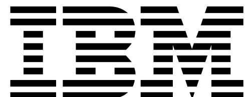
IBM Enterprise Metal C for z/OS, V3.1

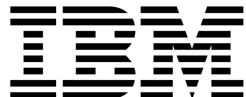


User's Guide

Version 3 Release 1

SC27-9051-00

IBM Enterprise Metal C for z/OS, V3.1



User's Guide

Version 3 Release 1

SC27-9051-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 273.

This edition applies to Version 3 Release 1 of IBM Enterprise Metal C for z/OS (5655-MCE) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: June 12, 2018

© Copyright IBM Corporation 2018.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document vii
Where to find more information x
z/OS Basic Skills in IBM Knowledge Center . . . x
Technical support x
How to send your comments to IBM xi
If you have a technical problem xi

Chapter 1. About IBM Enterprise Metal C for z/OS 1
The Enterprise Metal C for z/OS compiler 2
The C language 2
Enterprise Metal C for z/OS compiler features . . 3
metalc utility 4
About assembling, linking, and binding 4
File format considerations 4
z/OS UNIX System Services 4
Additional features of Enterprise Metal C for z/OS . 5

Chapter 2. Compiler options 7
Specifying compiler options 7
IPA considerations 7
Using special characters 8
Specifying compiler options using #pragma options 9
Specifying compiler options under z/OS UNIX 10
Compiler option defaults 10
Summary of compiler options 11
Compiler output options 14
Compiler input options 15
Language element control options 15
Object code control options 16
Floating-point and integer control options 17
Error-checking and debugging options 17
Listings, messages, and compiler information options 18
Optimization and tuning options 19
Portability and migration options 21
Compiler customization options 21
Description of compiler options 21
AGGRCOPY 22
AGGREGATE | NOAGGREGATE 23
ANSIALIAS | NOANSIALIAS 24
ARCHITECTURE 27
ARGPARSE | NOARGPARSE 31
ARMODE | NOARMODE 32
ASM | NOASM 33
ASMDATASIZE 34
ASSERT(RESTRICT) | ASSERT(NORESTRICT) 35
BITFIELD(SIGNED) | BITFIELD(UNSIGNED) . . 36
CHARS(SIGNED) | CHARS(UNSIGNED) . . . 37
COMPACT | NOCOMPACT 38
COMPRESS | NOCOMPRESS 40
CONVLIT | NOCONVLIT 41
CSECT | NOCSECT 43
DEBUG | NODEBUG 46
DEFINE 48
DIGRAPH | NODIGRAPH 49
DSAUSER | NODSAUSER 50
ENUMSIZE 51
EPILOG 53
EVENTS | NOEVENTS 55
EXPMAC | NOEXPMAC 56
FLAG | NOFLAG 57
FLOAT 58
GOFF | NOGOFF 63
HALT 64
HALTONMSG | NOHALTONMSG 65
HGPR | NOHGPR 66
HOT | NOHOT 67
INCLUDE | NOINCLUDE 68
INFO | NOINFO 69
INITAUTO | NOINITAUTO 71
INLINE | NOINLINE 73
IPA | NOIPA 75
KEYWORD | NOKEYWORD 78
LANGLVL 79
LIBANSI | NOLIBANSI 82
LIST | NOLIST 83
LOCALE | NOLOCALE 85
LONGLONG | NOLONGLONG 87
LONGNAME | NOLONGNAME 88
LP64 | ILP32 89
LSEARCH | NOLSEARCH 91
MAKEDEP 97
MARGINS | NOMARGINS 99
MAXMEM | NOMAXMEM 100
MEMORY | NOMEMORY 102
METAL 103
NESTINC | NONESTINC 104
OE | NOOE 104
OPTFILE | NOOPTFILE 106
OPTIMIZE | NOOPTIMIZE 108
PHASEID | NOPHASEID 111
PPONLY | NOPPONLY 112
PREFETCH | NOPREFETCH 115
PROLOG 116
RENT | NORENT 117
RESERVED_REG 119
RESTRICT | NORESTRICT 120
ROCONST | NOROCONST 121
ROSTRING | NOROSTRING 123
ROUND 124
SEARCH | NOSEARCH 126
SEQUENCE | NOSEQUENCE 127
SERVICE | NOSERVICE 128
SEVERITY | NOSEVERITY 130
SHOWINC | NOSHOWINC 131
SHOWMACROS | NOSHOWMACROS . . . 132
SKIPSRC 133
SOURCE | NOSOURCE 134
SPLITLIST | NOSPLITLIST 135

© Copyright IBM Corp. 2018

SSCOMM | NOSSCOMM 138
STRICT | NOSTRICT 139
STRICT_INDUCTION | NOSTRICT_INDUCTION 141
SUPPRESS | NOSUPPRESS 142
SYSSTATE 143
TERMINAL | NOTERMINAL. 144
TUNE 145
UNDEFINE 148
UNROLL | NOUNROLL 149
UPCONV | NOUPCONV 150
VECTOR | NOVECTOR. 151
WARN64 | NOWARN64 153
WSIZEOF | NOWSIZEOF 154
Using compiler listing 155
IPA considerations. 155
Compiler listing components 155
Using the IPA link step listing. 156
IPA link step listing components 157

Chapter 3. Compiling 161
Input to the compiler. 161
Output from the compiler 162
Specifying output files 162
Valid input/output file types 164
Compiling under z/OS batch 165
Using cataloged procedures 165
Using special characters 166
Specifying source files 166
Specifying include files 167
Specifying output files 167
Compiling in the z/OS UNIX System Services environment. 168
Building a 64-bit application using metalc utility 169
Invoking IPA using metalc utility. 169
Compiling with IPA 170
IPA compile step 170
IPA link step 171
Working with object files 172
Browsing object files 172
Identifying object file variations 173
Using feature test macros 173
Using include files. 173
Specifying include file names 173
Forming file names 174
Forming data set names with LSEARCH | SEARCH options 175
Search sequence 177
Determining whether the file name is in absolute form 178
Using SEARCH and LSEARCH 180
Search sequences for include files 182

Chapter 4. Using IPA link step with programs 185
Invoking IPA using metalc utility. 185
Compiling under z/OS batch 186
Reference Information 186
IPA link step control file. 186
Object file directives understood by IPA . . . 190
Troubleshooting 190

Chapter 5. Assembling 191

Chapter 6. Binding programs 193
Binding under z/OS UNIX. 193
Binding under z/OS batch 193

Chapter 7. Running a C application 195

Chapter 8. Building Enterprise Metal C for z/OS programs 197

Chapter 9. Cataloged procedures . . . 199
Tailoring cataloged procedures 199
Data sets used 200
Description of data sets used 200

Chapter 10. CDAHLASM — Use the HLASM assembler to create DWARF debug information 205

Chapter 11. make utility 207
Creating makefiles. 207

Chapter 12. BPXBATCH utility 209

Chapter 13. as — Use the HLASM assembler to produce object files . . 213

Chapter 14. metalc — Compiler invocation using a customizable configuration file 219
Setting up the compilation environment 219
Environment variables 219
Setting up a configuration file 221
Configuration file attributes 221
Tailoring a configuration file 224
Default configuration file 224
Invoking the compiler 224
Supported options. 225
–q options syntax 225
Flag options syntax 226
Specifying compiler options 229

Appendix. Accessibility 233
Accessibility features 233
Consult assistive technologies 233
Keyboard navigation of the user interface 233
Dotted decimal syntax diagrams 233

Glossary 237
A 237
B 240
C 241
D 246
E 249

F 251
G 252
H 253
I. 254
J. 256
K 256
L 256
M 258
N 259
O 260
P 261
Q 264
R 264
S 266
T 269
U 270
V 270
W 271

Notices 273
Programming interface information 274
Trademarks 274
Standards 274

Index 277

About this document

This edition of Enterprise Metal C for z/OS User's Guide is intended for users of the IBM® Enterprise Metal C for z/OS®, V3.1 compiler. It provides you with information about implementing (compiling, assembling, linking, and running) programs that are written in C. It contains guidelines for preparing C programs to run on the z/OS operating system.

Who should read this document

This document is intended for users of Enterprise Metal C for z/OS.

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives that contain lower-case letters.	The **metalc** invocation command invokes the Enterprise Metal C for z/OS compiler.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.

How to read syntax diagrams

This section describes how to read syntax diagrams. It defines syntax diagram symbols, items that may be contained within the diagrams (keywords, variables, delimiters, operators, fragment references, operands) and provides syntax examples that contain these items.

Syntax diagrams pictorially display the order and parts (options and arguments) that comprise a command statement. They are read from left to right and from top to bottom, following the main path of the horizontal line.

For users accessing IBM Knowledge Center using a screen reader, syntax diagrams are provided in dotted decimal format.

The following symbols may be displayed in syntax diagrams:

Symbol
: **Definition**

►►—— Indicates the beginning of the syntax diagram.

© Copyright IBM Corp. 2018

──► Indicates that the syntax diagram is continued to the next line.

►── Indicates that the syntax is continued from the previous line.

──►◄ Indicates the end of the syntax diagram.

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

Note: If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

Item type
: **Definition**

Required
: Required items are displayed on the main path of the horizontal line.

Optional
: Optional items are displayed below the main path of the horizontal line.

Default
: Default items are displayed above the main path of the horizontal line.

The following table provides syntax examples.

Table 2. Syntax examples

Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	`►►──KEYWORD──required_item──────────────►◄`
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	`►►──KEYWORD──┬─required_choice1─┬──────►◄` `             └─required_choice2─┘`

Table 2. Syntax examples (continued)

Item	Syntax example
Optional item. Optional items appear below the main path of the horizontal line.	►►—KEYWORD—┬───────────────┬—►◄ └—optional_item—┘
Optional choice. An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	►►—KEYWORD—┬──────────────────┬—►◄ ├—optional_choice1—┤ └—optional_choice2—┘
Default. Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	┌—default_choice1——┐ ►►—KEYWORD—┼──────────────────┼—►◄ ├—optional_choice2—┤ └—optional_choice3—┘
Variable. Variables appear in lowercase italics. They represent names or values.	►►—KEYWORD—*variable*—►◄
Repeatable item. An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. A character within the arrow means you must separate repeated items with that character. An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected,or a single item can be repeated.	┌────────────────┐ ►►—KEYWORD—▼—repeatable_item—┴—►◄ ┌—,──────────────┐ ►►—KEYWORD—▼—repeatable_item—┴—►◄
Fragment. The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	►►—KEYWORD—┤ fragment ├—►◄ **fragment:** ├—┬—,required_choice1——————————————┬—┤ │ ┌—,default_choice——┐ │ └—,required_choice2—┴──────────────────┴—┘ └—,optional_choice—┘

Softcopy documents

The Enterprise Metal C for z/OS publications are supplied in PDF format and available for download from the Enterprise Metal C for z/OS Knowledge Center home page (www.ibm.com/support/knowledgecenter/en/SSSHGK_3.1.0/com.ibm.metalc.v3r1.doc/welcome.html).

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe website (www.adobe.com).

You can also browse the documents on the World Wide Web by visiting the Enterprise Metal C for z/OS Knowledge Center home page (www.ibm.com/support/knowledgecenter/en/SSSHGK_3.1.0/com.ibm.metalc.v3r1.doc/welcome.html).

Where to find more information

For an overview of the information associated with z/OS, see *z/OS Information Roadmap*.

Additional information on Enterprise Metal C for z/OS is available on the Marketplace page for Enterprise Metal C for z/OS (www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

z/OS Basic Skills in IBM Knowledge Center

z/OS Basic Skills in IBM Knowledge Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. IBM Knowledge Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, z/OS Basic Skills is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

z/OS Basic Skills in IBM Knowledge Center (www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zbasics/homepage.html) is available to all users (no login required).

Technical support

Additional technical support is available from the Enterprise Metal C for z/OS Support page (https://www.ibm.com/support/home/product/A032956W76367A04/IBM_Enterprise_Metal_C_for_z/OS). This page provides a portal with search capabilities to technical support FAQs and other support documents.

For the latest information about Enterprise Metal C for z/OS, visit Marketplace page for Enterprise Metal C for z/OS (www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

If you cannot find what you need, you can e-mail:

compinfo@cn.ibm.com

How to send your comments to IBM

We appreciate your input on this documentation. Please provide us with any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

You can send an email to compinfo@cn.ibm.com and include the following information:
- Your name and address
- Your email address
- Your phone or fax number
- The publication title and order number:
  Enterprise Metal C for z/OS User's Guide
  SC27-9051-00
- The topic and page number or URL of the specific information to which your comment relates
- The text of your comment.

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

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem, take one or more of the following actions:
- Visit the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Chapter 1. About IBM Enterprise Metal C for z/OS

IBM Enterprise Metal C for z/OS provides support for C application development on the z/OS platform.

Enterprise Metal C for z/OS includes:
- A C compiler (referred to as the Enterprise Metal C for z/OS compiler)
- A set of utilities for C application development

IBM Enterprise Metal C for z/OS delivers a high-level language alternative to writing programs in High Level Assembler (HLASM) and creates low-level, freestanding applications with high performance. The generated HLASM code has no Language Environment® runtime dependencies and follows the MVS linkage conventions for passing parameters, returning values, and setting up function save areas that are described in the *MVS Programming: Assembler Services Guide*. Enterprise Metal C for z/OS enables the use of C as the language for system programming where the Language Environment is either unavailable or undesirable, for example, writing system user exits. You cannot use the Enterprise Metal C for z/OS compiler to compile Language Environment dependent C source files.

The resulting programs could have direct access to z/OS system services and do not require the C runtime for execution. The final code generated by the Enterprise Metal C for z/OS compiler is in HLASM source code format. You need to invoke the HLASM assembler as an additional step to produce the object code from the compiler-generated HLASM source code.

A subset of the C library functions is provided. For further information on programming with Enterprise Metal C for z/OS and the library that is provided, see *Enterprise Metal C for z/OS Programming Guide and Reference*.

