, GC28-1758
OS/390 MVS Programming: Assembler Services
Reference, GC28-1910
OS/390 MVS System Codes, GC28-1780
OS/390 MVS System Commands, GC28-1781
OS/390 MVS System Messages, Volumes 1-5, GC28-1784 - GC27-1788

OpenEdition:
MVS/ESA OpenEdition MVS User's Guide, SC23-3013
z/OS UNIX System Services User's Guide, SA22-7801

MVS/DFP \({ }^{\text {™ }}\) :
MVS/DFP Version 3.3: Utilities, SC26-4559
MVS/DFP Version 3.3: Linkage Editor and Loader, SC26-4564

DFSMS/MVS®:
I z/OS DFSMS Program Management, SC27-1130
z/OS DFSMSdfp Utilities, SC26-7414
TSO/E (z/OS):
I z/OS TSO/E Command Reference, SA22-7782
TSO/E (OS/390):
OS/390 TSO/E Command Reference, SC28-1869
SMP/E (z/OS):
SMP/E Messages, Codes and Diagnosis, GA22-7770

SMP/E Reference, SA22-7772
SMP/E User's Guide, SA22-7773
SMP/E (OS/390):
SMP/E Messages and Codes, SC28-1738

\section*{Bibliography}

z/VM XEDIT Command and Macro Reference,
SC24-5973
I
z/VM CP Command and Utility Reference,
SC24-5967
i z/VM Planning and Administration, SC24-5995
z/VM Service Guide, GC24-5946
I
Z/VM VMSES/E Introduction and Reference,
GC24-5994

\section*{Related Publications for VSE}

VSE/ESA Administration, SC33-6705
VSE/ESA Guide to System Functions, SC33-6711
VSE/ESA Installation, SC33-6704
VSE/ESA Planning, SC33-6703
VSE/ESA System Control Statements, SC33-6713
VSE/ESA Messages and Codes, Vols. 1-3, SC33-6796, SC33-6798, SC33-6799

\section*{Index}

\section*{Special Characters}

See pairing rules
See quotation marks
1
as division operator 358
concatenation operator for strings 383
in qualified symbols 66
*
See asterisks (*)
*PROCESS statement 102
initiating the first control section 54
restricted options 102
\&
See ampersands (\&)
See pairing rules
See variable symbols
\&SYS
as start of system variable names 262
\&SYSADATA_DSN system variable symbol 263
\&SYSADATA_MEMBER system variable symbol 264
\&SYSADATA_VOLUME system variable symbol 265
\&SYSASM system variable symbol 265
\&SYSCLOCK system variable symbol 266
\&SYSDATC system variable symbol 266
\&SYSDATE system variable symbol 267
\&SYSECT system variable symbol 267
CSECT 268
DSECT 268
RSECT 268
\&SYSIN_DSN system variable symbol 269
\&SYSIN_MEMBER system variable symbol 270
\&SYSIN_VOLUME system variable symbol 271
\&SYSJOB system variable symbol 272
\&SYSLIB_DSN system variable symbol 272
\&SYSLIB_MEMBER system variable symbol 273
\&SYSLIB_VOLUME system variable symbol 273
\&SYSLIN_DSN system variable symbol 274
\&SYSLIN_MEMBER system variable symbol 275
\&SYSLIN_VOLUME system variable symbol 275
\&SYSLIST system variable symbol 276
\&SYSLOC system variable symbol 278
LOCTR 278
\&SYSM_HSEV system variable symbol 279
\&SYSM_SEV system variable symbol 280
\&SYSMAC system variable symbol 279
\&SYSNDX system variable symbol
controlling its value using MHELP 398
definition 281
\&SYSNEST system variable symbol 283
\&SYSOPT_DBCS system variable symbol 284
\&SYSOPT_OPTABLE system variable symbol 285
\&SYSOPT_RENT system variable symbol 285
\&SYSOPT_XOBJECT system variable symbol 285
\&SYSPARM system variable symbol 286
\&SYSPRINT_DSN system variable symbol 286
\&SYSPRINT_MEMBER system variable symbol 288
\&SYSPRINT_VOLUME system variable symbol 288
\&SYSPUNCH_DSN system variable symbol 289
\&SYSPUNCH_MEMBER system variable symbol 290
\&SYSPUNCH_VOLUME system variable symbol 290
\&SYSSEQF system variable symbol 291
\&SYSSTEP system variable symbol 292
\&SYSSTMT system variable symbol 292
\&SYSSTYP system variable symbol 292
CSECT 293
DSECT 293
RSECT 293
\&SYSTEM_ID system variable symbol 293
\&SYSTERM_DSN system variable symbol 294
\&SYSTERM_MEMBER system variable symbol 295
\&SYSTERM_VOLUME system variable symbol 295
\&SYSTIME system variable symbol 296
\&SYSVER system variable symbol 296
-
as negative unary operator 358
as subtraction operator 358
\(+\)
as addition operator 358
as positive unary operator 358

\section*{=}
defining a literal constant 171

\section*{Numerics}

24
AMODE instruction 111
RMODE instruction 211
31
AMODE instruction 111
RMODE instruction 211
64
AMODE instruction 111
RMODE instruction 211
64-bit addressing mode 101

\section*{A}

A-type address constant 153
A2B (SETC built-in function) 374

\section*{Index}

A2C (SETC built-in function) 375
A2D (SETC built-in function) 375
A2X (SETC built-in function) 375
absolute addresses
base registers for 220
absolute expression 47
absolute symbol
defined 31
absolute terms 28
ACONTROL
PUSH instruction 209
ACONTROL instruction 103
ACTR instruction 394
ADATA assembler option 108
ADATA instruction 107
address constants
A-type 153
complex relocatable 153
J-type 160
Q-type 159
R-type 160
S-type 156
V-type 157
Y-type 153
addressability
by means of the DROP instruction 172
by means of the USING instruction 218
dependent 67
establishing 64
qualified 66
relative 67
using base register instructions 66
addresses
absolute 87
explicit 64
implicit 64
relocatable 87
addressing mode
64-bit 101
addressing mode (AMODE) 68
AEJECT instruction 257
AFPR assembler option 86,103
AGO instruction
alternative statement format 393
general statement format 392

\section*{AGOB}
as synonym of AGO instruction 394
AIF instruction 390
alternative statement format 392
AIFB
as synonym of AIF instruction 392
AINSERT instruction 108, 257
ALIAS instruction 109
maximum operand length 110
ALIGN
suboption of FLAG 105

ALIGN assembler option 129, 176
alphabetic character defined 30
alternative statement format 246
AGO instruction 393
AIF instruction 392
continuation lines 16
extended SET statements 387
GBLx instructions 344
LCLx instructions 345
macro instructions 298
prototype statements 246
summary 16
AMODE
indicators in ESD 68
instruction to specify addressing mode 110
ampersands (\&)
as special character 308
DBCS ampersand not delimiter 197
in variable symbols 30
not recognized in double-byte data 145
paired in CA string 144
paired in MNOTE message 197
paired in PUNCH instruction 208
paired in Unicode data 144
pairing in character relations 335
pairing in DC 144
representation in character constant 143, 208
variable symbol identifier in PUNCH statement 208
AND (SETA built-in function) 353
AND (SETB built-in function) 365
AND NOT (SETB built-in function) 365
ANOP instruction 395
ANY
AMODE instruction 111
RMODE instruction 211
ANY31
AMODE instruction 111
ANY64
AMODE instruction 111
apostrophes
paired in CA string 144
AREAD instruction 257
CLOCKB operand 258
CLOCKD operand 258
arithmetic (SETA) expressions
built-in functions 353
evaluation of 359
rules for coding 358
SETC variables in 359
using 347
arithmetic external function calls 388
arithmetic relations in logical expressions 367
array 322
dimensioned 319
subscripted 319

ASCII character constants 144
type extension 135
ASCII translation table 14
ASPACE instruction 259
assembler instruction statements
base register instructions 66
data definition instructions 126
exit-control parameters 187
listing control instructions 215
operation code definition instruction 198
program control instructions 189
program sectioning and linking instructions 51
symbol definition instructions 184
assembler language
assembler instruction statements 4
coding aids summary 9
coding conventions of 14
coding form for 14
compatibility with other languages 4
conditional assembly instructions 318
introduction to 3
machine instruction statements 4, 78
macro instruction statements 4, 297
statements
summary of 405
structure of 21
summary of instructions 402
assembler options
ADATA 108
AFPR 86, 103
ALIGN 129, 176
BATCH 341
CODEPAGE 144
COMPAT 12, 103, 278, 304, 307, 308, 314, 322, 386
controlling output using 6
DBCS 12, 15, 34, 36, 145, 197, 208, 217, 247, 249, 251, 253, 284, 298, 301, 302, 309, 360, 404, 413
DECK 188, 289, 290
EXIT 188
FLAG 16, 103, 198, 247, 373
FOLD 14
GOFF \(37,62,72,108,111,112,124,125,160\), 216, 274, 275
LIBMAC 103, 241
NOAFPR 86
NOALIGN 129
NODECK 209, 210
NOGOFF 37, 62, 72
NOLIST 207
NOOBJECT 209, 210
NOXOBJECT 72
OBJECT 274, 275
OPTABLE 285
PROFILE 55
assembler options (continued)
RA2 103, 155
RENT 212, 285
SECTALGN 113, 194, 201
specifying with PROCESS statements 102
SYSPARM 286
TYPECHECK 103
USING 222
XOBJECT 72, 108, 110, 112, 274, 275
assembler program
basic functions 6
processing sequence 7
relationship to operating system 8
assembler type
returned by SYSATTRA built-in function 380
assembler type value
assigned by EQU instruction 186
associated data file
ADATA instruction 107
contents 6
EXITCTL instruction 187
writing to 107
association of code and data areas 232
asterisks (*)
as location counter reference 37
as multiplication operator 358
as relocatable term 37
defining comment statements 18
attribute reference
notation 325
attributes
assembler
EQU instruction 186
count (K') 335
data 324
defined ( \(D^{\prime}\) ) 337
definition mode 340
in combination with symbols 326
integer (I') 334
length (L') 332
lookahead 340
number (N') 336
of expressions 47, 326
of symbols 326
operation code ( \(\mathrm{O}^{\prime}\) ) 337
program type
EQU instruction 186
reference notation 309
relocatable term 47
scale (S') 333
summary of 409, 415
type (T') 328
XATTR operands 230

\section*{B}

B-type binary constant 141
B2A (SETA built-in function) 353
B2C (SETC built-in function) 375
B2D (SETC built-in function) 376
B2X (SETC built-in function) 376
base register instructions
DROP instruction 172
POP instruction 204
PUSH instruction 209
USING instruction 218
base registers
for absolute addresses 220
BATCH assembler option 341
binary constant (B) 141
binary floating point constant (DB) 167
binary floating point constant (EB) 167
binary floating point constant (LB) 167
binary floating-point constants 167
binary operators 358
binary self-defining term 35
bit patterns
for masks 91
bit-length modifier 137
blank lines
ASPACE instruction 259
in macros 18
in open code 18
books
High Level Assembler xii
branching
conditional assembly 390
AGO instruction 392
AIF instruction 390
extended AIF instruction 391
machine instructions
based 80
extended mnemonics 80
relative 80,83
built-in functions
A2B 374
A2C 375
A2D 375
A2X 375
AND 353, 365
AND NOT 365
arithmetic (SETA) expressions 353
B2A 353
B2C 375
B2D 376
B2X 376
BYTE 376
C2A 353
C2B 377
C2D 377

built-in functions (continued)
    character (SETC) expressions 374
    D2A 354
    D2B 377
    D2C 378
    378
    DCLEN 354
    DCVAL 378
    DEQUOTE 379
    DOUBLE 379
    FIND 355
    function-invocation format 350
    INDEX 355
    introduction 349
    SBIN 365
    SDEC 366
    ISHEX 366
    ISSYM 366
    logical-expression format 350
    LOWER 379
    NOT 356, 366
    OR 356, 366
    OR NOT 367
    SIGNED 380
    SLA 356
    SLL 356
    357
    SRL 357
    summary table 350
    SYSATTRA 380
    SYSATTRP 380
    UPPER 380
    X2A 358
    X2B 381
    X2C 381
    X2D 381
    XOR 358, 367
    XOR NOT 367
BYTE (SETC built-in function) 376
C-type character constant 142
C-type character self-defining terms 36
C2A (SETA built-in function) 353
C2B (SETC built-in function) 377
C2D (SETC built-in function) 377
C2X (SETC built-in function) 377
CATTR instruction 112
CCW instruction 115
CCWO instruction 115
CCW1 instruction 116
CD-ROM publications xiii
CEJECT instruction 118
character (SETC) expressions
built-in functions 374
using 369
character constant 142
DBCS 145
Unicode UTF-16 144
character constant (C) 142
character expressions
evaluation of 382
character external function calls 389
character relations in logical expressions 367, 368
comparing comparands of unequal length 368
character self-defining term 35
character set
code table
standard 421
CODEPAGE option 144
default 11
double-byte 12
standard 11
translation table 14
character strings 309
concatenating 383
defined 371
evaluation of
logical 367
in SETC instruction 369
relational operators for 367
values 383
characters
special
See special characters
class names
assigned by default 60
assigned explicitly 59
CATTR instruction 112
classes
assignment to
by binding and loading properties 59
default names 60
defined by CATTR instruction 112
entry point 60
group of machine language blocks 50
CNOP instruction 119
COBOL communication 57
code areas
association with data areas 232
code table
standard character set 421
CODEPAGE assembler option 144
coding
functions in addition to machine instructions 9
coding aids summary 9
coding conventions
assembler language
alternative format for AGO 393
alternative format for AIF 392
coding conventions (continued) assembler language (continued)
alternative format for LCLx 347
alternative format for SETx 388
alternative statement format for GBLx 345
coding conventions, assembler language
comment statement 18
continuation line errors 16
continuation lines 15
standard coding format 14
statement coding rules 19
COM instruction 57, 121
\&SYSECT 268
\&SYSSTYP 293
combining keyword and positional parameters 255,
303
comment statements
format 18
function of 240
internal macro 261
ordinary 261
comparisons in logical expressions 367
COMPAT assembler option 103, 314
CASE suboption 104
LITTYPE suboption 104
MACROCASE suboption 12, 104, 308
NOLITTYPE suboption 104
NOMACROCASE suboption 104
NOSYSLIST suboption 104
SYSLIST suboption 104, 278, 304, 307, 314, 322, 386
compatibility
default classes 60
object files 60
of HLASM with other languages 4
complex relocatable 47
address constant operands 155
address constants 153
defined 47
EQU instruction 184
expressions 48
computed AGO instruction 393
concatenating character string values 383
concatenation of characters in model statements 249
concatenation operator for strings 383
conditional assembly instructions
ACTR instruction 394
AGO instruction 392
alternative statement format 393
AIF instruction 390
alternative statement format 392
ANOP instruction 395
computed AGO instruction 393
extended AIF instruction 391
function of 256
GBLA instruction 344
alternative statement format 345
conditional assembly instructions (continued)
GBLB instruction 344
alternative statement format 345
GBLC instruction 344
alternative statement format 345
how to write 318
LCLA instruction 345
alternative statement format 347
LCLB instruction 345
alternative statement format 347
LCLC instruction 345
alternative statement format 347
list of 343
MHELP instruction 397
OPSYN assembler instruction 200
effect of 200
redefining 200
SETA instruction 347
alternative statement format 388
SETAF instruction 388
SETB instruction 362
alternative statement format 388
SETC instruction 369
alternative statement format 388
SETCF instruction 389
substring notations in 371
conditional assembly language
See also macro language
CATTR instruction 59, 61
summary 242
summary of expressions 413
constants
address 153, 157
alignment of 129
binary 141
binary floating-point 167
character 142
comparison with literals and self-defining terms 41
decimal 151
duplication factor 132
fixed-point 148
floating-point 161
hexadecimal 164
IEEE binary 169
general information 129
graphic 145
hexadecimal 147
length 129, 160
length attribute value of symbols naming 129
literal
See literal constants
modifiers of 136
nominal values of 140
offset 159
padding of values 130
subfield 1 (duplication factor) 132
constants (continued)
subfield 2 (type) 133
subfield 3 (type extension) 134
subfield 4 (modifier) 136
subfield 5 (nominal value) 140
summary of 407
symbolic addresses of 129
truncation of values 130
type extension 134
types of 126, 133
CONT
suboption of FLAG 105
continuation line errors 247
continuation lines 15
See also alternative statement format
description 15
errors in 16
unlimited number of 16
continuation-indicator field 15
control instructions 79
control sections
concept of 52
defining blank common 57
executable 53
first 54
identifying 123, 212
reference 56
segments 68
unnamed 55
controlling the assembly 6
converting SETA symbol to SETC symbol 387
COPY instruction 122, 260
count attribute ( \(\mathrm{K}^{\prime}\) ) 335
created SET symbols 323
CSECT instruction 123
\&SYSECT 268
\&SYSSTYP 293
interaction with LOCTR instruction 193
Customization book xii
CXD instruction 125