Using the Enterprise Metal C for z/OS compiler can be viewed as a joint venture between the compiler and users. The compiler is responsible for generating the machine instructions that represent the C program. Users are responsible for providing the stack space (or the dynamic storage area) required by the C program. Users can decide if the stack space is provided by using the default prolog and epilog code generated by the compiler, or by supplying their own prolog and epilog code. Users are also given the facilities to embed assembly statements within the C program so, for example, system macros can be invoked.

You may need to switch addressing mode (AMODE) between programs. The default AMODE assigned by the Enterprise Metal C for z/OS compiler is based on the LP64 compiler option or the ILP32 compiler option. AMODE 64 is assigned when LP64 is specified and AMODE 31 is assigned when ILP32 is specified. The Enterprise Metal C for z/OS compiler can generate code for calling an external function with an AMODE that is different from the default AMODE. This capability supports the creation of METAL C programs that require AMODE switching across functions. The resulting compiler generated code follows the linkage conventions expected by the called function, particularly in the areas of save area format and the parameter list width.

Notes:

© Copyright IBM Corp. 2018

1. The compiler-generated code does not establish code base registers.
2. Because of the flat name space and the case insensitivity required by HLASM, the compiler prepends extra qualifiers to user names to maintain uniqueness of each name seen by HLASM. This is referred to as *name-encoding*. For local symbols, HLASM also has the 63-character length limit. Name-encoded local symbols have a maximum of 63 characters. External symbols are not subject to the name-encoding scheme as they need to be referenced by the exact names.
3. The maximum length of an external symbol allowed by HLASM is 256 characters. You must ensure that all external symbols are acceptable to HLASM.
4. You must provide C library functions that are not provided by IBM if you need them.
5. It is your responsibility to ensure the correctness of your assembly code, including prolog and epilog code, and inlined assembly code.
6. When binding or linking, you may need to specify the ENTRY name.
7. No ASCII version of the Metal C runtime libraries is available, even though the ASCII compiler option is supported.

The IPA compile phase only produces a binary IPA object as the output file. It does not produce object code or HLASM source code.

During the IPA link phase, all external references must be resolved. IPA does not attempt to convert external object modules or load modules into object code for the inclusion in the IPA produced program. You need to provide the same set of library data sets to both IPA link and the binder for symbol resolution.

If you supply your own prolog/epilog code using the PROLOG and EPILOG compiler options, IPA link will keep the relationship between the prolog/epilog code and the designated functions at the compilation unit level.

If you have `#pragma insert_asm` in your source file, IPA link will assume the strong connection between the string provided by the pragma and the functions in the source file. IPA link will not move functions defined in that source file to anywhere else.

The output file from the IPA link step is one single HLASM source file for the whole program. Under z/OS UNIX, the output HLASM source file resides in the directory where the IPA link took place. The default output file name for z/OS UNIX is `a.s`. In BATCH mode, the output HLASM source file goes in the data set allocated to DD SYSLIN in the IPA link step.

The Enterprise Metal C for z/OS compiler

The following sections describe the C language and the Enterprise Metal C for z/OS compiler.

The C language

The C language is a general purpose, versatile, and functional programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

Enterprise Metal C for z/OS compiler features

The Enterprise Metal C for z/OS compiler offers many features to increase your productivity and improve program execution times:

- Minimizes dependence on expert HLASM skills.
- Optimization support:
  - Algorithms to take advantage of the IBM Z® environment to achieve improved optimization and memory usage through the **ARCH**, **TUNE**, **OPTIMIZE**, and **IPA** compiler options.
  - The **OPTIMIZE** compiler option, which instructs the compiler to optimize the machine instructions it generates to try to produce faster-running object code and improve application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across procedural and compilation unit boundaries, thereby optimizing application performance at run time.
  - Additional optimization capabilities are available with the **INLINE** compiler option.
- Take advantage of new hardware features by simply recompiling the code.
- Full program reentrancy

  With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the Link Pack Area (LPA) or the Extended Link Pack Area (ELPA) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. Programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, programmers can use constructed reentrancy. To do this, compile programs with the **RENT** option and use the program management binder supplied with z/OS and program management binder.
- The ability to call and be called by other languages such as assembler, C/C++, COBOL, PL/1, compiled Java™, and Fortran, to enable you to integrate Enterprise Metal C for z/OS code with existing applications.
- Support runtime independent features in the following standards at the system level:
  - System programming capabilities, which allow you to use Enterprise Metal C for z/OS in place of assembler
  - *ISO/IEC 9899:1999*
  - *ISO/IEC 9945-1:1990 (POSIX-1)/IEEE POSIX 1003.1-1990*
  - The core features of *IEEE POSIX 1003.1a, Draft 6, July 1991*
  - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
  - The core features of *IEEE POSIX 1003.4a, Draft 6, February 1992* (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
  - *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*
  - The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the IBM Z® environment.
  - *X/Open CAE Specification, Networking Services, Issue 4*
  - *X/Open Specification Programming Languages, Issue 3, Common Usage C*
  - A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*
  - A subset of *ISO/IEC 9899:2011*
  - *ANSI/ISO 9899:1990[1992]* (formerly *ANSI X3.159-1989 C*)
  - *FIPS-160*

- Support for the Euro currency

metalc utility

The Enterprise Metal C for z/OS compiler provide the **metalc** utility to invoke the compiler using a customizable configuration file.

About assembling, linking, and binding

You can use the Enterprise Metal C for z/OS compiler to compile your programs that are written in the C language syntax to generate code in assembler source program format that can be compiled by the High Level Assembler compiler. When describing the process to assemble an application, this document refers to the *assemble step*.

When describing the process to build an application, this document refers to the *bind step*.

Normally, the program management binder is used to perform the bind step. However, in many cases the link step can be used in place of the bind step. When they cannot be substituted, and the program management binder alone must be used, it will be stated.

The terms *bind* and *link* have multiple meanings.

- With respect to building an application:

  In both instances, the program management binder is performing the actual processing of converting the object file(s) into the application executable module.

  Object files with reentrant writable static symbols and DLL-style function calls require additional processing to build global data for the application. The term *link* refers to the case where the binder does not perform this additional processing, because none of the object files in the application use constructed reentrancy or long names.

  The term *bind* refers to the case where the binder is required to perform this processing.
- With respect to executing code in an application:

  The *linkage definition* refers to the program call linkage between program functions. This includes the passing of control and parameters.

  Enterprise Metal C for z/OS supports the MVS™ linkage convention for C.

File format considerations

You can use the binder in place of the linkage editor but there are exceptions involving file format considerations. For further information, on when you cannot use the binder, see Chapter 6, “Binding programs,” on page 193.

z/OS UNIX System Services

z/OS UNIX System Services provides capabilities under z/OS to make it easier to implement or port applications in an open, distributed environment. You can build Metal C applications under z/OS UNIX.

z/OS UNIX provides support for both existing z/OS applications and new z/OS UNIX applications through the following ways:

- C programming language support as defined by ISO C

- z/OS UNIX extensions that provide z/OS-specific support beyond the standards
- The z/OS UNIX Shell and Utilities feature, which provides:
  - A shell, based on the Korn Shell and compatible with the Bourne Shell
  - A shell, `tcsh`, based on the C shell, `csh`
  - Tools and utilities that support the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide z/OS support. The following list is a partial list of utilities that are included:

    as Invokes HLASM to create assembler applications

    BPXBATCH
    Allows you to submit batch jobs that run shell commands, scripts, or Enterprise Metal C for z/OS executable files in z/OS UNIX files from a shell session

    ld Combines object files and archive files into an output executable file, resolving external references

    metalc Allows you to invoke the compiler using a customizable configuration file
- Access to the Hierarchical File System (HFS), with support for the POSIX.1 and XPG4 standards
- Access to the z/OS File System (zFS), which provides performance improvements over HFS, and also supports the POSIX.1 and XPG4 standards

For application developers who have worked with other UNIX environments, the z/OS UNIX Shell and Utilities is a familiar environment for Enterprise Metal C for z/OS application development. If you are familiar with existing MVS development environments, you may find that the z/OS UNIX System Services environment can enhance your productivity. Refer to *z/OS UNIX System Services User's Guide* for more information about the Shell and Utilities.

Additional features of Enterprise Metal C for z/OS

Feature	Description
`long long` data type	Enterprise Metal C for z/OS supports `long long` as a native data type when the compiler option **`LANGLVL(LONGLONG)`** is in effect. This option is enabled by default by the compiler option **`LANGLVL(EXTENDED)`**. The compiler supports `long long` as a native data type when the **`LANGLVL(STDC99)`** option or **`LANGLVL(EXTC99)`** option is in effect.
Extended precision floating-point numbers	Enterprise Metal C for z/OS provides three IBM z/Architecture® floating-point number data types: single precision (32 bits), declared as `float`; double precision (64 bits), declared as `double`; and extended precision (128 bits), declared as `long double`. Extended precision floating-point numbers give greater accuracy to mathematical calculations. Enterprise Metal C for z/OS also supports IEEE 754 floating-point representation (base-2 or binary floating-point formats). By default, `float`, `double`, and `long double` values are represented in the IEEE 754 floating-point representation . For details on this support, see "FLOAT" on page 58.
Selected built-in library functions	For selected library functions, the compiler generates an instruction sequence directly into the object code during optimization to improve execution performance. String and character functions are examples of these built-in functions. No actual calls to the library are generated when built-in functions are used.

Feature	Description
Packed structures and unions	Enterprise Metal C for z/OS provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a Enterprise Metal C for z/OS program or to define structures that are laid out according to COBOL or PL/I structure alignment rules.
Exploitation of hardware	Use the **ARCHITECTURE** compiler option to select the minimum level of machine architecture on which your program will run. Note that certain features provided by the compiler require a minimum architecture level. For more information, refer to "ARCHITECTURE" on page 27. Use the **TUNE** compiler option to optimize your application for a specific machine architecture within the constraints imposed by the **ARCHITECTURE** option. The **TUNE** level must not be lower than the setting in the **ARCHITECTURE** option. For more information, refer to "TUNE" on page 145.
Long name support	For portability, external names can be mixed case and up to 32 K - 1 characters in length.
Built-in functions for floating-point and other hardware instructions	Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to Enterprise Metal C for z/OS programs.
Vector processing support	Enterprise Metal C for z/OS compiler provides vector programming support for programmers to make use of the Vector Facility for z/Architecture.

Chapter 2. Compiler options

This information describes the options that you can use to alter the compilation of your program.

Specifying compiler options

You can override your installation default options when you compile your Enterprise Metal C for z/OS program, by specifying an option in one of the following ways:

- In the CPARM parameter of the IBM-supplied cataloged procedures, when you are compiling under z/OS batch.

  See Chapter 3, "Compiling," on page 161 and Chapter 9, "Cataloged procedures," on page 199 for details.
- In your own JCL procedure, by passing a parameter string to the compiler.
- In an options file. See "OPTFILE | NOOPTFILE" on page 106 for details.
- In a **#pragma options** preprocessor directive within your source file. See "Specifying compiler options using #pragma options" on page 9 for details.

  Compiler options that you specify on the command line or in the CPARM parameter of IBM-supplied cataloged procedures can override compiler options that are used in **#pragma options**. The exception is CSECT, where the **#pragma csect** directive takes precedence.
- On the command line of the **metalc** utility, by using the **-q** option or the **-Wc** and **-Wl,I** options to pass options to the compiler.

IPA considerations

The following sections explain what you should be aware of if you request Interprocedural Analysis (IPA) through the IPA option.

Applicability of compiler options under IPA

You should keep the following points in mind when specifying compiler options for the IPA compile or IPA link step:

- Many compiler options do not have any special effect on IPA. For example, the **PPONLY** option processes source code, then terminates processing prior to IPA compile step analysis.
- `#pragma` directives in your source code, and compiler options you specify for the IPA compile step, may conflict across compilation units.

  `#pragma` directives in your source code, and compiler options you specify for the IPA compile step, may conflict with options you specify for the IPA link step.

  IPA will detect such conflicts and apply default resolutions with appropriate diagnostic messages. The Compiler Options Map section of the IPA link step listing displays the conflicts and their resolutions.

  To avoid problems, use the same options and suboptions on the IPA compile and IPA link steps. Also, if you use `#pragma` directives in your source code, specify the corresponding options for the IPA link step.
- If you specify a compiler option that is irrelevant for a particular IPA step, IPA ignores it and does not issue a message.

© Copyright IBM Corp. 2018

During IPA compile step processing, IPA handles conflicts between IPA suboptions and certain compiler options that affect code generation.

If you specify a compiler option for the IPA compile step, but do not specify the corresponding suboption of the IPA option, the compiler option may override the IPA suboption. Table 3 shows how the OPT and LIST compiler options interact with the OPT and LIST suboptions of the IPA option. The xxxx indicates the name of the option or suboption. NOxxxx indicates the corresponding negative option or suboption.

Table 3. Interactions between compiler options and IPA suboptions

Compiler Option	Corresponding IPA Suboption	Value used in IPA Object
no option specified	no suboption specified	NOxxxx
no option specified	NOxxxx	NOxxxx
no option specified	xxxx	xxxx
NOxxxx	no option specified	NOxxxx
NOxxxx	NOxxxx	NOxxxx
NOxxxx	xxxx	xxxx
xxxx	no option specified	xxxx
xxxx	NOxxxx	xxxx [1]
xxxx	xxxx	xxxx

Note: [1]An informational message is produced that indicates that the suboption NOxxxx is promoted to xxxx.

Using special characters

Under TSO

When z/OS UNIX file names contain the special characters
- blank
- backslash
- double quotation mark

A backslash (\) must precede these characters.

Note: Under TSO, a backslash \ must precede special characters in file names and options.

Two backslashes must precede suboptions that contain these special characters:
- left parenthesis (
- right parenthesis)
- comma
- backslash
- blank
- double quotation mark
- less than <
- greater than >

For example:

```
def(errno=\\(*__errno\\(\\)\\))
```

Under the z/OS UNIX System Services shell

The z/OS UNIX System Services shell imposes its own parsing rules.

metalc uses the **-q** syntax, which does not use parentheses and is more convenient for shell invocation. See Chapter 14, “metalc — Compiler invocation using a customizable configuration file,” on page 219 for more information.