\section*{D}

D' defined attribute 337
D-type floating-point constant 161
D2A (SETA built-in function) 354
D2B (SETC built-in function) 377
D2C (SETC built-in function) 378
D2X (SETC built-in function) 378
data
immediate
See immediate data
PRINT instruction 206
data areas
association with code areas 232
data attributes 324
data definition instructions
CCW instruction 115
CCWO instruction 115
CCW1 instruction 116
DC instruction 126
DS instruction 174
DB-type floating-point constant 167
DBCS
See also double-byte data
PUNCH instruction 208
DBCS assembler option 12, 15, 34, 36, 145, 197, 208, 217, 247, 249, 251, 253, 284, 298, 301, 302, 309, 360, 404, 413
\&SYSOPT_DBCS system variable symbol 284
determining if supplied 284
extended continuation-indicators 17
DC instruction 126
DCLEN (SETA built-in function) 354
DCVAL (SETC built-in function) 378
decimal constant
P-type 151
packed 151
Z-type 151
zoned 151
decimal constant (P) 151
decimal constant (Z) 151
decimal instructions 79
decimal self-defining term 34
DECK assembler option 188, 290
\&SYSPUNCH_DSN system variable symbol 289
\&SYSPUNCH_MEMBER system variable symbol 290
\&SYSPUNCH_VOLUME system variable symbol 290
defaults
class names 60
entry point 60
defined attribute ( \(D^{\prime}\) ) 337
definition mode 340
dependent addressing 67
dependent USING
domain 228
instruction syntax 226
range 228
DEQUOTE (SETC built-in function) 379
DH-type floating-point constant 161
dimensioned SET symbols 344, 346
documentation
High Level Assembler xii, 427
domain
dependent USING instruction 228
labeled USING instruction 226
ordinary USING instruction 223
DOUBLE (SETC built-in function) 379
double-byte character set (DBCS)
See double-byte data
double-byte data
code conversion in the macro language 360
concatenation in SETC expressions 384
concatenation of fields 251
continuation of 15,16
definition of 12
duplication of 369
graphic constants 126, 145
graphic self-defining term 36
in C-type constants 145
in character self-defining terms 35
in comments 18
in G-type constants 145
in keyword operands 302
in macro comments 261
in macro operands 253
in MNOTE operands 197
in positional operands 301
in PUNCH operands 208
in quoted strings 309
in remarks 21
in REPRO operands 210
in TITLE operands 217
listing of macro-generated fields 249
mixed 145
notation xvi
pure 145
DROP instruction 172
DS instruction 174
DSECT instruction 56, 178
\&SYSECT 268
\&SYSSTYP 293
dummy section
external 57
identifying 56, 178
See also external dummy sections
duplication factor
and substrings 369
in character expressions 369
in constants 132
DXD instruction 180
no conflict with other external names 58

\section*{\(E\)}

E-Decks
reading in VSE 4
E-type floating-point constant 161
EB-type floating-point constant 167
edited macros 242
edited macros in VSE 4
EH-type floating-point constant 161
EJECT instruction 181

\section*{Index}
\begin{tabular}{|c|c|c|}
\hline elements of conditional assembly language & 318 & expressions (continued) \\
\hline END instruction 182, 341 nominated entry point 182 & & paired relocatable terms 47 relocatable 48 \\
\hline ENTRY instruction 183 & & rules for coding 45, 363 \\
\hline entry point symbol & & extended AGO instruction 393 \\
\hline referencing using the ENTRY instruction & 183 & extended AIF instruction 391 \\
\hline transfer control to using END instruction 182 & & extended continuation-indicator double-byte data continuation 16 \\
\hline EQU instruction 184 & & listing of macro-generated fields 249 \\
\hline assembler type 184 & & extended SET statement 387 \\
\hline assigning the length attribute 333 & & external dummy sections \\
\hline assigning the type attribute 341 & & CXD instruction to define cumulative length \\
\hline program type 184 & & discussion of 57 \\
\hline equal sign & & DSECT name in Q-type constant 58 \\
\hline designating a literal constant 43 & & DXD instruction to define an 180 \\
\hline ESD entries 72 & & external function calls \\
\hline exclusive OR (XOR) & & arithmetic 388 \\
\hline SETA built-in function 358 & & character 389 \\
\hline EXCP access method 116 & & SETAF instruction 388 \\
\hline EXIT assembler option & & SETCF instruction 389 \\
\hline ADEXIT suboption 188 & & external names \\
\hline INEXIT suboption 188 & & no conflict with DXD instruction 58 \\
\hline LIBEXIT suboption 188 & & external symbol dictionary entries 72 \\
\hline OBJEXIT suboption 188 & & external symbols \\
\hline PRTEXIT suboption 188 & & See also symbols, external \\
\hline TRMEXIT suboption 188 & & ALIAS command 109 \\
\hline exit-control parameters 187 & & in V-type address constant 230 \\
\hline EXITCTL instruction 187 & & length restrictions 110 \\
\hline exiting macros 245 & & providing alternate names 109 \\
\hline exits & & EXTRN instruction 189 \\
\hline See user l/O exits & & \\
\hline EXLITW & & \\
\hline suboption of FLAG 105 & & \\
\hline explicit address & & F-type fixed-point constant 148 \\
\hline specifying 88 & & field boundaries 14 \\
\hline explicit addresses 64 & & FIND (SETA built-in function) 355 \\
\hline explicit length attribute 129 & & first control section 54 \\
\hline exponent modifier & & fixed-point constant (F) 148 \\
\hline floating-point constants 162 & & fixed-point constant (H) 148 \\
\hline specifying 139 & & FLAG assembler option 103, 247 \\
\hline expressions & & CONT suboption 16, 247 \\
\hline absolute 47 & & nnn suboption 198 \\
\hline arithmetic 347 & & NOSUBSTR suboption 373 \\
\hline attributes 47 & & floating point constant (D) 161 \\
\hline character 369 & & floating point constant (DH) 161 \\
\hline See also character expressions & & floating point constant (E) 161 \\
\hline complex relocatable 48 & & floating point constant (EH) 161 \\
\hline conditional assembly & & floating point constant (L) 161 \\
\hline summary of 413 & & floating point constant (LH) 161 \\
\hline discussion of 44 & & floating point constant (LQ) 161 \\
\hline EQU instruction 184 & & floating point constants \\
\hline evaluation of 367 & & binary \\
\hline character 382 & & See binary floating-point constants \\
\hline multiterm 46 & & floating-point constants 161 \\
\hline single-term 46 & & floating-point instructions 79 \\
\hline
\end{tabular}

FOLD assembler option 14
format notation, description xiv-xvi
format-0 channel command word 115
format-1 channel command word 116
FORTRAN communication 57
function calls
See external function calls
function-invocation format built-in function 350
functions of conditional assembly language 318

\section*{G}

G-type graphic constant 145
GBLA instruction 344
alternative statement format 345
GBLB instruction 344
alternative statement format 345
GBLC instruction 344
alternative statement format 345
GEN
PRINT instruction 205
General Information book xii
general instructions 78
generated fields
listing 248
generating END statements 341
global-scope system variable symbols 262
GOFF assembler option \(72,108,124,125,160,216\), 274, 275
affect on RI-format instructions 92
CATTR instruction 112
entry point 111
location counter maximum value 37, 62
program object 50
sections 68
XATTR instruction 230
GOFF option
interaction with PUNCH instruction 209
interaction with REPRO instruction 210
graphic constant (G) 145
graphic self-defining term 36

\section*{H}

H-type fixed-point constant 148
header
macro definition 244
hexadecimal constant (X) 147
hexadecimal self-defining term 35
High Level Assembler
publications xii

\section*{I}

I' integer attribute 334

ICTL instruction 189
identification-sequence field 15
immediate data 91
in machine instructions 91
IMPLEN suboption of FLAG 105
implicit address specifying 88
implicit addresses 64
implicit length attribute 129
implied addresses
See implicit addresses
INDEX (SETA built-in function) 355
information retrieval functions
SYSATTRA 380
SYSATTRP 380
inner macro definitions 313
inner macro instructions 256, 312
passing sublists to 307
input format control statement
See ICTL instruction
input stream 10
input/output operations 80
installation and customization
book information xii
instruction statement format 19
instructions
\&SYSOPT_OPTABLE system variable symbol 285
assembler
ACONTROL 103
ADATA 107
ALIAS 109
AMODE 110
CATTR 112
CCW 115
CCWO 115
CCW1 116
CEJECT 118
CNOP 119
COM 121
COPY 122
CSECT 123
CXD 125
DC 126
DROP 172
DS 174
DSECT 178
DXD 180
EJECT 181
END 182
ENTRY 183
EQU 184
EXTRN 189
ICTL 189
ISEQ 190
LOCTR 191
LTORG 193

\section*{Index}
```

instructions (continued)
assembler (continued)
OPSYN }19
ORG }20
POP }20
PRINT }20
PUNCH 208
PUSH }20
REPRO 210
RMODE 211
RSECT }21
SPACE 213
START }21
TITLE }21
USING 218
WXTRN }22
XATTR }23
conditional assembly
ACTR }39
AGO }39
AIF 390
ANOP 395
GBLA 344
GBLB }34
GBLC 344
LCLA }34
LCLB 345
LCLC 345
SETA }34
SETAF 388
SETB }36
SETC }36
SETCF }38
machine
examples }9
OPTABLE option 285
macro
AEJECT 257
AINSERT 108,257
ASPACE }25
COPY 260
MEXIT }26
MHELP }39
MNOTE }19
integer attribute (I') }33
internal macro comment statement format 18
internal macro comment statements }26
ISBIN (SETB built-in function) }36
ISDEC (SETB built-in function) 366
ISEQ instruction }19
ISHEX (SETB built-in function) }36
ISSYM (SETB built-in function) }36

| length fields in machine instructions 90 | location counter reference |
| :---: | :---: |
| length modifier | effect of duplication factor in constants 133 |
| constant 136 | effect of duplication factor in literals 133 |
| syntax 136 | relocatable term 37, 44 |
| length of control section 62 | location counter setting 61 |
| LH-type floating-point constant 161 | LOCTR instruction 191 |
| LIBMAC assembler option 103, 241 | ORG instruction 200 |
| library macro definitions 241 | START instruction 61 |
| license inquiry 425 | THREAD option 62 |
| Licensed Program Specifications xiii | LOCTR instruction 191 |
| lines | \&SYSLOC 278 |
| how to continue 15 | interaction with CSECT instruction 193 |
| LINKAGE | logical (SETB) expressions 362, 365 |
| XATTR operands 231 | logical functions |
| linkages | AND 353 |
| by means of the ENTRY instruction 183 | NOT 356 |
| by means of the EXTRN instruction 189 | OR 356 |
| by means of the WXTRN instruction 229 | XOR 358 |
| symbolic 68 | logical operators |
| linking 51 | AND 365 |
| LIST(121) option | AND NOT 365 |
| displaying location counter 37 | NOT 366 |
| LIST(133) option | OR 366 |
| displaying location counter 37 | OR NOT 367 |
| listing | XOR 358, 367 |
| generated fields 248 | XOR NOT 367 |
| listing control instructions | logical XOR 358 |
| AEJECT instruction 257 | logical-expression format |
| AINSERT instruction 108, 257 | defined 365 |
| ASPACE instruction 259 | logical-expression format built-in function 350 |
| CEJECT instruction 118 | lookahead mode 340 |
| EJECT instruction 181 | LookAt message retrieval tool xiii |
| PRINT instruction 204 | LOWER (SETC built-in function) 379 |
| SPACE instruction 213 | LQ-type floating-point constant 161 |
| TITLE instruction 215 | LTORG instruction 193 |
| literal constants |  |
| coding 43 | M |
| definition 171 | M |
| literal pool 44, 68, 194 | machine instruction formats |
| alignment 194 | RI format 91 |
| conditions of creation 194 | RR format 93 |
| LTORG instruction 193 | RS format 94 |
| literals | RSI format 95 |
| comparison with constants and self-defining | RX format 95 |
| terms 41 | SI format 97 |
| duplicate 195 | SS format 97 |
| explanation of 40 | machine instruction statements |
| general rules for usage 42 | addresses 87 |
| type attribute 331 | control 79 |
| local scope system variable symbols 262 | decimal 79 |
| location counter | floating-point 79 |
| defined 36 | formats 83 |
| for control sections 62 | general 78 |
| maximum value 37 | immediate data 91 |
| effect of GOFF assembler option 37 | input/output 80 |
| effect of NOGOFF assembler option 37 | length field in 90 |

length fields in machine instructions 90 gh modirer
constant 136
length of control section 62
LH-type floating-point constant 161
AC assembler option 103, 24
license inquiry 425
Licensed Program Specifications xiii es
how to continue 15
LINKAGE
XATTR operands 231
linkages
by means of the ENTRY instruction 183
by means of the EXTRN instruction 189
229
linking 51
LIST(121) option
displaying location counter 37
(133) option n counter 37
generated fields 248
ting control instructions
AINSERT instruction 108, 257
ASPACE instruction 259
CEJECT instruction 118
EJECT instruction 181
PRINT instruction 204
SPACE instruction 213
TITLE instruction 215
literal constants
coding 43
definition 171
, pool 44, 68, 104
conditions of creation 194
LTORG instruction 193
comparison with constants and self-defining
terms 41
duplicate 195
explanation of 40
general rules for usage 42
type attribute 331
local scope system variable symbols 262
location counter
defined 36
for control sections 62
effect of GOFF assembler option 37
effect of NOGOFF assembler option 37
location counter reference
effect of duplication factor in constants 133 effect of duplication factor in literals 133
relocatable term 37, 44
location counter setting 61
ORG instruction 200
START instruction 61
THREAD option 62
OCTR instruction 191
\&SSLOC 278
ogical (SETB) expressions 362, 365
ogical functions

XOR 358
ogical operators

AND NOT 365
NOT 366
OR 366
OR NOT 367
XOR 358, 367
XOR NOT 367
ogical XOR 358
ical-expression format
ogical-expression format built-in function 350
lookahead mode 340
LookAt message retrieval tool xiii
ER (SETC bulth-in function) 379

LTORG instruction 193
machine instruction formats
RI format 91
RR format 93

RSI format 95
RX format 95
SI format 97
addresses 87
control 79
decimal 79
79
formats 83
immediate data 91
input/output 80
length field in 90

## Index