Under z/OS batch

When invoking the compiler directly (not through a cataloged procedure), you should type a single quotation mark (') within a string as two single quotation marks (''), as follows:

```
//COMPILE EXEC PGM=CJTDRVR,PARM='OPTFILE(''USERID.OPTS'')'
```

If you are using the same string to pass a parameter to a JCL PROC, use four single quotation marks (''''), as follows:

```
//COMPILE EXEC MTCC,CPARM='OPTFILE(''''USERID.OPTS'''')'
```

Special characters in z/OS UNIX file names that are referenced in DD cards do not need a preceding backslash. For example, the special character blank in the file name `obj 1.o` does not need a preceding backslash when it is used in a `DD` card:

```
//SYSLIN DD PATH='u/user1/obj 1.o'
```

A backslash must precede special characters in z/OS UNIX file names that are referenced in the PARM statement. The special characters are: backslash, blank, and double quotation mark. For example, a backslash precedes the special character blank in the file name `opt file`, when used in the PARM keyword:

```
//STEP1 EXEC PGM=CJTDRVR,PARM='OPTFILE(/u/user1/opt\ file)'
```

Specifying compiler options using #pragma options

You can use the **#pragma options** preprocessor directive to override the default values for compiler options. The exception is **LONGNAME** | **NOLONGNAME**, where the compiler options override the `#pragma preprocessor` directives. Compiler options that are specified on the command line or in the CPARM parameter of the IBM-supplied cataloged procedures can override compiler options that are used in **#pragma options**. The exception is **CSECT**, where the **#pragma csect** directive takes precedence.

The **#pragma options** preprocessor directive must appear before the first C source statement in your input source file. Only comments and other preprocessor directives can precede the **#pragma options** directive. Only the options that are listed below can be specified in a **#pragma options** directive. If you specify a compiler option that is not in the following list, the compiler generates a warning message, and does not use the option.

AGGREGATE	ANSIALIAS
ARCHITECTURE	INLINE
LIBANSI	MAXMEM
OPTIMIZE	RENT
SERVICE	TUNE
UPCONV	

Notes:

1. When you specify conflicting attributes explicitly, or implicitly by the specification of other options, the last explicit option is accepted. The compiler usually does not issue a diagnostic message indicating that it is overriding any options.
2. When you compile your program with the **SOURCE** compiler option, an options list in the listing indicates the options in effect at invocation. The values in the list are the options that are specified on the command line, or the default options that were specified at installation. These values do not reflect options that are specified in the **#pragma options** directive.

Specifying compiler options under z/OS UNIX

The **metalc** utility invokes the Enterprise Metal C for z/OS compiler with the compiler options. For further information, see "Compiler option defaults."

To change compiler options, use an appropriate **metalc** utility option. For example, use **-I** to set the search option that specifies where to search for #include files. If there is no appropriate **metalc** option, use **-q** or **-Wc** to specify a needed compiler option. For example, specify **-Wc,expo** to export all functions and variables.

For a detailed description **metalc** utility, refer to Chapter 14, "metalc — Compiler invocation using a customizable configuration file," on page 219.

For compiler options that take file names as suboptions, you can specify a sequential data set, a partitioned data set, or a partitioned data set member by prefixing the name with two slashes (//). The rest of the name follows the same syntax rule for naming data sets. Names that are not preceded with two slashes are z/OS UNIX file names. For example, to specify HQ.PROG.LIST as the source listing file (HQ being the high-level qualifier), use SOURCE(//'HQ.PROG.LIST'). The single quotation mark is needed for specifying a full file name with a high-level qualifier.

Note: Both the IPA link step and IPA compile step make use of 64-bit virtual memory, which might cause the Enterprise Metal C for z/OS compiler to abend if there is insufficient storage. Increasing the default MEMLIMIT system parameter size in the SMFPRMx parmlib member to 3000 MB can overcome the problem. The default takes effect if a job does not specify MEMLIMIT in the JCL JOB or EXEC statement, or REGION=0 in the JCL; the MEMLIMIT specified in an IEFUSI exit routine overrides all other MEMLIMIT settings. For information on the **ulimit** command, which can be used in z/OS UNIX to set MEMLIMIT, see *z/OS UNIX System Services Command Reference*. For additional information about the MEMLIMIT system parameter, see *z/OS MVS Programming: Extended Addressability Guide*.

Compiler option defaults

You can use various options to change the compilation of your program. You can specify compiler options when you invoke the compiler or, in a C program, in a **#pragma options** directive in your source program. Options, that you specify when you invoke the compiler, override installation defaults and most compiler options that are specified through a **#pragma options** directive.

The compiler option defaults that are supplied by IBM can be changed to other selected defaults when Enterprise Metal C for z/OS is installed.

To find out the current defaults, compile a program with only the **SOURCE** compiler option specified. The compiler listing shows the options that are in effect at invocation. The listing does not reflect options that are specified through a **#pragma options** directive in the source file.

The **metalc** utility that runs in the z/OS UNIX shell specify certain compiler options in order to support POSIX standards. For a detailed description, refer to Chapter 14, "metalc — Compiler invocation using a customizable configuration file," on page 219, or to the *z/OS UNIX System Services Command Reference*. For some options, the utility specify values that are different than the supplied defaults in MVS batch or TSO environments. However, for many options, the utility specifies the same values as in MVS batch or TSO. There are also some options that the utility does not specify explicitly. In those cases, the default value is the same as in batch or TSO. An option that you specify explicitly using the **metalc** utility overrides the setting of the same option if it is specified using a **#pragma options** directive. The exception is CSECT, where the **#pragma csect** directive takes precedence.

In effect, invoking the compiler with the **metalc** utility overrides the default values for many options, compared to running the compiler in MVS batch or TSO. Any overrides of the defaults by the **metalc** utility are noted in the DEFAULT category for the option. As the compiler defaults can always be changed during installation, you should always consult the compiler listing to verify the values passed to the compiler. See "Using compiler listing" on page 155 for more information.

Summary of compiler options

Most compiler options have a positive and negative form. The negative form is the positive with **NO** before it. For example, **NOASM** is the negative form of **ASM**.

Table 4 lists the compiler options in alphabetical order, their abbreviations, and the defaults that are supplied by IBM. Suboptions inside square brackets are optional.

Note: For a description of the compiler options that can be specified with **metalc**, type **metalc** without arguments to access the help file.

The *Compile* and *IPA link* columns, which are shown in Table 4, indicate where the option is accepted by the compiler but this acceptance does not necessarily cause an action; for example, IPA LINK accepts the **MARGINS** option but ignores it. This acceptance also means that a diagnostic message is not generated. These options are accepted regardless of whether they are for **NOIPA** or **IPA(NOLINK)**.

Table 4. Compiler options, abbreviations, and IBM-supplied defaults

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	Compile	IPA Link	More Information
AGGRCOPY[(OVERLAP | NOOVERLAP)]	NOAGGRC(NOOVERL)	✓	✓	See detail
AGGREGATE | NOAGGREGATE	NOAGG	✓	✓	See detail
ANSIALIAS | NOANSIALIAS	ANS	✓	✓	See detail
ARCHITECTURE(*n*)	ARCH(10)	✓	✓	See detail
ARGPARSE | NOARGPARSE	ARG	✓	✓	See detail
ARMODE | NOARMODE	NOARMODE	✓	✓	See detail
ASM | NOASM	ASM	✓	✓	See detail
ASMDATASIZE(num)	ASMDS(256)	✓	✓	See detail

Table 4. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	Compile	IPA Link	More Information
ASSERT(RESTRICT) | ASSERT(NORESTRICT)	ASSERT(RESTRICT)	✓	✓	See detail
BITFIELD(SIGNED|UNSIGNED)	BITF(UNSIGNED)	✓	✓	See detail
CHARS(SIGNED | UNSIGNED)	CHARS(UNSIGNED)	✓	✓	See detail
COMPACT | NOCOMPACT	NOCOMPACT	✓	✓	See detail
COMPRESS | NOCOMPRESS	NOCOMPRESS	✓	✓	See detail
CONVLIT[(subopts)] | NOCONVLIT[(subopts)]	NOCONV (IBM-1047, NOWCHAR)	✓	✓	See detail
CSECT([qualifier]) | NOCSECT([qualifier])	CSE	✓	✓	See detail
DEBUG[(subopts)] | NODEBUG[(subopts)]	NODEBUG	✓		See detail
DEFINE(name1[= | =def1], name2[= | =def2],...)	**Note:** No default user definitions.	✓	✓	See detail
DIGRAPH | NODIGRAPH	DIGR	✓	✓	See detail
DSAUSER | NODSAUSER	NODSAUSER	✓	✓	See detail
ENUMSIZE(subopts)	ENUM(SMALL)	✓	✓	See detail
EPILOG(subopts)	**Note:** The compiler generates default epilog code for the functions that do not have user-supplied epilog code.	✓	✓	See detail
EVENTS[(filename)] | NOEVENTS	NOEVENT	✓	✓	See detail
EXPMAC | NOEXPMAC	NOEXP	✓	✓	See detail
FLAG(severity) | NOFLAG	FL(I)	✓	✓	See detail
FLOAT(subopts)	FLOAT(IEEE, FOLD, NOMAF, NORRM, AFP)	✓	✓	See detail
GOFF | NOGOFF	NOGOFF	✓	✓	See detail
HALT(num)	HALT(16)	✓	✓	See detail
HALTONMSG(msgno) | NOHALTONMSG	NOHALTON	✓	✓	See detail
HGPR[(subopt)] | NOHGPR	HGPR(PRESERVE)	✓	✓	See detail
HOT | NOHOT	NOHOT	✓		See detail
INCLUDE(file) | NOINCLUDE	NOINCLUDE	✓	✓	See detail
INFO[(subopts)] | NOINFO	NOIN	✓	✓	See detail
INITAUTO(number [,word]) | NOINITAUTO	NOINITA	✓	✓	See detail
INLINE | NOINLINE	NOINLINE	✓	✓	See detail
IPA[(subopts)] | NOIPA[(subopts)]	NOIPA	✓	✓	See detail
KEYWORD(name) | NOKEYWORD(name)	All of the built-in keywords defined in the C language standard are reserved as keywords.	✓	✓	See detail
LANGLVL(subopts)	LANG(EXTENDED)	✓	✓	See detail
LIST[(filename)] | NOLIST [(filename)]	NOLIS	✓	✓	See detail
LIBANSI | NOLIBANSI	NOLIB	✓	✓	See detail
LOCALE[(name)] | NOLOCALE	NOLOC	✓	✓	See detail
LONGNAME | NOLONGNAME	NOLO	✓	✓	See detail

Table 4. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	Compile	IPA Link	More Information
LP64 ǀ ILP32	ILP32	✓	✓	See detail
LONGLONG ǀ NOLONGLONG	LONGLONG	✓	✓	See detail
LSEARCH(subopts) ǀ NOLSEARCH	NOLSE	✓	✓	See detail
MAKEDEP[(GCC ǀ PPONLY)]	Not applicable.	✓		See detail
MARGINS(m,n) ǀ NOMARGINS	For fixed record format C source files, the default is MAR(1,72).	✓	✓	See detail
MAXMEM(size) ǀ NOMAXMEM	MAXM(2097152)	✓	✓	See detail
MEMORY ǀ NOMEMORY	MEM	✓	✓	See detail
METAL	METAL	✓	✓	See detail
NESTINC(num) ǀ NONESTINC	NEST(255)	✓	✓	See detail
OE[(filename)] ǀ NOOE[(filename)]	NOOE	✓	✓	See detail
OPTFILE[(filename)] ǀ NOOPTFILE[(filename)]	NOOPTF	✓	✓	See detail
OPTIMIZE[(level)] ǀ NOOPTIMIZE	NOOPT	✓	✓	See detail
PHASEID ǀ NOPHASEID	NOPHASEID	✓	✓	See detail
PPONLY[(subopts)] ǀ NOPPONLY[(subopts)]	NOPP	✓	✓	See detail
PREFETCH ǀ NOPREFETCH	PREFETCH	✓		See detail
PROLOG(subopt)	The compiler generates default prolog code for the functions that do not have user-supplied prolog code.	✓	✓	See detail
RENT ǀ NORENT	NORENT	✓	✓	See detail
RESERVED_REG(subopt)	**Note:** No default user definitions.	✓	✓	See detail
RESTRICT[(subopts)] ǀ NORESTRICT	NORESTRICT	✓	✓	See detail
ROCONST ǀ NOROCONST	NOROC	✓	✓	See detail
ROSTRING ǀ NOROSTRING	RO	✓	✓	See detail
ROUND(subopt)	ROUND(N)	✓	✓	See detail
SEARCH(opt1,opt2,...) ǀ NOSEARCH	SEARCH(/usr/include/)	✓	✓	See detail
SEQUENCE(m,n) ǀ NOSEQUENCE	For variable record format C source files, the default is NOSEQUENCE. For fixed record format C source files, the default is SEQUENCE(73,80).	✓	✓	See detail
SERVICE(string) ǀ NOSERVICE	NOSERV	✓	✓	See detail
SEVERITY(severity level(msg-no)) ǀ NOSEVERITY	NOSEVERITY	✓		See detail
SHOWINC ǀ NOSHOWINC	NOSHOW	✓	✓	See detail
SHOWMACROS[(subopts)] ǀ NOSHOWMACROS	NOSHOWM	✓		See detail
SKIPSRC (SHOW ǀ HIDE)	SKIPS(SHOW)	✓	✓	See detail
SOURCE[(filename)] ǀ NOSOURCE[(filename)]	NOSO	✓	✓	See detail

Table 4. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	Compile	IPA Link	More Information
SPLITLIST | NOSPLITLIST	NOSPLITLIST	✓	✓	See detail
SSCOMM | NOSSCOMM	NOSS	✓	✓	See detail
STRICT | NOSTRICT	STRICT	✓	✓	See detail
STRICT_INDUCTION | NOSTRICT_INDUCTION	NOSTRICT_INDUC	✓	✓	See detail
SUPPRESS(msg-no) | NOSUPPRESS(msg-no)	NOSUPP	✓	✓	See detail
SYSSTATE(subopts)	SYSSTATE(NOASCENV, OSREL(NONE))	✓	✓	See detail
TERMINAL | NOTERMINAL	TERM	✓	✓	See detail
UNDEFINE(name)	No default.	✓	✓	See detail
TUNE(n)	TUN(10)	✓	✓	See detail
UNROLL(subopts)	UNROLL(AUTO)	✓	✓	See detail
UPCONV | NOUPCONV	NOUPC	✓	✓	See detail
VECTOR | NOVECTOR	NOVECTOR	✓	✓	See detail
WARN64 | NOWARN64	NOWARN64	✓	✓	See detail
WSIZEOF | NOWSIZEOF	NOWSIZEOF	✓	✓	See detail

Compiler output options

The options in Table 5 control the type of file output the compiler produces, as well as the locations of the output. These are the basic options that determine the compiler components that will be invoked, the preprocessing, compilation, assemble, and linking steps that will (or will not) be taken, and the kind of output to be generated.