```
machine instruction statements (continued)
    operand entries 85
    symbolic operations codes in 84
machine instructions 24
machine instructions, publications 427
macro comment statement format 18
macro definition header (MACRO) 244
macro definition trailer (MEND) 244
macro definitions
    alternative statement format 246
    arguments 300
    body of a 247
    combining positional and keyword parameters 255
    comment statements 261
    COPY instruction 260
    description 238
    format of 244
    header 239, 244
    how to specify 243
    inner macro instructions 256
    internal macro comment statements 261
    keyword parameters 255
    MEXIT instruction 260
    MNOTE instruction 196
    model statements 239
    name field parameter 245
    nesting in 311
    operand entry 300
    parameters 245
    parts of a macro definition 239
    positional parameters 254
    prototype 239
    sequence symbols 339
    subscripted symbolic parameters 255
    symbolic parameters 253
    trailer 239, 244
    where to define in a source module 243
    where to define in open code 243
macro instruction statements 297
macro instructions
    See also inner macro instructions
    See also outer macro instructions
    alternative statement format 298
    arguments 300
    description 241, 297
    format of 297
    general rules and restrictions 313
    inner and outer 311
    M type attribute 329
    multilevel sublists 306
    name entry 299
    name field type attribute 329
    operand entry 300
    operation entry 299
    passing sublists to inner 307
    passing values through nesting levels 314
machine instruction statements (continued)
```

macro instructions (continued)
prototype statement 244
sequence symbols 339
sublists in operands 304
summary of 405
values in operands 307
macro language
comment statements 240
conditional assembly language 242
defining macros 238
library macro definition 241
macro instruction 241
model statements 239
processing statements 240
source macro definition 241
summary of 409
using 238
macro library 242
MACRO statement (header) 244
macros
continuation line errors 247
edited macros 242
effects of LIBMAC option 241
exiting 245
format of a macro definition 244
how to specify 243
library macro definition 241
macro definition 238
macro definition header (MACRO) 239, 244
macro definition trailer (MEND) 239, 244
macro instruction 241
macro library 242
macro prototype statement 239
MACRO statement (header) 244
MEND statement (trailer) 245
MEXIT instruction 260
MNOTE instruction 196
model statements 239
source macro definition 241
using macros 238
manuals
High Level Assembler xii, 427
masks
specifying their bit pattern using immediate data 91
MCALL
PRINT instruction 206
message retrieval tool, LookAt xiii
MEXIT instruction 260
MHELP instruction 397
mnemonic codes
extended 80
machine instruction 84
MNOTE instruction 196
model statements
explanation of 247
function of 239
model statements (continued)
in macros 239, 247
in open code 248
rules for concatenation of characters in 249
rules for specifying fields in 251
summary of 405
variable symbols as points of substitution in 247
modifiers of constants
exponent 139
length 136
scale 138
MSOURCE
PRINT instruction 206
multilevel sublists 306
MVS publications 427

## N

N' number attribute 336
name entry coding 19
name field parameter
in macro definition 245
nesting
macro calls 312
macro definition 311
passing values through nesting levels 314
recursion 311
system variable symbols in nested macros 316
nesting macro instructions
in calls 312
in definitions 313
NOAFPR assembler option 86
NOALIGN
suboption of FLAG 105
NOALIGN assembler option 129
NOCOMPAT assembler option 104
NOCONT
suboption of FLAG 105
NODATA
PRINT instruction 206
NODECK assembler option 209, 210
NOEXLITW
suboption of FLAG 105
NOGEN
PRINT instruction 205
NOGOFF assembler option 37, 62, 72
NOIMPLEN
suboption of FLAG 105
NOLIST assembler option 207
NOMCALL
PRINT instruction 206
nominal values of constants and literals
address 153
binary 141
binary floating-point 167
character 142
nominal values of constants and literals (continued) decimal 151
fixed-point 148
floating-point 161
graphic 145
hexadecimal 147
NOMSOURCE
PRINT instruction 207
NOOBJECT assembler option 209, 210
NOPAGEO
suboption of FLAG 106
NOPRINT
PRINT instruction 207
PUSH instruction 210
NOPRINT operand
AREAD instruction 258
POP instruction 204
PRINT instruction 207
PUSH instruction 209
NOSUBSTR
suboption of FLAG 106
NOT (SETA built-in function) 356
NOT (SETB built-in function) 366
notation, description xiv-xvi
NOUHEAD
PRINT instruction 207
NOUSINGO
suboption of FLAG 106
NOXOBJECT assembler option 72
number attribute ( $\mathrm{N}^{\prime}$ ) 336

## 0

O' operation code attribute 337
OBJECT assembler option 188, 274, 275
object external class name establishing 112
object program structure
load module model 50
program object model 50
OFF
PRINT instruction 205
offset constant (Q) 159
omitted operands 307, 308
ON
PRINT instruction 205
online publications xiii
open code 244, 342
blank lines within 18
conditional assembly instructions in 342
defined 244
operand entries 20
operand entry 300
operands
assembler instruction statements 85
combining positional and keyword 303

## Index

operands (continued)
compatibility with earlier assemblers 308
in machine instructions 85
keyword 301
machine instruction statements 85
multilevel sublists in 306
omitted 307, 308
positional 300
special characters in 308
statement coding rules 20
sublists in 304
unquoted operands 308
values in 307
operating system
relationship to assembler program 8
operation code attribute ( $\mathrm{O}^{\prime}$ ) 337
operation codes, symbolic
extended 80
machine instruction 84
operation entry coding 19
operator
binary
See binary operators
unary
See unary operators
OPSYN instruction 198
OPTABLE assembler option 285
\&SYSOPT_OPTABLE system variable symbol 285
determining value 285
OR (SETA built-in function) 356
OR (SETB built-in function) 366
OR NOT(SETB built-in function) 367
ordinary comment statements 261
ordinary symbols
defined 30
ordinary USING instruction
difference from labeled using instruction 224