Table 5. Compiler output options

Option	Description	Compile	IPA Link	More Information
MAKEDEP	Analyzes each source file to determine what dependency it has on other files and places this information into an output file.	✓		See detail
PPONLY	Specifies that only the preprocessor is to be run and not the compiler.	✓	✓	See detail
SHOWMACROS	Emits macro definitions at the end of compilation to preprocessed output.	✓		See detail

Compiler input options

The options in Table 6 specify the type and location of your source files.

Table 6. Compiler input options

Option	Description	Compile	IPA Link	More Information
INCLUDE	Inserts an `#include` statement for each file specified with the **INCLUDE** option before the first line of the source file.	✓		See detail
LSEARCH	Specifies the directories or data sets to be searched for user include files.	✓		See detail
MARGINS	Specifies, inclusively, the range of source column numbers that will be compiled.	✓		See detail
NESTINC	Specifies the number of nested include files to be allowed in your source program.	✓		See detail
OE	Specifies the rules used when searching for files specified with `#include` directives.	✓		See detail
SEARCH	Specifies the directories or data sets to be searched for system include files.	✓		See detail
SEQUENCE	Specifies the columns used for sequence numbers.	✓		See detail

Language element control options

The options in Table 7 allow you to specify the characteristics of the source code. You can also use these options to enforce or relax language restrictions and enable or disable language extensions.

Table 7. Language element control options

Option	Description	Compile	IPA Link	More Information
ASM	Enables embedded assembler source inside C programs.	✓		See detail
DEFINE	Defines a macro as in a `#define` preprocessor directive.	✓		See detail
DIGRAPH	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.	✓		See detail
KEYWORD	Controls whether the specified *name* is treated as a keyword or an identifier whenever it appears in your source.	✓		See detail
LANGLVL	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.	✓		See detail
LONGLONG	Controls whether to allow the pre-C99 `long long` integer types in your programs.	✓		See detail
SSCOMM	Allows comments to be specified by two slashes (//), which supports C++ style comments in C code.	✓		See detail

Table 7. Language element control options (continued)

Option	Description	Compile	IPA Link	More Information
UNDEFINE	Undefines preprocessor macro names.	✓		See detail
VECTOR	Enables compiler support for vector data types and operations.	✓		See detail

Object code control options

The options in Table 8 affect the characteristics of the object code generated by the compiler.

Table 8. Object code control options

Option	Description	Compile	IPA Link	More Information
ARGPARSE	Parses arguments provided on the invocation line.	✓	✓	See detail
ARMODE	Specifies that all functions in the C source file will operate in access-register (AR) mode.	✓		See detail
ASMDATASIZE	Provides the default data area size for the data areas defined by user-supplied assembly statements.	✓		See detail
COMPRESS	Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.	✓	✓	See detail
CSECT	Instructs the compiler to generate CSECT names in the output object module.	✓	✓	See detail
DSAUSER	Requests a user field of the size of a pointer to be reserved on the stack.	✓		See detail
EPILOG	Enables you to provide your own function exit code for all your functions that have `extern` scope.	✓		See detail
GOFF	Instructs the compiler to produce an object file in the Generalized Object File Format (GOFF).	✓	✓	See detail
ILP32	Instructs the compiler to generate AMODE 31 code.	✓	✓	See detail
LOCALE	Specifies the locale to be used by the compiler as the current locale throughout the compilation unit.	✓	✓	See detail
LONGNAME	Provides support for external names of mixed case and up to 1024 characters long.	✓	✓	See detail
LP64	Instructs the compiler to generate AMODE 64 code using the z/Architecture 64-bit instructions.	✓	✓	See detail
METAL	METAL is accepted and ignored to allow interoperability with the z/OS XL C compiler.	✓		See detail
PROLOG	Enables you to provide your own function entry code for all your functions that have `extern` scope.	✓		See detail
RENT	Generates reentrant code.	✓	✓	See detail

Table 8. Object code control options (continued)

Option	Description	Compile	IPA Link	More Information
RESERVED_REG	Instructs the compiler not to use the specified general purpose register (GPR) during the compilation.	✓		See detail
ROCONST	Specifies the storage location for constant values.	✓	✓	See detail
ROSTRING	Specifies the storage type for string literals.	✓	✓	See detail
SYSSTATE	Provides additional SYSSTATE macro parameters to the SYSSTATE macro that is generated by the compiler.	✓	✓	See detail
WSIZEOF	Causes the `sizeof` operator to return the widened size for function return types.	✓	✓	See detail

Floating-point and integer control options

Specifying the details of how your applications perform calculations can allow you to take better advantage of your system's floating-point performance and precision, including how to direct rounding. However, keep in mind that strictly adhering to IEEE floating-point specifications can impact the performance of your application. Using the options in Table 9, you can control trade-offs between floating-point performance and adherence to IEEE standards.

The table also lists options that allow you to control the characteristics of integer variables, values and types.

Table 9. Floating-point and integer control options

Option	Description	Compile	IPA Link	More Information
BITFIELD	Specifies whether bit fields are `signed` or `unsigned`.	✓	✓	See detail
CHARS	Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.	✓	✓	See detail
ENUMSIZE	Specifies the amount of storage occupied by enumerations.	✓	✓	See detail
FLOAT	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.	✓	✓	See detail
ROUND	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.	✓	✓	See detail

Error-checking and debugging options

You can use the options in Table 10 on page 18 to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your compile time, or introduce runtime checking that can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the "Listings, messages, and compiler information options" section.

Table 10. Error-checking and debugging options

Option	Description	Compile	IPA Link	More Information
DEBUG	Instructs the compiler to generate debug information.	✓		See detail
EVENTS	Produces an event file that contains error information and source file statistics.	✓	✓	See detail
HALT	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity specified for this option.	✓	✓	See detail
HALTONMSG	Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.	✓	✓	See detail
INFO	Produces groups of informational messages.	✓	✓	See detail
INITAUTO	Initializes automatic variables to a specific value for debugging purposes.	✓	✓	See detail
SERVICE	Places a *string* in the object module, which is displayed in the traceback if the application fails abnormally.	✓	✓	See detail
WARN64	Generates diagnostic messages, which enable checking for possible data conversion problems between 32-bit and 64-bit compiler modes.	✓	✓	See detail

Listings, messages, and compiler information options

The options in Table 11 allow you to control the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those in the "Error-checking and debugging options" on page 17 section to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 11. Listings, messages, and compiler information options

Option	Description	Compile	IPA Link	More Information
AGGREGATE	Lists structures and unions, and their sizes.	✓	✓	See detail
EXPMAC	Lists all expanded macros in the source listing.	✓		See detail
FLAG	Limits the diagnostic messages to those of a specified level or higher.	✓	✓	See detail
LIST	Produces a compiler listing that includes a list of options and the compiler version.	✓	✓	See detail

Table 11. Listings, messages, and compiler information options (continued)

Option	Description	Compile	IPA Link	More Information
PHASEID	Causes each compiler component (phase) to issue an informational message as each phase begins execution, which assists you with determining the maintenance level of each compiler component (phase). This message identifies the compiler phase module name, product identification, and build level.	✓	✓	See detail
SEVERITY	Changes the default severity for certain messages that the user has specified, if these messages are generated by the compiler.	✓		See detail
SHOWINC	When used with the **SOURCE** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.	✓		See detail
SKIPSRC	Controls whether or not source statements skipped by the compiler are shown in the listing, when the **SOURCE** option is in effect.	✓		See detail
SOURCE	Produces a compiler listing file that includes the source section of the listing.	✓	✓	See detail
SPLITLIST	Enables the compiler to write the IPA Link phase listing to multiple PDS members, PDSE members, or z/OS UNIX files.		✓	See detail
SUPPRESS	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.	✓	✓	See detail
TERMINAL	Directs diagnostic messages to be displayed on the terminal.	✓	✓	See detail

Optimization and tuning options

You can control the optimization and tuning process, which can improve the performance of your application at run time, using the options in Table 12. Remember that not all options benefit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Table 12. Optimization and tuning options

Option	Description	Compile	IPA Link	More Information
AGGRCOPY	Enables destructive copy operations for structures and unions, which can improve performance.	✓	✓	See detail
ANSIALIAS	Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C standard, and can therefore be compiled with higher performance optimization of the generated code.	✓	✓	See detail
ARCHITECTURE	Specifies the machine architecture for which the executable program instructions are to be generated.	✓	✓	See detail

Table 12. Optimization and tuning options (continued)

ASSERT(RESTRICT)	Enables optimizations for restrict qualified pointers.	✓	✓	See detail
COMPACT	Avoids optimizations that increase object file size.	✓	✓	See detail
HGPR	Enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.	✓	✓	See detail
HOT	Performs high-order loop analysis and transformations (HOT) during optimization.	✓		See detail
INLINE	Attempts to inline functions instead of generating calls to those functions, for improved performance.	✓	✓	See detail
IPA	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).	✓	✓	See detail
LIBANSI	Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard.	✓	✓	See detail
MAXMEM	Limits the amount of memory used for local tables, and that the compiler allocates while performing specific, memory-intensive optimizations, to the specified number of kilobytes.	✓	✓	See detail
OPTIMIZE	Specifies whether to optimize code during compilation and, if so, at which level.	✓	✓	See detail
PREFETCH	Inserts prefetch instructions automatically where there are opportunities to improve code performance.	✓		See detail
RESTRICT	Indicates to the compiler that all pointer parameters in some or all functions are disjoint.	✓	✓	See detail
STRICT	Used to prevent optimizations done by default at optimization levels **OPT(3)**, and, optionally at **OPT(2)**, from re-ordering instructions that could introduce rounding errors.	✓	✓	See detail
STRICT_INDUCTION	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.	✓	✓	See detail
TUNE	Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture.	✓	✓	See detail
UNROLL	Controls loop unrolling, for improved performance.	✓	✓	See detail

Portability and migration options

The options in Table 13 can help you maintain application behavior compatibility on past, current, and future hardware, operating systems and compilers, or help move your applications to the Enterprise Metal C for z/OS compiler with minimal change.

Table 13. Portability and migration options

Option	Description	Compile	IPA Link	More Information
CONVLIT	Turns on string literal code page conversion.	✓		See detail
UPCONV	Specifies whether the `unsigned` specification is preserved when integral promotions are performed.	✓	✓	See detail

Compiler customization options

The options in Table 14 allow you to specify alternate locations for configuration files, and internal compiler operation. You should only need to use these options in specialized installation or testing scenarios.

Table 14. Compiler customization options

Option	Description	Compile	IPA Link	More Information
MEMORY	Improves compile-time performance by using a memory file in place of a temporary work file, if possible.	✓	✓	See detail
OPTFILE	Specifies where the compiler should look for additional compiler options.	✓	✓	See detail

Description of compiler options

The following sections describe the compiler options and their usage. Compiler options are listed alphabetically.

For each option, the following information is provided:

Category
: The functional category to which the option belongs is listed here.

Pragma equivalent
: Many compiler options allow you to use an equivalent pragma directive to apply the option's functionality within the source code, limiting the scope of the option's application to a single source file, or even selected sections of code. Where an option supports the **#pragma options** (*option_name*) and/or **#pragma *name*** form of the directive, this is indicated.

Purpose
: This section provides a brief description of the effect of the option (and equivalent pragmas), and why you might want to use it.

Syntax
: This section provides the syntax for the option. The abbreviation of the option is used in the syntax diagram. You can also specify the option using its full name.

Defaults
In most cases, the default option setting is clearly indicated in the syntax diagram. However, for many options, there are multiple default settings, depending on other compiler options in effect. This section indicates the different defaults that may apply.

Parameters
This section describes the suboptions that are available for the option.

Usage
This section describes any rules or usage considerations you should be aware of when using the option. These can include restrictions on the option's applicability, valid placement of pragma directives, precedence rules for multiple option specifications, and so on.

IPA effects
Where appropriate, provides information on the effect of the option during the IPA compile and/or IPA link steps.

Predefined macros
Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section.

Examples
Where appropriate, examples of the command-line syntax are provided in this section.

Related information
Where appropriate, provides cross-references to related information.

AGGRCOPY

Category

Optimization and tuning

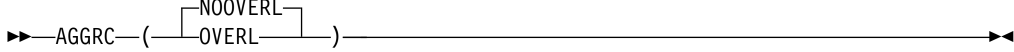
Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions, which can improve performance.

Syntax



Defaults

AGGRCOPY(NOOVERLAP)

Parameters

OVERLAP
Specifies that the source and destination in a structure assignment might

overlap in memory. Programs that do not comply to the ANSI C standard as it pertains to non-overlap of source and destination assignment may need to be compiled with the OVERLAP suboption.

NOOVERLAP
Instructs the compiler to assume that the source and destination for structure and union assignments do not overlap. This assumption lets the compiler generate faster code.