ORG instruction 200
location counter setting 200
organization of this manual xi
outer macro definitions 313
outer macro instructions 312
OVERRIDE 102

## P

P-type decimal constant 151
packed decimal constant 151
PAGEO
suboption of FLAG 106
paired relocatable terms 47
pairing rules
'1 335
\&\& 335
character self-defining terms 35
DC instruction 144
parameters
combining positional and keyword 255
keyword 255
positional 254
subscripted symbolic 255
symbolic 253
parentheses
enclosing terms 28
PL/I communication 57
pool
literal
See literal pool
POP instruction 204
positional parameters 254, 300
predefined absolute symbols
in logical expressions 363
in SETA expressions 359
legal use 363
not permitted in character expressions 383
previously defined symbols 33
PRINT
PUSH instruction 209
PRINT instruction 204
DATA 206
GEN 205
MCALL 206
MSOURCE 206
NODATA 206
NOGEN 205
NOMCALL 206
NOMSOURCE 207
NOPRINT 207
NOUHEAD 207
OFF 205
ON 205
UHEAD 207
private code 54, 187
private control section
See private code
privileged instructions 79
process
override 102
PROCESS statement
See *PROCESS statement
processing of statements conditional assembly instructions 256
COPY instruction 260
inner macro instructions 256
MEXIT instruction 260
MNOTE instruction 196
PROFILE assembler option 55
program control instructions
CNOP instruction 119
COPY instruction 122
END instruction 182
ICTL instruction 189

| program control instructions (continued) | publications (continued) |
| :---: | :---: |
| ISEQ instruction 190 | VSE 428 |
| LTORG instruction 193 | PUNCH instruction 208 |
| ORG instruction 200 | DBCS 208 |
| POP instruction 204 | PUSH instruction 209 |
| PUNCH instruction 208 | ACONTROL 209 |
| PUSH instruction 209 | NOPRINT 210 |
| REPRO instruction 210 | PRINT 209 |
| program object model | USING 209 |
| element 50 |  |
| part 50 | Q |
| section 50 |  |
| program sectioning 51 | Q-type offset constant 159 |
| program sectioning and linking instructions | qualified addressing 66 |
| AMODE instruction 110 | labeled USING instructions 66 |
| CATTR instruction 112 | qualified symbols 223 |
| COM instruction 57, 121 | composition 66 |
| CSECT instruction 123 | labeled USING 223 |
| CXD instruction 125 | qualifiers |
| DSECT instruction 56, 178 | for symbols 223 |
| DXD instruction 180 | relocatable 224 |
| ENTRY instruction 183 | quotation marks 371 |
| EXTRN instruction 189 | as delimiter for character string in SETC 371 |
| LOCTR instruction 191 | as string terminator PUNCH instruction 208 |
| RMODE instruction 211 | DBCS apostrophe not delimiter 197 |
| RSECT instruction 212 | not recognized in double-byte data 145 |
| WXTRN instruction 229 | paired in MNOTE message 197 |
| XATTR instruction 230 | paired in PUNCH instruction 208 |
| program type | paired in Unicode data 144 |
| assigned by modifier in DC instruction 135 | representation in character constant 143 |
| program type value assigned by EQU instruction 186 | quoted strings 309 |
| Programmer's Guide xiii | R |
| Prototype Control Section | R-type address constant 132, 160, 232 |
| See PSECT | RA2 assembler option 103, 155 |
| alternative statement format 246, 298 | railroad track format, how to read xiv-xvi range |
| function of 244 | dependent USING instruction 228 |
| name field 245 | labeled USING instruction 225 |
| operand field 246 | ordinary USING instruction 222 |
| operation field 245 | reading edited macros in VSE 4 |
| summary of 405 | redefining conditional assembly instructions 200 |
| PSECT 232 | REFERENCE |
| address 132, 160 | XATTR operands 231 |
| discussed 232 | reference constant (R) 160 |
| XATTR operands 231 | reference notation for attribute 309 |
| pseudo-registers 57 | register zero |
| publications xii | as base address 222 |
| High Level Assembler xii, 427 | in USING instruction 220 |
| HLASM Toolkit 427 | registers |
| machine instructions 427 | use by machine instructions 86 |
| MVS 427 | relational operators |
| online (CD-ROM) xiii | for character strings 367 |
| organization of this manual xi | relative address |
| VM 428 | specifying 88 |

## Index

relative addressing 67
relocatability attribute 47
relocatable expression
complex 48
definition 48
EQU instruction 184
relocatable symbol
defined 31
relocatable terms 28
See also self-defining terms
remarks entries 21
RENT assembler option 212, 285
\&SYSOPT_RENT system variable symbol 285
determining if supplied 285
representation conversion functions
A2 375
A2B 374
A2D 375
A2X 375
B2A 353
B2C 375
B2D 376
B2X 376
BYTE 376
C2A 353
C2B 377
C2D 377
C2X 377
D2A 354
D2B 377
D2C 378
D2X 378
SIGNED 380
X2A 358
X2B 381
X2C 381
X2D 381
REPRO instruction 210
residence mode (RMODE)
See RMODE
RI format 91
RMODE
establishing values 68
indicators in ESD 68
instruction 211
RMODE instruction
24211
31211
64211
ANY 211
RR format 93
RS format 94
RSECT instruction 212
\&SYSECT 268
\&SYSSTYP 293

RSI format 95
rules for model statement fields 251
RX format 95

## S

S' scale attribute 333
S-type address constant 156
scale attribute (S') 333
scale modifier 136, 138
SCOPE
XATTR operands 232
scope of SET symbols 319
SECTALGN assembler option
interaction with ALIGN 113
interaction with LTORG instruction 194
interaction with ORG instruction 201
sectioning, program
addressing mode of a control section 110
control sections 52
CSECT instruction 123
defining 180
ESD entries 72
external symbols 189
first control section 54
identifying a blank common control section 57
identifying a dummy section 56
location counter 61
maximum length of control section 62
multiple location counters in a control section 191
read-only control section 212
residence mode of a control section 211
source module 52
total length of external dummy sections 125
unnamed control section 55
weak external symbols 229
sections
See also control sections
GOFF option considerations 68
segments of control sections 68
self-defining terms
binary 35
character 35
comparison with literals and constants 41
decimal 34
graphic 36
hexadecimal 35
overview 34
using 34
semiprivileged instructions 79
sequence symbols 339
defined 31
SET statement
extended 387
SET symbols
See also symbols, SET

```
SET symbols (continued)
    arrays 319
    assigning values to }34
    created }32
    declaring 343
        global 344
        local 345
    description of }31
    dimensioned }31
    external function calls 388,389
    scope of }31
    SETA (set arithmetic) }34
    SETB (set binary) }36
    SETC (set character) }36
    specifications 320
    specifications for subscripted }32
    subscripted 319
SETA
    arithmetic expression 347
    built-in functions 353
    instruction format 347
    symbol in operand field of SETC
        in arithmetic expressions }35
        leading zeros }38
        sign of substituted value }38
    symbols
        subscripted 347
        using 360
SETAF instruction 388
SETB
    character relations in logical expressions 367,368
    instruction format }36
    logical expression 362
    symbols
        subscripted 362
        using 368
SETC
    built-in functions 374
    character expression }36
    character expressions 371
    instruction format 369
    SETA symbol in operand field 387
    statement format }36
    substring notation 369
    symbols
        subscripted }36
SETCF instruction 389
shift codes
    shift-in (SI) DBCS character delimiter 12
    shift-out (SO) DBCS character delimiter 12
shift functions
    SLA 356
    SLL }35
    SRA }35
    SRL }35
```

shift left arithmetic (SETA built-in function) 356
shift left logical (SETA built-in function) 356
shift right arithmetic (SETA built-in function) 357
shift right logical (SETA built-in function) 357
SI (shift-in) character
continuation of double-byte data 16
continuation-indicator field 15
double-byte character set 12
SI format 97
SIGNED (SETC built-in function) 380
simply relocatable address
defined 89
SLA (SETA built-in function) 356
SLL (SETA built-in function) 356
SO (shift-out) character
continuation of double-byte data 16
continuation-indicator field 15
double-byte character set 12
softcopy publications xiii
source macro definitions 241
source module 10
SPACE instruction 213
blank lines 213
special characters 308
SRA (SETA built-in function) 357
SRL (SETA built-in function) 357
SS format 97
stacked items xv
START instruction 214
\&SYSECT 268
\&SYSSTYP 293
beginning a source module 52
control section 53
syntax 214
statement coding rules 19
statement field 14
string manipulation functions
DCLEN 354
DCVAL 378
DEQUOTE 379
DOUBLE 379
LOWER 379
UPPER 380
string scanning functions
FIND 355
INDEX 355
strings
character 309
quoted 309
structure of assembler language 21
subfield 1 of constant (duplication factor) 132
subfield 2 of constant (type) 133
subfield 3 of constant (type extension) 134
subfield 4 of constant (modifier) 136
subfield 5 of constant (nominal value) 140

## Index

sublists
compatibility with Assembler H 278
effect of COMPAT(SYSLIST) assembler option 304, 307
in operands 304
multilevel 306
passing
to inner macro instructions 307
subscripted local SET symbol 347
subscripted SET symbols 319, 322
See also dimensioned SET symbols
subscripted symbolic parameters 255
SUBSTR
suboption of FLAG 106
substring notation
arithmetic expressions in 352
assigning SETC symbols 369,389
concatenating double-byte data 384
concatenation 386
definition 371
duplicating double-byte data 370
duplication factor 369
evaluation of 372
level of parentheses 359
using count ( $\mathrm{K}^{\prime}$ ) attribute 325
symbol definition (EQU) instruction 184
symbol length attribute reference 38
symbol qualifier
label unavailable as result of DROP instruction 172
symbol qualifiers 223
symbol table 30
symbolic linkages 68
symbolic operation codes 84
defining 198
deleting 198
OPSYN 198
symbolic parameters 253
symbols
absolute 184
See also absolute symbol
attributes in combination with 326
complexly relocatable
EQU instruction 185
defining 31
explanation of 29
extended SET 387
external 157, 229
EXTRN instruction 189
labeled USING 223
length attribute reference 38
ordinary
See ordinary symbols
previously defined 33
qualifiers 223
relocatable 31
See also relocatable symbol
symbols (continued)
restrictions on 32
sequence 339
See also sequence symbols
SET 347
declaring global 343
declaring local 345
USING instruction labels 223
variable
See also variable symbols
as points of substitution in model statements 247
SET symbols 319
subscripted 319
symbolic parameters 253
weak 229
syntax notation, description xiv-xvi
SYSADATA file
ADATA instruction 107
SYSATTRA (SETC built-in function) 186, 380
SYSATTRP (SETC built-in function) 186, 380
SYSPARM assembler option 286
\&SYSPARM system variable symbol 286
system macro instructions 241
system variable symbols
\&SYS naming convention 262
\&SYSADATA_DSN 263
\&SYSADATA_MEMBER 264
\&SYSADATA_VOLUME 265
\&SYSASM 265
\&SYSCLOCK 266
\&SYSDATC 266
\&SYSDATE 267
\&SYSECT 267
\&SYSIN_DSN 269
\&SYSIN_MEMBER 270
\&SYSIN_VOLUME 271
\&SYSJOB 272
\&SYSLIB_DSN 272
\&SYSLIB_MEMBER 273
\&SYSLIB_VOLUME 273
\&SYSLIN_DSN 274
\&SYSLIN_MEMBER 275
\&SYSLIN_VOLUME 275
\&SYSLIST 276
\&SYSLOC 278
\&SYSM_HSEV 196, 279
\&SYSM_SEV 196, 280
\&SYSMAC 279
\&SYSNDX 281
\&SYSNEST 283
\&SYSOPT_DBCS 284
\&SYSOPT_OPTABLE 285
\&SYSOPT_RENT 285
\&SYSOPT_XOBJECT 285
\&SYSPARM 286

| system variable symbols (continued) |  | Unicode UTF-16 constant nominal data |
| :---: | :---: | :---: |
| \&SYSPRINT_DSN 286 |  | CODEPAGE option 144 |
| \&SYSPRINT_MEMBER 288 |  | unknown type attribute 330 |
| \&SYSPRINT_VOLUME 288 |  | unnamed control section 55 |
| \&SYSPUNCH_DSN 289 |  | unquoted operands 308 |
| \&SYSPUNCH_MEMBER 290 |  | unsigned integer conversion 387 |
| \&SYSPUNCH_VOLUME 290 |  | UPPER (SETC built-in function) 380 |
| \&SYSSEQF 291 |  | user I/O exits 187 |
| \&SYSSTEP 292 |  | user records |
| \&SYSSTMT 292 |  | ADATA instruction 107 |
| \&SYSSTYP 292 |  | USING |
| \&SYSTEM_ID 293 |  | PUSH instruction 209 |
| \&SYSTERM_DSN 294 |  | USING assembler option |
| \&SYSTERM_MEMBER 295 |  | WARN suboption 222 |
| \&SYSTERM_VOLUME 295 |  | USING instruction 218 |
| \&SYSTIME 296 |  | base registers for absolute addresses |
| \&SYSVER 296 |  | dependent 226 |
| defined 262 |  | discussion of 218 |
| in nested macros 316 |  | domain of a 223 |
| in open code 262, 316, 343 |  | for executable control sections 219 |
| summary of 416 |  | for reference control sections 219 |
| variability 262 |  | labeled 223 |
|  |  | labeled dependent 226 |
| T |  | range |
|  |  | dependent 228 |
| T' type attribute |  | labeled 225 |
| See type attribute |  | ordinary 222 |
| terms 28 enclosed in parentheses 28 |  | USINGO <br> suboption of FLAG 106 |
| TITLE instruction 215 |  | Suboption of FLAG 106 |
| Toolkit Customization book xiii trailer |  | UTF-16 Unicode character constant 135 |
| macro definition 244 |  |  |
| TRANSLATE option |  | V |
| converting default EBCDIC characters | 142 | V-type address constant 157, 232 |