Usage

The AGGRCOPY option instructs the compiler on whether or not the source and destination assignments for structures can overlap. They cannot overlap according to ISO Standard C rules. For example, in the assignment `a = b;`, where `a` and `b` are `structs`, `a` is the destination and `b` is the source.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step accepts the AGGRCOPY option, but ignores it.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition.

The value of the AGGRCOPY option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different AGGRCOPY settings may be combined in the same partition. When this occurs, the resulting partition is always set to AGGRCOPY(OVERLAP).

Predefined macros

None.

AGGREGATE | NOAGGREGATE

Category

Listings, messages, and compiler information

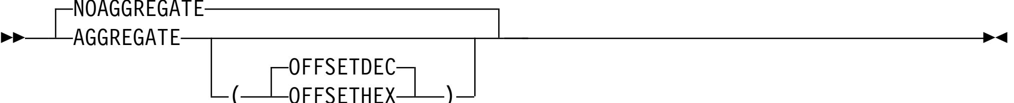
Pragma equivalent

`#pragma options (aggregate)`, **`#pragma options (noaggregate)`**

Purpose

Lists structures and unions, and their sizes.

Syntax



Defaults

NOAGGREGATE

For AGGREGATE, the default is OFFSETDEC.

Parameters

OFFSETDEC
Lists the structure member offsets in decimal format.

OFFSETHEX
Lists the structure member offsets in hexadecimal format.

Usage

Specifying AGGREGATE with no suboption is equivalent to specifying AGGREGATE(OFFSETDEC).

When the AGGREGATE compiler option is in effect, the compiler includes a layout of all `struct` or `union` types in the compiler listing.

Depending on the `struct` or `union` declaration, the maps are generated as follows:

- If the typedef name refers to a `struct` or `union`, one map is generated for the `struct` or `union` for which the typedef name refers to. If the typedef name can be qualified with the `_Packed` keyword, then a packed layout of the `struct` or `union` is generated as well. Each layout map contains the offset and lengths of the structure members and the union members. The layout map is identified by the `struct/union` tag name (if one exists) and by the typedef names.
- If the `struct` or `union` declaration has a tag, two maps are created: one contains the unpacked layout, and the other contains the packed layout. The layout map is identified by the `struct/union` tag name.
- If the `struct` or `union` declaration does not have a tag, one map is generated for the `struct` or `union` declared. The layout map is identified by the variable name that is specified on the `struct` or `union` declaration.

Predefined macros

None.

ANSIALIAS | NOANSIALIAS

Category

Optimization and tuning

Pragma equivalent

`#pragma options (ansialias)`, **`#pragma options (noansialias)`**

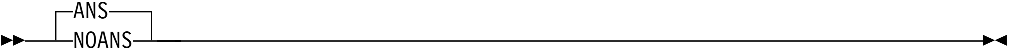
Purpose

Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C standard, and can therefore be compiled with higher performance optimization of the generated code.

When ANSIALIAS is in effect, you are making a promise to the compiler that your source code obeys the constraints in the ISO standard. On the basis of using this compiler option, the compiler front end passes aliasing information to the optimizer, which performs optimization accordingly.

When NOANSIALIAS is in effect, the optimizer assumes that a given pointer of a given type can point to an external object or any object whose address is taken, regardless of type. This assumption creates a larger aliasing set at the expense of performance optimization.

Syntax



Defaults

ANSIALIAS

Usage

When type-based aliasing is used during optimization, the optimizer assumes that pointers can only be used to access objects of the same type.

Type-based aliasing improves optimization in the following ways.
- It provides precise knowledge of what pointers can and cannot point at.
- It allows more loads to memory to be moved up and stores to memory moved down past each other, which allows the delays that normally occur in the original written sequence of statements to be overlapped with other tasks. These re-arrangements in the sequence of execution increase parallelism, which is desirable for optimization.
- It allows the removal of some loads and stores that otherwise might be needed in case those values were accessed by unknown pointers.
- It allows more identical calculations to be recognized ("commoning").
- It allows more calculations that do not depend on values modified in a loop to be moved out of the loop ("code motion").
- It allows better optimization of parameter usage in inlined functions.

Simplified, the rule is that you cannot safely dereference a pointer that has been cast to a type that is not closely related to the type of what it points at. The ISO C standard defines the closely related types.

The following are not subject to type-based aliasing:
- Types that differ only in reference to whether they are signed or unsigned. For example, a pointer to a signed int can point to an `unsigned int`.
- Character pointer types (`char`, `unsigned char`).

- Types that differ only in their `const` or `volatile` qualification. For example, a pointer to a `const int` can point to an `int`.

Enterprise Metal C for z/OS compiler often exposes type-based aliasing violations that other compilers do not.

In addition to the specific optimizations to the lines of source code that can be obtained by compiling with the ANSIALIAS compiler option, other benefits and advantages, which are at the program level, are described below:

- It reduces the time and memory needed for the compiler to optimize programs.
- It allows a program with a few coding errors to compile with optimization, so that a relatively small percentage of incorrect code does not prevent the optimized compilation of an entire program.
- It positively affects the long-term maintainability of a program by supporting ISO-compliant code.

It is important to remember that even though a program compiles, its source code may not be completely correct. When you weigh tradeoffs in a project, the short-term expedience of getting a successful compilation by forgoing performance optimization should be considered with awareness that you may be nurturing an incorrect program. The performance penalties that exist today could worsen as the compilers that base their optimization on strict adherence to ISO rules evolve in their ability to handle increased parallelism.

The ANSIALIAS compiler option only takes effect if the OPTIMIZE option is in effect.

If you specify LANGLVL(COMMONC), the ANSIALIAS option is automatically turned off. If you want ANSIALIAS turned on, you must explicitly specify it. Using LANGLVL(COMMONC) and ANSIALIAS together may have undesirable effects on your code at a high optimization level. See “LANGLVL” on page 79 for more information on LANGLVL(COMMONC).

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Although type-based aliasing does not apply to the `volatile` and `const` qualifiers, these qualifiers are still subject to other semantic restrictions. For example, casting away a `const` qualifier might lead to an error at run time.

IPA effects

If the ANSIALIAS option is specified, then the IPA link step phase will take advantage of the knowledge that the program will adhere to the standard C aliasing rules in order to improve its variable aliasing calculations.

Predefined macros

None.

Examples

The following example executes as expected when compiled unoptimized or with the NOANSIALIAS option; it successfully compiles optimized with ANSIALIAS, but does not necessarily execute as expected. On non-IBM compilers, the following code may execute properly, even though it is incorrect.

```
1  extern int y = 7.;
2
3  void main() {
4       float x;
5       int i;
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.\n", i, x);
9  }
```

In this example, the value in object x of type float has its stored value accessed via the expression * (int *) &x. The access to the stored value is done by the * operator, operating on the expression (int *) &x. The type of that expression is (int *), which is not covered by the list of valid ways to access the value in the ISO standard, so the program violates the standard.

When ANSIALIAS (the default) is in effect, the compiler front end passes aliasing information to the optimizer that, in this case, an object of type float could not possibly be pointed to by an (int *) pointer (that is, that they could not be aliases for the same storage). The optimizer performs optimization accordingly. When it compares the instruction that stores into x and the instruction that loads out of *(int *), it believes it is safe to put them in either order. Doing the load before the store will make the program run faster, so it interchanges them. The program becomes equivalent to:

```
1  extern int y = 7.;
2
3  void main() {
4       float x;
5       int i;
6       int temp;
7       temp = *(int *) &x;  /* uninitialized */
8       x = y;
9       i = temp;
10      printf("i=%d. x=%f.\n", i, x);
9  }
```

The value stored into variable i is the old value of x, before it was initialized, instead of the new value that was intended. IBM compilers apply some optimizations more aggressively than some other compilers so correctness is more important.

ARCHITECTURE

Category

Optimization and tuning

Pragma equivalent

#pragma options (architecture)

Purpose

Specifies the machine architecture for which the executable program instructions are to be generated.

Syntax

```
►►—ARCH—(—n—)——————————————————————————————————————————►◄
```

Defaults

ARCH(10)

Parameters

n Specifies the group to which a model number belongs.

The following groups of models are supported:

0 Produces code that is executable on all models.

1 Produces code that uses instructions available on the following system machine models:

- 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900
- 9021-xx1 and 9021-xx2
- 9672-Rx1, 9672-Rx2 (G1), 9672-Exx, and 9672-Pxx

Specifically, these ARCH(1) machines and their follow-ons add the C Logical String Assist hardware instructions. These instructions are exploited by the compiler, when practical, for a faster and more compact implementation of some functions, for example, `strcmp()`.

2 Produces code that uses instructions available on the following system machine models:

- 9672-Rx3 (G2), 9672-Rx4 (G3), 9672-Rx5 (G4), and 2003

Specifically, these ARCH(2) machines and their follow-ons add the Branch Relative instruction (Branch Relative and Save - BRAS), and the halfword Immediate instruction set (for example, Add Halfword Immediate - AHI) which may be exploited by the compiler for faster processing.

3 Produces code that uses instructions available on the 9672-xx6 (G5), 9672-xx7 (G6), and follow-on models.

Specifically, these ARCH(3) machines and their follow-ons add a set of facilities for IEEE floating-point representation, as well as 12 additional floating-point registers and some new floating-point support instructions that may be exploited by the compiler.

Note that ARCH(3) is required for execution of a program that specifies the FLOAT(IEEE) compiler option. However, if the program is executed on a physical processor that does not actually provide these ARCH(3) facilities, any program check (operation or specification exception), resulting from an attempt to use features associated with IEEE floating point or the additional floating point registers, will be intercepted by the underlying operating system, and simulated by software. There will be a significant performance degradation for the simulation.

4 Produces code that uses instructions available on the 2064-xxx (z900) and 2066-xxx (z800) models in ESA/390 mode.

Specifically, the following instructions are used for long long operations:

- 32-bit Add-With-Carry (ALC, ALCR) for long long addition (rather than requiring a branch sequence)

- 32-bit Subtract-With-Borrow (SLB, SLBR) for long long subtraction (rather than requiring a branch sequence)
- Inline sequence with 32-bit Multiply-Logical (ML, MLR) for long long multiplication (rather than calling `@@MULI64`)

5
Produces code that uses instructions available on the 2064-xxx (z900) and 2066-xxx (z800) models in z/Architecture mode.

Specifically, ARCH(5) is the minimum requirement for execution of a program in 64-bit mode. If you explicitly set ARCH to a lower level, the compiler will issue a warning message and ignore your setting. ARCH(5) specifies the target machine architecture and the application can be either 31-bit or 64-bit.

6
Produces code that uses instructions available on the 2084-xxx (z990) and 2086-xxx (z890) models in z/Architecture mode.

Specifically, these ARCH(6) machines and their follow-ons add the long-displacement facility. For further information on the long-displacement facility, refer to *z/Architecture Principles of Operation*.

7
Produces code that uses instructions available on the 2094-xxx (IBM System z9® Business Class) and 2096-xxx (IBM System z9 Business Class) models in z/Architecture mode.

Specifically, these ARCH(7) machines and their follow-ons add instructions supported by the extended-immediate facility, which may be exploited by the compiler. For further information on these facilities, refer to *z/Architecture Principles of Operation*.

8
Produces code that uses instructions available on the 2097-xxx (IBM System z10® Enterprise Class) and 2098-xxx (IBM System z10 Business Class) models in z/Architecture mode.

Specifically, these ARCH(8) machines and their follow-ons add instructions supported by the general instruction extensions facility, which may be exploited by the compiler. For further information on these facilities, refer to *z/Architecture Principles of Operation*.

9
Produces code that uses instructions available on the 2817-xxx (IBM zEnterprise® 196 (z196)) and 2818-xxx (IBM zEnterprise 114 (z114)) models in z/Architecture mode.

Specifically, these ARCH(9) machines and their follow-ons add instructions supported by the high-word facility, the interlocked-access facility, the load/store-on-condition facility, the distinct-operands-facility and the population-count facility. For further information about these facilities, see *z/Architecture Principles of Operation*.

10
Is the default value. Produces code that uses instructions available on the 2827-xxx (IBM zEnterprise EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models in z/Architecture mode.

Specifically, these ARCH(10) machines and their follow-ons add instructions supported by the execution-hint facility, the load-and-trap facility, the miscellaneous-instruction-extension facility, and the transactional-execution facility. For further information about these facilities, see *z/Architecture Principles of Operation*.

11
Produces code that uses instructions available on the 2964-xxx (IBM z13™ (z13)) and the 2965-xxx (IBM z13s (z13s)) models in z/Architecture mode.

Specifically, these ARCH(11) machines and their follow-ons add instructions supported by the vector facility, the decimal floating point packed conversion facility, and the load/store-on-condition facility 2. The VECTOR option is required for the compiler to use the vector facility. For further information about these facilities, see *z/Architecture Principles of Operation*.

12 Produces code that uses instructions available on the 3906-xxx (IBM z14) and 3907-xxx (IBM z14 ZR1) models in z/Architecture mode.

Specifically, these ARCH(12) machines and their follow-ons add instructions supported by the vector enhancement facility 1, the vector packed decimal facility, and the miscellaneous instruction extension facility 2. The VECTOR option is required for the compiler to use the vector enhancement facility 1 and vector packed decimal facility. For further information about these facilities, see *z/Architecture Principles of Operation*.

Usage

When ARCHITECTURE is in effect, the compiler selects the instruction set available during the code generation of your program based on the specified machine architecture.

Specifying a higher ARCH level generates code that uses newer and faster instructions instead of the sequences of common instructions.

Notes:

1. Your application will not run on a lower architecture processor than what you specified using the ARCH option. Use the ARCH level that matches the lowest machine architecture where your program will run.
2. Code that is compiled at ARCH(1) runs on machines in the ARCH(1) group and later machines, including those in the ARCH(2) and ARCH(3) groups. It may not run on earlier machines. Code that is compiled at ARCH(2) may not run on ARCH(1) or earlier machines. Code that is compiled at ARCH(3) may not run on ARCH(2) or earlier machines.
3. For the system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RX4, not just 9672-RX4.

If you specify a group that does not exist or is not supported, the compiler uses the default, and issues a warning message.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition.

If you specify the ARCH option on the IPA link step, it uses the value of that option for all partitions. The IPA link step Prolog and all Partition Map sections of the IPA link step listing display that value.

If you do not specify the option on the IPA link step, the value used for a partition depends on the value that you specified for the IPA compile step for each compilation unit that provided code for that partition. If you specified the same value for each compilation unit, the IPA link step uses that value. If you specified different values, the IPA link step uses the lowest level of ARCH.

The level of ARCH for a partition determines the level of TUNE for the partition.

The Partition Map section of the IPA link step listing, and the object module display the final option value for each partition. If you override this option on the IPA link step, the Prolog section of the IPA link step listing displays the value of the option.

The Compiler Options Map section of the IPA link step listing displays the option value that you specified for each IPA object file during the IPA compile step.

Predefined macros

__ARCH__ is predefined to the integer value of the ARCH compiler option.

Related information

- Use the ARCH option with the TUNE option. For more information about the interaction between ARCH and TUNE, see “TUNE” on page 145.
- “VECTOR | NOVECTOR” on page 151

ARGPARSE | NOARGPARSE

Category

Object code control

Pragma equivalent

None.

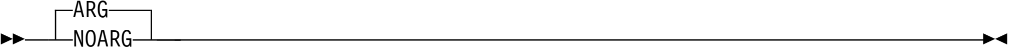
Purpose

Parses arguments provided on the invocation line.

When ARGPARSE is in effect, arguments supplied to your program on the invocation line are parsed and passed to the `main()` routine in the C argument format, commonly `argc` and `argv`. `argc` contains the argument count, and `argv` contains the tokens after the command processor has parsed the string.

When NOARGPARSE is in effect, arguments on the invocation line are not parsed, `argc` has a value of `2`, and `argv` contains a pointer to the string.

Syntax



Defaults

ARGPARSE

Usage

Note: NOARGPARSE is ignored for the following programs:

- Programs that use `spawn()` or `exec()`.
- Programs that are started by the z/OS UNIX System Services shell or by the BPXBATCH utility.
- METAL programs that are dubbed.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

If you specify this option for both the IPA Compile and the IPA link steps, the setting on the IPA link step overrides the setting on the IPA compile step. This applies whether you use ARGPARSE and NOARGPARSE as compiler options, or specify them using the **`#pragma runopts`** directive on the IPA compile step.

If you specified ARGPARSE on the IPA compile step, you do not need to specify it again on the IPA link step to affect that step. The IPA link step uses the information generated for the compilation unit that contains the `main()` function. If it cannot find a compilation unit that contains `main()`, it uses the information generated by the first compilation unit that it finds.

Predefined macros

None.

ARMODE | NOARMODE

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies that all functions in the C source file will operate in access-register (AR) mode.

When ARMODE is in effect, all functions in the compilation unit will be compiled in AR mode. AR mode functions can access data stored in additional data spaces supported by IBM Z®.

To override the effect of the ARMODE option and selectively re-set particular functions to be in non-AR mode (or primary address space control mode), use `__attribute__((noarmode))`. For more information on this attribute, see The armode | noarmode type attribute in *Enterprise Metal C for z/OS Language Reference* and *Enterprise Metal C for z/OS Programming Guide and Reference*.

When NOARMODE is in effect, functions are not in AR mode unless `__attribute__((armode))` is specifically specified for the functions.

Syntax

Defaults

NOARMODE

Usage

If the ARMODE compiler option is specified, all functions in the compilation unit will be compiled in AR mode.

Note: If the `armode` attribute is specified on a function in a compilation unit, it overrides the compiler option.

AR mode enables a program to manipulate large amounts of data in memory by using `__far` pointers. This means that a program working with a large table, for example, would not need to use temporary disk files to move the data in and out of disk storage. It also means that program logic can be less complicated, easier to maintain, and less error prone. Currently, only assembler can make use of AR Mode directly.

Predefined macros

None.

Related information

For more information on `__far` pointers, see *Enterprise Metal C for z/OS Language Reference*.

ASM | NOASM

Category

Language element control

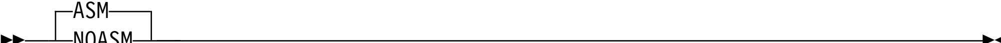
Pragma equivalent

None.

Purpose

Enables inlined assembly code inside C programs.

Syntax



Default

ASM

Usage

Specify the ASM compiler option to instruct the compiler to recognize the `__asm` and `__asm__` keywords (as well as the `asm` keyword).

If the NOASM option is in effect, any `__asm` or `__asm__` statements will be treated as identifiers.

The ASM option implies the KEYWORD(asm) option.

When compiling programs with inlined assembly code, you must be aware of the following constraints to the source code:

- User labels in inlined assembly code are not supported by the compiler. If the labels are necessary, you must ensure that each label is uniquely defined because the inlined assembly code might get duplicated by various optimization phases, and therefore user labels might be defined multiple times when they are presented to the assembler.
- HLASM symbols within another asm block are not supported.
- If an `asm` statement is used to define data, it cannot contain assembly instructions for other purposes.
- Only `asm` statements that are used to define data can exist in global scope.
- Each assembly statement can define only one variable.
- The symbol used in the assembly statement must be unique within the scope of the source file and be valid according to the assembler's requirements.
- Referencing an external symbol directly without going through the operand list is not supported.
- Using registers that are reserved (for example, killing a register used by the linkage) is not supported.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The ASM option needs to be specified again in the IPA link step.

Predefined macros

__IBM_ASM_SUPPORT is predefined to 1 if ASM is specified.

Related information

- Inline assembly statements (IBM extension) in *Enterprise Metal C for z/OS Language Reference*

ASMDATASIZE

Category

Object code control

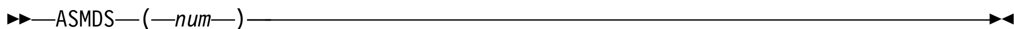
Pragma equivalent

None.

Purpose

Provides the default data area size for the data areas defined by user-supplied assembly statements.

Syntax



Defaults

ASMDATASIZE(256)

Parameters

num
It is a positive integer number. The default value is 256.

IPA effects

The ASMDATASIZE option is ignored in the IPA link step. The IPA link step uses the data area size from the IPA compile step.

ASSERT(RESTRICT) | ASSERT(NORESTRICT)

Category

Optimization and tuning

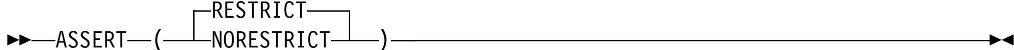
Pragma equivalent

None.

Purpose

Enables optimizations for restrict qualified pointers.

Syntax



Defaults

ASSERT(RESTRICT)

Parameters

RESTRICT
Optimizations based on restrict qualified pointers are enabled.

NORESTRICT
Optimizations based on restrict qualified pointers are disabled.

Usage

Restrict qualified pointers were introduced in the C99 Standard and provide exclusive initial access to the object that they point to. This means that two restrict qualified pointers, declared in the same scope, designate distinct objects and thus should not alias each other (in other words, they are disjoint). The compiler can use this aliasing in optimizations that may lead to additional performance gains.

Optimizations based on restrict qualified pointers will occur unless the user explicitly disables them with the option ASSERT(NORESTRICT).

ASSERT(RESTRICT) does not control whether the keyword restrict is a valid qualifier or not. Syntax checking of the restrict qualifier is controlled by the language level or KEYWORD option.

You are responsible for ensuring that if a restrict pointer *p* references an object *A*, then within the scope of *p*, only expressions based on the value of *p* are used to access *A*. A violation of this rule is not diagnosed by the compiler and may result in incorrect results. This rule only applies to ASSERT(RESTRICT).

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- “LANGLVL” on page 79
- “KEYWORD | NOKEYWORD” on page 78

BITFIELD(SIGNED) | BITFIELD(UNSIGNED)

Category

Floating-point and integer control

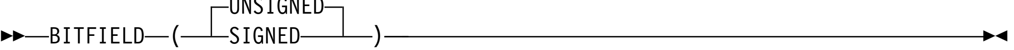
Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax



Defaults

BITFIELD(UNSIGNED)

Parameters

SIGNED
Bit fields are signed.

UNSIGNED
Bit fields are unsigned.

Usage

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

CHARS(SIGNED) | CHARS(UNSIGNED)

Category

Floating-point and integer control

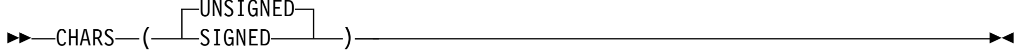
Pragma equivalent

#pragma chars

Purpose

Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.

Syntax



Defaults

CHARS(UNSIGNED)

Parameters

UNSIGNED
Variables defined as `char` are treated as `unsigned char`.

SIGNED
Variables defined as `char` are treated as `signed char`.

Usage

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

- _CHAR_SIGNED is predefined to 1 when the CHARS(SIGNED) compiler option is in effect; otherwise it is undefined.
- _CHAR_UNSIGNED is predefined to 1 when the CHARS(UNSIGNED) compiler option is in effect; otherwise it is undefined.

COMPACT | NOCOMPACT

Category

Optimization and tuning

Pragma equivalent

```
#pragma option_override(subprogram_name, "OPT(COMPACT)")
```

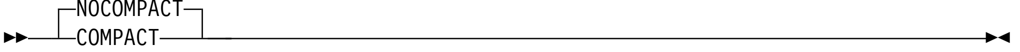
Purpose

Avoids optimizations that increase object file size.

When the COMPACT option is in effect, the compiler favors those optimizations that tend to limit object file size.

When the NOCOMPACT option is in effect, the compiler might use optimizations that result in an increased object file size.

Syntax



Defaults

NOCOMPACT

Usage

During optimizations that are performed as part of code generation, for both NOIPA and IPA, choices must be made between those optimizations that tend to result in faster but larger code and those that tend to result in smaller but slower code. The COMPACT option influences these choices.

Because of the interaction between various optimizations, code that is compiled with the COMPACT option might not always generate smaller code and data.

When COMPACT is specified, as examples, it has the following effects:

- Not all subprograms are inlined. To determine the final status of inlining, generate and check the inline report.
- The compiler might not generate inline code for some built-in versions of the C library and the Metal C runtime library functions.

To evaluate the use of the COMPACT option for your application:

- Compare the size of the objects generated with COMPACT and NOCOMPACT
- Compare the size of the modules generated with COMPACT and NOCOMPACT

- Compare the execution time of a representative workload with COMPACT and NOCOMPACT

If the objects and modules are smaller with an acceptable change in execution time, then you can consider the benefit of using COMPACT.

As new optimizations are added to the compiler, the behavior of the COMPACT option might change. You should reevaluate the use of this option for each new release of the compiler and when you change the application code.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

During a compilation with IPA Compile-time optimizations active, any subprogram-specific COMPACT option that is specified by **#pragma option_override(subprogram_name, "OPT(COMPACT)")** directives will be retained.

The IPA compile step generates information for the IPA link step.

If you specify the COMPACT option for the IPA link step, it sets the compilation unit values of the COMPACT option that you specify. The IPA link step Prolog listing section will display the value of this option.

If you do not specify COMPACT option in the IPA link step, the setting from the IPA compile step for each compilation unit will be used.

In either case, subprogram-specific COMPACT options will be retained.

The IPA link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same COMPACT setting.

The COMPACT setting for a partition is set to the specification of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same COMPACT setting. A NOCOMPACT subprogram is placed in a NOCOMPACT partition, and a COMPACT subprogram is placed in a COMPACT partition.

The option value that you specified for each IPA object file on the IPA compile step appears in the IPA link step Compiler Options Map listing section.

The Partition Map sections of the IPA link step listing and the object module END information section display the value of the COMPACT option. The Partition Map also displays any subprogram-specific COMPACT values.

Predefined macros

None.

COMPRESS | NOCOMPRESS

Category

Object code control

Pragma equivalent

None.

Purpose

Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.

Syntax

Defaults

NOCOMPRESS

Usage

Function names are used by the dump service to provide you with meaningful diagnostic information when your program encounters a fatal program error. Without these function names, the reports generated by these services and tools may not be complete.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

If you specify the COMPRESS option for the IPA link step, it uses the value of the option that you specify. The IPA link step Prolog listing section will display the value of the option that you specify.

If you do not specify COMPRESS option in the IPA link step, the setting from the IPA compile step will be used.

The IPA link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same COMPRESS setting.

The COMPRESS setting for a partition is set to the specification of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same COMPRESS setting. A NOCOMPRESS mode

subprogram is placed in a NOCOMPRESS partition, and a COMPRESS mode subprogram is placed in a COMPRESS partition.

The option value that you specified for each IPA object file on the IPA compile step appears in the IPA link step Compiler Options Map listing section.

The Partition Map sections of the IPA link step listing and the object module END information section display the value of the COMPRESS option.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- "DEBUG | NODEBUG" on page 46

CONVLIT | NOCONVLIT

Category

Portability and migration

Pragma equivalent

#pragma convlit

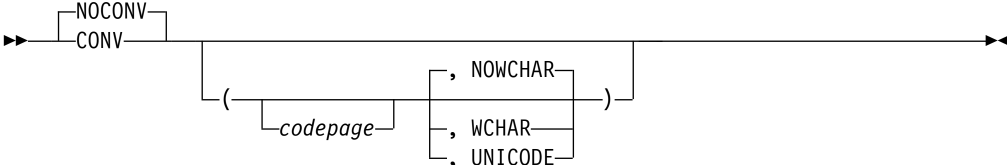
Purpose

Turns on string literal code page conversion.

When the CONVLIT option is in effect, the compiler changes the assumed code page for character and string literals within the compilation unit.

When the NOCONVLIT option is in effect, the default code page, or the code page specified by the LOCALE option is used.

Syntax



Defaults

NOCONVLIT(, NOWCHAR)

Parameters

codepage
: You can use an optional suboption to specify the code page that you want to use for string literals.