| not modifying ASCII 144 |  | validity checking functions |
| translation table 14 |  | ISBIN 365 |
| type attribute ( $\mathrm{T}^{\prime}$ ) 328 |  | ISDEC 366 |
| literals 330, 331 |  | ISHEX 366 |
| name field of macro instruction 329 |  | ISSYM 366 |
| undefined type attribute 330 |  | values |
| unknown type attribute 330 |  | passing through nesting levels 314 |
| type extension of constants 134 |  | values in operands 307 |
| TYPECHECK assembler option 103 |  | variable symbols |
| types of constants 133 |  | See also SET symbols |
|  |  | See also symbols |
| U |  | \&SYSADATA_DSN 263 |
|  |  | \&SYSADATA_MEMBER 264 |
| UHEAD |  | \&SYSADATA_VOLUME 265 |
| PRINT instruction 207 |  | \&SYSASM 265 |
| unary operators 358 |  | \&SYSCLOCK 266 |
| undefined type attribute 330 |  | \&SYSDATC 266 |
| Unicode character constant 142 |  | \&SYSDATE 267 |
| Unicode UTF-16 |  | \&SYSECT 267 |
| character constant 144 |  | \&SYSIN_DSN 269 |
|  |  | \&SYSIN_MEMBER 270 |

system variable symbols (continued)
SSYSPRINT DSN 286
\&SYSPRINT_VOLUME 288
\&SYSPUNCH_DSN 289
促
\&SYSSEQF 291
\&SYSSTEP 292
\&SYSSTMT 292
\&SYSSTYP 292
SYSTEMID 293
\&SYSTERM_MEMBER 295
\&SYSTERM_VOLUME 295
\&SYSTIME 296
\&SYVER 2 g
in nested macros 316
in open code 262, 316, 343
summary of 416
variability 262

T' type attribute
See type attribute
rms 28
enclosed in parentheses 28
ITLE instruction 215
tookit Customization book xili
macro definition 244
TRANSLATE option
converting default EBCDIC characters 142
not modifying ASCII 144
translation table 14
type attribute (T') 328
literals 330, 331
name field of macro instruction 329
undefined type attribute 330
unknown type attribute 330
type extension of constants 134
TYPECHECK assembler option 103
types of constants 133

UHEAD
PRINT instruction 207
unary operators 358
undefined type attribute 330
Unicode character constant 142
icode UTF-16
character constant 144

Unicode UTF-16 constant nominal data CODEPAGE option 144
unknown type attribute 330
unnamed control section 55
unquoted operands 308
387
UPPER(SETC bultin function) 380
user records
ADATA instruction 107
PUSH instruction 209
USING assembler option
WARN suboption 222
USING instruction 218
base registers for absolute addresses 220
226
domain of a 223
for executable control sections 219
for reference control sections 219
223
range
dependent 228
labeled 225
ordinary 222
USINGO
suboption of FLAG 106
UTF-16 145
UTF-16 Unicode character constant 135

V-type address constant 157, 232
ISBIN 365
ISDEC 366
SHEX 366
SSYM 366
passing through nesting levels 314
values in operands 307
variable symbols
See also SET symbols
also symbols
\&SYSADATA_MEMBER 264
\&SYSADATA_VOLUME 265
\&SYSASM 265
SYSCLOCK 266

SSYSECT 207
\&SYSIN_DSN 269
\&SYSIN_MEMBER 270

## Index

| variable symbols (continued) |
| :---: |
| \&SYSIN_VOLUME 271 |
| \&SYSJOB 272 |
| \&SYSLIB_DSN 272 |
| \&SYSLIB_MEMBER 273 |
| \&SYSLIB_VOLUME 273 |
| \&SYSLIN_DSN 274 |
| \&SYSLIN_MEMBER 275 |
| \&SYSLIN_VOLUME 275 |
| \&SYSLIST 276 |
| \&SYSLOC 278 |
| \&SYSM_HSEV 279 |
| \&SYSM_SEV 280 |
| \&SYSMAC 279 |
| \&SYSNDX 281 |
| \&SYSNEST 283 |
| \&SYSOPT_DBCS 284 |
| \&SYSOPT_OPTABLE 285 |
| \&SYSOPT_RENT 285 |
| \&SYSOPT_XOBJECT 285 |
| \&SYSPARM 286 |
| \&SYSPRINT_DSN 286 |
| \&SYSPRINT_MEMBER 288 |
| \&SYSPRINT_VOLUME 288 |
| \&SYSPUNCH_DSN 289 |
| \&SYSPUNCH_MEMBER 290 |
| \&SYSPUNCH_VOLUME 290 |
| \&SYSSEQF 291 |
| \&SYSSTEP 292 |
| \&SYSSTMT 292 |
| \&SYSSTYP 292 |
| \&SYSTEM_ID 293 |
| \&SYSTERM_DSN 294 |
| \&SYSTERM_MEMBER 295 |
| \&SYSTERM_VOLUME 295 |
| \&SYSTIME 296 |
| \&SYSVER 296 |
| attributes 326, 413 |
| defined 30 |
| dimensioned 322 |
| implicitly declared 320, 322 |
| SETA 347 |
| SETB 362 |
| SETC 369 |
| subscripted 322 |
| summary of 415 |
| symbolic parameters 246 |
| See system variable symbols |
| type 320 |
| VM publications 428 |
| VSE publications 428 |

## w

web site xiii
WXTRN instruction 229

## X

X-type hexadecimal constant 147
X2A (SETA built-in function) 358
X2B (SETC built-in function) 381
X2C (SETC built-in function) 381
X2D (SETC built-in function) 381
XATTR instruction 230
XATTR operands
ATTRIBUTES 230
LINKAGE 231
PSECT 231
REFERENCE 231
SCOPE 232
XOBJECT assembler option $72,108,188,274,275$
See also GOFF assembler option
ALIAS string 110
CATTR instruction 112
XATTR instruction 230
XOR (SETA built-in function) 358
XOR (SETB built-in function) 367
XOR NOT (SETB built-in function) 367

## Y

Y-type address constant 153

## Z

Z-type decimal constant 151
z/OS publications
zero
division by, in expression 46
zero, register
See register zero
zoned decimal constant
See Z-type decimal constant

## We'd Like to Hear from You

High Level Assembler for MVS \& VM \& VSE
Language Reference
Release 5
Publication No. SC26-4940-04

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.
- Electronic mail—Use this Internet ID:
- Internet: comments@us.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 Language Reference

## Release 5

Publication No. SC26-4940-04
How satisfied are you with the information in this book?

|  | Very <br> Satisfied | Satisfied | Neutral | Dissatisfied | Very <br> Dissatisfied |
| :--- | :---: | :---: | :---: | :---: | :---: |
| Technically accurate | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Complete | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Easy to find | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Easy to understand | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Well organized | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Applicable to your tasks | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Grammatically correct and consistent | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Graphically well designed | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| Overall satisfaction | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |

Please tell us how we can improve this book:

May we contact you to discuss your comments? $\square$ Yes $\square$ No

| $\dagger$ | NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES |
| :---: | :---: |
| BUSINESS REPLY MA\|L |  |
| FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK |  |
| POSTAGE WILL BE PAID BY ADDRESSEE |  |
| IBM Corporation J87/D325 555 Bailey Avenue SAN JOSE, CA 95141-9989 |  |

## 

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 HLASM Programmer's Guide,
GC26-4943 HLASM General Information
GC26-4944 HLASM Licensed Program Specifications.
SC26-4940 HLASM Lanquage Reference
SC26-3494 HLASM Installation and Customization Guide
High Level Assembler Toolkit Feature Publications
GC26-8709 HLASM Toolkit Feature Interactive Debug Facility User's Guide.
GC26-8710 HLASM Toolkit Feature User's Guide
GC26-8711 HLASM Toolkit Feature Installation and Customization Guide.
GC26-8712 HLASM Toolkit Feature Debug Reference Summary