NOWCHAR
: The default is NOWCHAR. Only wide character constants and string literals made up of single byte character set (SBCS) characters are converted. If there are any shift-out (SO) and shift-in (SI) characters in the literal, the compilation will end with an error message.

WCHAR
: Instructs the compiler to change the code page for wide character constants and string literals declared with the L'' or L"" prefix.

UNICODE
: The compiler interprets the CONVLIT(, UNICODE) suboption as a request to convert the wide string literals and wide character constants (`wchar_t`) to Unicode (UCS-2) regardless of the code page used for conversion of string literals and character constants (`char`). The conversion is supported for wide string literals and wide character constants that are coded using characters from the basic character set defined by the *Programming languages - C (ISO/IEC 9899:1999)* standard. The behavior is undefined if wide string literals and wide character constants are coded using characters outside the basic character set.

Usage

The CONVLIT option affects all the source files that are processed within a compilation unit, including user header files and system header files. All string literals and character constants within a compilation unit are converted to the specified *codepage* unless you use **#pragma convlit(suspend)** and **#pragma convlit(resume)** to exclude sections of code from conversion.

The CONVLIT option only affects string literals within the compilation unit. The following determines the code page that the program uses:

- If you specified a LOCALE, the remainder of the program will be in the code page that you specified with the LOCALE option.
- If you specify the CONVLIT option with empty sub option list, CONVLIT() or `-qconvlit=`, the compiler preserves any previous settings of the suboptions. It will not use the default code page, or the code page specified by the LOCALE option. For example, **-Wc,'CONVLIT(IBM-273) CONVLIT()'** is interpreted as **CONVLIT(IBM-273,NOWCHAR)**.

The CONVLIT option does not affect the following types of string literals:

- literals in the `#include` directive
- literals in the `#pragma` directive
- literals used to specify linkage, for example, `extern "C"`
- literals used for the `__func__` variables

If **#pragma convlit(suspend)** is in effect, no string literals or character constants (wide included) will be converted.

If **#pragma convert** is in effect, string literals and character constants will be converted, but wide string literals and wide character constants are not affected by **#pragma convert**, even when the CONVLIT(, UNICODE) suboption is specified.

If you specify PPONLY with CONVLIT, the compiler ignores CONVLIT.

If you specify the CONVLIT option, the *codepage* appears after the locale name and locale code set in the Prolog section of the listing. The option appears in the END card at the end of the generated object module.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Notes:

1. Although you can continue to use the __STRING_CODE_SET__ macro, you should use the CONVLIT option instead. If you specify both the macro and the option, the compiler diagnoses it and uses the option regardless of the order in which you specify them
2. The **`#pragma convert`** directive provides similar functionality to the CONVLIT option. It has the advantage of allowing more than one character encoding to be used for string literals in a single compilation unit.

IPA effects

The CONVLIT option only controls processing for the IPA step for which you specify it.

During the IPA compile step, the compiler uses the code page that is specified by the CONVLIT option to convert the character string literals.

Predefined macros

None.

Examples

The result of the following specifications is the same:

- `NOCONV(IBM-1027) CONV`
- `CONV(IBM-1027)`

Related information

For more information on the LOCALE compiler option, see "LOCALE | NOLOCALE" on page 85.

CSECT | NOCSECT

Category

Object code control

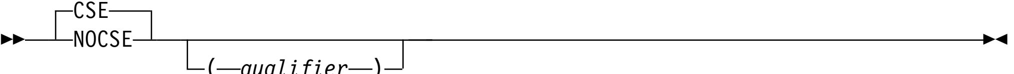
Pragma equivalent

`#pragma csect`

Purpose

Instructs the compiler to generate CSECT names in the output object module.

Syntax



Defaults

CSECT

Parameters

qualifier
Enables the compiler to generate long CSECT names.

Usage

When the CSECT option is in effect, the compiler should ensure that the code, static data, and test sections of your object module are named. Use this option if you will be using SMP/E to service your product and to aid in debugging your program.

When you specify CSECT(*qualifier*) and the NOGOFF option is in effect, the LONGNAME option is assumed.

For GOFF, both the NOLONGNAME and LONGNAME options are supported.

The CSECT option names sections of your object module differently depending on whether you specified CSECT with or without a qualifier.

If you specify the CSECT option without the *qualifier* suboption, the CSECT option names the code, static data, and test sections of your object module as *csectname*, where *csectname* is one of the following:

- The member name of your primary source file, if it is a PDS member
- The low-level qualifier of your primary source file, if it is a sequential data set
- The source file name with path information and the right-most extension information removed, if it is a z/OS UNIX file.
- For NOGOFF, if the NOLONGNAME option is in effect, then the *csectname* is truncated to 8 characters long starting from the left. For GOFF, the full *csectname* is always used.

code CSECT
Is named with *csectname* name in uppercase.

data CSECT
Is named with *csectname* in lower case.

test CSECT
The test CSECT is the static CSECT name with the prefix $. If the static CSECT name is 8 characters long, the right-most character is dropped and the compiler issues an informational message except in the case of GOFF. The test CSECT name is always truncated to 8 characters.

For example, if you compile `/u/cricket/project/mem1.ext.c`:

- with the options NOGOFF and CSECT, the test CSECT will have the name `$mem1.ex`

- with the options GOFF and CSECT, the test CSECT will have the name `$mem1.ext`

If you specify the CSECT option with the *qualifier* suboption, the CSECT option names the code, static data, and test sections of your object module as *qualifier#basename#suffix*, where:

qualifier
: Is the suboption you specified as a *qualifier*

basename
: Is one of the following:
  - The member name of your primary source file, if it is a PDS member
  - There is no basename, if your primary source file is a sequential data set or instream JCL
  - The source file name with path information and the right-most extension information removed, if it is a z/OS UNIX file

suffix
: Is one of the following:

  C For code CSECT

  S For static CSECT

  T For test CSECT

Notes:

1. If the *qualifier* suboption is longer than 8 characters, you must use the binder.
2. The *qualifier* suboption takes advantage of the capabilities of the binder.
3. The # that is appended as part of the #C, #S, or #T suffix is not locale-sensitive.
4. The string that is specified as the *qualifier* suboption has the following restrictions:
   - Leading and trailing blanks are removed
   - You can specify a string of any length. However if the complete CSECT name exceeds 1024 bytes, it is truncated starting from the left.
5. If the source file is either sequential or instream in your JCL, you must use the **`#pragma csect`** directive to name your CSECT. Otherwise, you may receive an error message at bind time.

The CSECT names for all the sections (including the code, static data and test sections) must conform to the following rules:

- The first character must be an alphabetic character. An alphabetic character is a letter from A through Z, or from a through z, or _, $(code point X'5B'), #(code point X'7B') or @(code point X'7C'). The other characters in the CSECT name may be alphabetic characters, digits, or a combination of the two.
- No other special characters may be included in the CSECT name.
- No spaces are allowed in the CSECT name.
- No double-byte data is allowed in the CSECT name.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

For the IPA link step, this option has the following effects:

1. If you specify the CSECT option, the IPA link step names all of the CSECTs that it generates.

   The IPA link step determines whether the IPA Link control file contains CSECT name prefix directives. If you did not specify the directives, or did not specify enough CSECT entries for the number of partitions, the IPA link step automatically generates CSECT name prefixes for the remaining partitions, and issues an error diagnostic message each time.

   The form of the CSECT name that IPA Link generates depends on whether the CSECT or CSECT(*qualifier*) format is used.
2. If you do not specify the CSECT option, but you have specified CSECT name prefix directives in the IPA Link control file, the IPA link step names all CSECTs in a partition. If you did not specify enough CSECT entries for the number of partitions, the IPA link step automatically generates a CSECT name prefix for each remaining partition, and issues a warning diagnostic message each time.
3. If you do not specify the CSECT option, and do not specify CSECT name prefix directives in the IPA Link control file, the IPA link step does not name the CSECTs in a partition.

The IPA link step ignores the information that is generated by **`#pragma csect`** on the IPA compile step.

Predefined macros

None.

Examples

The *qualifier* suboption of the CSECT option allows the compiler to generate long CSECT names.

When you specify CSECT(*qualifier*), the code, data, and test CSECTs are always generated.

If you use `CSECT("")` or `CSECT()`, the CSECT name has the form *basename#suffix*, where *basename* is:
- `@Sequential@` for a sequential data set
- `@InStream@` for instream JCL

DEBUG | NODEBUG

Category

Error checking and debugging

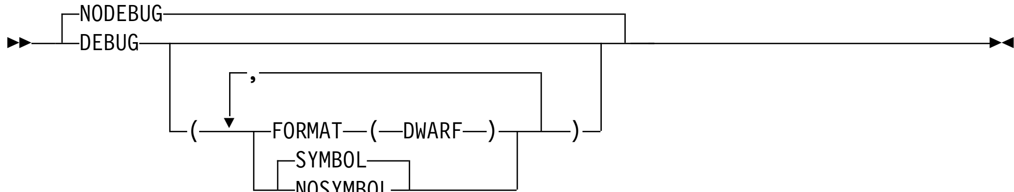
Pragma equivalent

None.

Purpose

Instructs the compiler to generate debugging information.

Syntax



Defaults

- NODEBUG
- For SYMBOL, the default is SYMBOL.

Parameters

FORMAT

The DWARF suboption produces debugging information in the DWARF Version 4 debugging information format, stored in the file specified by the FILE suboption, or in GOFF NOLOAD classes when the NOFILE suboption is specified.

SYMBOL

This option provides you with access to variable and other symbol information. For optimized code, the results are not always well-defined for every variable because the compiler might have optimized away their use.

Note: The default of this suboption is DEBUG(SYMBOL), but when the HOT or IPA option is used with DEBUG, DEBUG(NOSYMBOL) is forced.

Usage

When the DEBUG option is in effect, the compiler generates debugging information based on the DWARF Version 4 debugging information format, which has been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format.

If you specify the INLINE and DEBUG(FORMAT(DWARF)) compiler options when OPTIMIZE is in effect, the inline debugging information is generated for inline procedures as well as parameters and local variables of inline procedures.

If you specify the INLINE and DEBUG compiler options when NOOPTIMIZE is in effect, INLINE is ignored.

When OPT(2) or OPT(3) is used with DEBUG, the DEBUG(SYMBOL) suboption is enabled by default.

In the z/OS UNIX System Services environment, **-g** forces DEBUG(FORMAT(DWARF)), NOHOT, and NOOPTIMIZE.

If you specify DEBUG(FORMAT(DWARF)), automonitor debugging information is generated to list the variables that occur on each statement of the program source file.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

For the IPA compile step, you can specify all of the DEBUG suboptions that are appropriate for the language of the code that you are compiling. However, they affect processing only if you have requested code generation, and only the conventional object file is affected.

If only IPA object is produced at the IPA compile step, the DEBUG options are accepted and ignored.

Predefined macros

None.

Examples

If you specify DEBUG and NODEBUG multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
metalc -Wc,"NODEBUG(FORMAT(DWARF))" -Wc,"DEBUG(NOSYMBOL)" hello.c
metalc -WC,"DEBUG(FORMAT(DWARF), NOSYMBOL)" hello.c
```

DEFINE

Category

Language element control

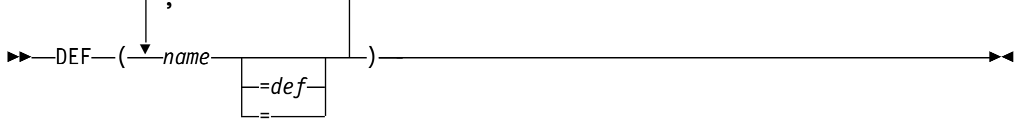
Pragma equivalent

None.

Purpose

Defines a macro as in a `#define` preprocessor directive.

Syntax



Defaults

No default user definitions.

Parameters

DEFINE(*name*)
Is equal to the preprocessor directive `#define` *name* `1`.

DEFINE(*name=def*)
Is equal to the preprocessor directive `#define` *name def*.

DEFINE(*name*=)
Is equal to the preprocessor directive `#define` *name*.

Usage

When the DEFINE option is in effect, the preprocessor macros that take effect before the compiler processes the file are defined.

You can use the DEFINE option more than once.

If the suboptions that you specify contain special characters, see "Using special characters" on page 8 for information on how to escape special characters.

Note: **metalc** passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 14, "metalc — Compiler invocation using a customizable configuration file," on page 219.

Predefined macros

To use the __STRING_CODE_SET__ macro to change the code page that the compiler uses for character string literals, you must define it with the DEFINE compiler option; for example:

```
DEFINE(__STRING_CODE_SET__="ISO8859-1")
```

Examples

Note: There is no command-line equivalent for function-like macros that take parameters such as the following:

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

DIGRAPH | NODIGRAPH

Category

Language element control

Pragma equivalent

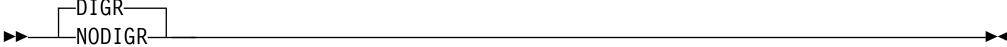
None.

Purpose

Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.

Note: A *digraph* is a combination of keys that produces a character that is not available on some keyboards.

Syntax



Defaults

DIGRAPH

Usage

Table 15 shows the digraphs that Enterprise Metal C for z/OS supports:

Table 15. Digraphs

Key Combination	Character Produced
<%	{
%>	}
<:	[
:>	]
%:	#
%%	#
%:%:	##
%%%%	##

IPA effects

The IPA link step issues a diagnostic message if you specify the DIGRAPH option on that step.

Predefined macros

__DIGRAPHS__ is predefined to 1 when the DIGRAPH compiler option is in effect.

Examples

Note: Digraphs are not replaced in string literals, comments, or character literals. For example:

```
char * s = "<%%>";   // stays "<%%>"

switch (c) {
  case '<%' : ...    // stays '<%'
  case '%>' : ...    // stays '%>'
}
```

DSAUSER | NODSAUSER

Category

Object code control

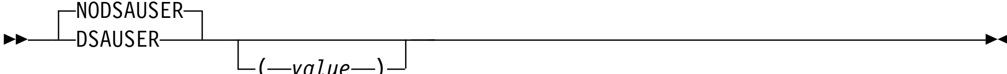
Pragma equivalent

None.

Purpose

Requests a user field to be reserved on the stack.

Syntax



Defaults

NODSAUSER

Parameter

value An integer in the range of 0 to 50.

Usage

When DSAUSER is specified, and no suboption is specified with DSAUSER, a field of the size of a pointer is reserved on the stack. The user field is a 4-byte field for AMODE 31 and an 8-byte field for AMODE 64. The user field is only allocated if the function has the user supplied prolog/epilog code.

If a *value* parameter is specified with DSAUSER, a user field with the size of *value* 32-bit words is allocated. Specifying DSAUSER with the *value* parameter requires a minimum architecture level of ARCH(6). A *value* of 0 has the same effect as NODSAUSER.

The reserved user field can be addressed by using the global set symbol &CCN_DSAUSER.

IPA effects

If the DSAUSER option is specified during any of the IPA compile steps, it is applied to all partitions created by the IPA link step. The largest *value* is used for all partitions in the IPA link step.

ENUMSIZE

Category

Floating-point and integer control

Pragma equivalent

#pragma enum

Purpose

Specifies the amount of storage occupied by enumerations

Syntax

Defaults

ENUM(SMALL)

Parameters

SMALL
Specifies that enumerations occupy a minimum amount of storage, which is either 1, 2, 4, or 8 bytes of storage, depending on the range of the `enum` constants.

INT
Specifies that enumerations occupy 4 bytes of storage and are represented by `int`.

1 Specifies that enumerations occupy 1 byte of storage.

2 Specifies that enumerations occupy 2 bytes of storage

4 Specifies that enumerations occupy 4 bytes of storage.

8 Specifies that enumerations occupy 8 bytes of storage. This suboption is only valid with LP64.

Usage

When the ENUMSIZE option is in effect, you can select the type used to represent all `enum` constants defined in a compilation unit.

The following tables illustrate the preferred sign and type for each range of `enum` constants:

Table 16. ENUM constants for C

ENUM Constants	small	1	2	4	8 *	int
0..127	unsigned char	signed char	short	int	long	int
-128..127	signed char	signed char	short	int	long	int
0..255	unsigned char	unsigned char	short	int	long	int
0..32767	unsigned short	ERROR	short	int	long	int
-32768..32767	short	ERROR	short	int	long	int
0..65535	unsigned short	ERROR	unsigned short	int	long	int
0..2147483647	unsigned int	ERROR	ERROR	int	long	int
$-2^{31}..2^{31}-1$	int	ERROR	ERROR	int	long	int

Table 16. ENUM constants for C (continued)

ENUM Constants	small	1	2	4	8 *	int
0..4294967295	unsigned int	ERROR	ERROR	unsigned int	long	ERROR
0..(2^{63}-1) *	unsigned long	ERROR	ERROR	ERROR	long	ERROR
-2^{63}..(2^{63}-1) *	long	ERROR	ERROR	ERROR	long	ERROR
0..2^{64} *	unsigned long	ERROR	ERROR	ERROR	unsigned long	ERROR

Note: The rows and columns marked with asterisks (*) in this table are only valid when the LP64 option is in effect.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

The __ENUM_OPT macro is predefined to 1 when the ENUMSIZE option is in effect; otherwise it is undefined.

Examples

If the specified storage size is smaller than that required by the range of `enum` constants, an error is issued by the compiler; for example:

```
#include <limits.h>
#pragma enum(1)
enum e_tag {
    a = 0,
     b = SHRT_MAX /* error */
 } e_var;
 #pragma enum(reset)
```

EPILOG

Category

Object code control

Pragma equivalent

`#pragma epilog`

Purpose

Enables you to provide your own function exit code for all functions that have `extern` scope, or for all extern and static functions.

Syntax

```
►►──EPILOG──(──┬─"text-string"──────────────────────┬─)──────────────────────►◄
               └─┬─EXTERN─┬──(──"text-string"──)─┘
                 └─ALL────┘
```

Defaults

The compiler generates default epilog code for the functions that do not have user-supplied epilog code.

Parameters

text-string

text-string is a C string, which must contain valid HLASM statements.

If the *text-string* consists of white-space characters only or if the *text-string* is not provided, then the compiler ignores the option specification. If the *text-string* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the compiler will insert the *text-string* into the function epilog location of the generated assembler source. The compiler does not understand or validate the contents of the *text-string*. In order to satisfy the assembly step later, the given *text-string* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like `newline` and `quote` are shell (or command line) meta characters, and maybe preprocessed before reaching the compiler. It is advisable to avoid using them. The intended use of this option is to specify an assembler macro as the function epilog.

EXTERN
If the EPILOG option is specified with this suboption or without any suboption, the epilog applies to all functions that have external linkage in the compilation unit.

ALL
If the EPILOG option is specified with this suboption, the epilog also applies to static functions defined in the compilation unit.

Usage

For more information on METAL C default epilog code, see *Enterprise Metal C for z/OS Programming Guide and Reference*.

Notes:
1. When the EPILOG option is specified multiple times with the same suboption **all** or **extern**, only the function entry code of the last suboption specified will be displayed.
2. The EPILOG option with the suboption **all** overwrites the one with **extern** suboption, or the one without any suboption.

IPA effects

See section Building Metal C programs with IPA in *Enterprise Metal C for z/OS Programming Guide and Reference*.

Predefined macros

None.

Related information

See "PROLOG" on page 116 for information on providing function entry code for system development.

EVENTS | NOEVENTS

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Produces an event file that contains error information and source file statistics.

Syntax

```
     ┌─NOEVENT─┐
►►───┴─EVENT───┴─┬──────────────────────────────────────────────────┬──────────►◄
                 └─(──┬─Sequential filename──────────────────┬──)──┘
                      ├─Partitioned data set─────────────────┤
                      ├─Partitioned data set (member)────────┤
                      ├─z/OS UNIX System Services filename───┤
                      └─z/OS UNIX System Services directory──┘
```

Defaults

NOEVENTS

Parameters

Sequential filename
: Specifies the sequential data set file name for the event file.

Partitioned data set
: Specifies the partitioned data set for the event file.

Partitioned data set (member)
: Specifies the partitioned data set (member) for the event file.

z/OS UNIX System Services filename
: Specifies the z/OS UNIX file name for the event file.

z/OS UNIX System Services directory
: Specifies the z/OS UNIX System Services directory for the event file.

Usage

The compiler writes the events data to the DD:SYSEVENT ddname, if you allocated one before you called the compiler. If this ddname is not allocated, the compiler will allocate one dynamically using default characteristics (LRECL=4095,RECFM=V,BLKSIZE=4099), and the name is the source file name with SYSEVENT as the lowest-level qualifier. You can control the name by specifying the file name as the suboption of the EVENTS option.

If you specified a suboption, the compiler uses the *data set* that you specified, and ignores the DD:SYSEVENT.

There is no set requirement on the file characteristics for the event file. If you want to allocate an event file, you should specify a record length that is large enough to contain the longest message that the compiler can emit plus approximately 40 bytes for the control information.

If the source file is a z/OS UNIX file, and you do not specify the event file name as a suboption, the compiler writes the event file in the current working directory. The event file name is the name of the source file with the extension `.err`.

The compiler ignores `#line` directives when the EVENTS option is active, and issues a warning message.

Predefined macros

None.

EXPMAC | NOEXPMAC

Category

Listings, messages, and compiler information

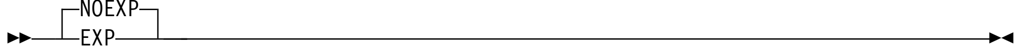
Pragma equivalent

None.

Purpose

Lists all expanded macros in the source listing.

Syntax



Defaults

NOEXPMAC

Usage

If you want to use the EXPMAC option, you must also specify the SOURCE compiler option to generate a source listing. If you specify the EXPMAC option but omit the SOURCE option, the compiler issues a warning message, and does not produce a source listing.

Predefined macros

None.

Related information

"SOURCE | NOSOURCE" on page 134

FLAG | NOFLAG

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Limits the diagnostic messages to those of a specified level or higher.

Syntax

```
   ┌─FL──(──severity──)─┐
►►─┴─NOFL───────────────┴──────────────────────────────────────────►◄
```

Defaults

FLAG(I)

Parameters

severity
  Specifies the minimum severity level.

severity may be one of the following:

I An informational message.

W A warning message that calls attention to a possible error, although the statement to which it refers is syntactically valid.

E An error message that shows that the compiler has detected an error and cannot produce an object deck.

S A severe error message that describes an error that forces the compilation to terminate.

U An unrecoverable error message that describes an error that forces the compilation to terminate.

Usage

When the FLAG option is in effect, you can specify the minimum severity level of diagnostic messages to be reported in a listing and displayed on a terminal.

If you specified the option SOURCE or LIST, the messages generated by the compiler appear immediately following the incorrect source line, and in the message summary at the end of the compiler listing.

The NOFLAG option is the same as the FLAG(U) option.

IPA effects

The FLAG option has the same effect on the IPA link step that it does on a regular compilation.

Predefined macros

None.

Related information

"SOURCE | NOSOURCE" on page 134

"LIST | NOLIST" on page 83

FLOAT

Category

Floating-point and integer control

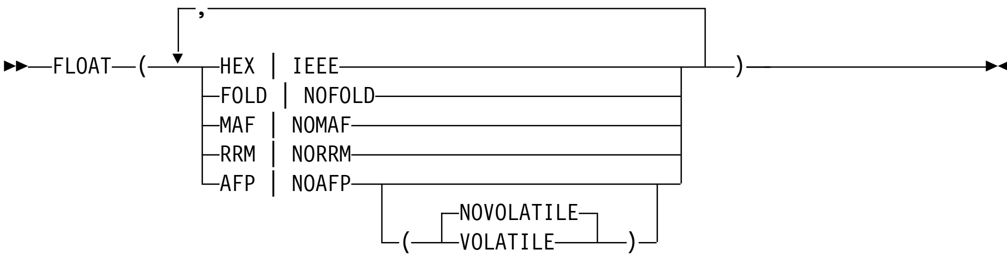
Pragma equivalent

None.

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax



Defaults

- FLOAT(IEEE, FOLD, NOMAF, NORRM, NOAFP)
- For ARCH(3) or higher, the default suboption is AFP(NOVOLATILE).

Parameters

HEX | IEEE

Specifies the format of floating-point numbers and instructions:

- IEEE instructs the compiler to generate binary floating-point numbers and instructions. The unabbreviated form of this suboption is IEEE754.
- HEX instructs the compiler to generate hexadecimal floating-point formatted numbers and instructions. The unabbreviated form of this suboption is HEXADECIMAL.

FOLD | NOFOLD

Specifies that constant floating-point expressions in function scope are to be evaluated at compile time rather than at run time. This is known as *folding*.

In binary floating-point mode, the folding logic uses the rounding mode set by the ROUND option.

In hexadecimal floating-point mode, the rounding is always towards zero. If you specify NOFOLD in hexadecimal mode, the compiler issues a warning and uses FOLD.

MAF | NOMAF

Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results may be produced. This option may affect the precision of floating-point intermediate results.

Note: The suboption MAF does not have any effect on extended floating-point operations.

For ARCH(8) or lower, MAF is not available for hexadecimal floating-point mode.

RRM | NORRM

Runtime Rounding Mode (RRM) prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. It informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not round to nearest at run time.

RRM is not available for hexadecimal floating-point mode.

AFP(VOLATILE | NOVOLATILE) | NOAFP

AFP instructs the compiler to generate code which uses the full complement of 16 floating point registers. These include the four original floating-point registers, numbered 0, 2, 4, and 6, and the Additional Floating Point (AFP) registers, numbered 1, 3, 5, 7 and 8 through 15.

AFP is not available before ARCH(3). If the code generated using AFP registers must run on a pre-ARCH(3) machine, emulation is provided by the operating system. Code with AFP registers will not run on a system that is older than G5 and OS/390® V2R6.

Note: This emulation has a significant performance cost to the execution of the application on the non-AFP processors. This is why the default is NOAFP when ARCH(2) or lower is specified.

If VOLATILE is specified then AFP FPRs 8-15 are considered volatile, which means that FPRs 8-15 are not expected to be preserved by the called program.

Note: The AFPs are FPR1, 3, 5, 7 and 8-15. However, FPRs 0-7 are always considered volatile. The AFP(VOLATILE | NOVOLATILE) option only controls how the compiler handles AFP FPRs 8-15, and not all the AFP registers.

NOAFP limits the compiler to generating code using only the original four floating-point registers, 0, 2, 4, and 6, which are available on all IBM Z® machine models.

Usage

When the FLOAT option is in effect, you can select the format of floating-point numbers. The format can be either base 2 IEEE-754 binary format, or base 16 z/Architecture hexadecimal format.

You should use IEEE floating-point in the following situations:

- You deal with data that are already in IEEE floating-point format.
- You need the increased exponent range.
- You want the changes in programming paradigm provided by infinities and NaN (not a number).

For more information about the IEEE format, refer to the *IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*.

When you use IEEE floating-point, make sure that you are in the same rounding mode at compile time (specified by the ROUND(*mode*) option), as at run time. Entire compilation units will be compiled with the same rounding mode throughout the compilation. If you switch runtime rounding modes inside a function, your results may vary depending upon the optimization level used and other characteristics of your code; switch rounding mode inside functions with caution.

If you have existing data in hexadecimal floating-point (the original base 16 z/Architecture hexadecimal floating-point format), and have no need to communicate this data to platforms that do not support this format, there is no reason for you to change to IEEE floating-point format.

Applications that mix the two formats are not supported.

The binary floating-point instruction set is physically available only on processors that are part of the ARCH(3) group or higher. You can request FLOAT(IEEE) code generation for an application that will run on an ARCH(2) or earlier processor, if that processor runs on the OS/390 Version 2 Release 6 or higher operating system. This operating system level is able to intercept the use of an "illegal" binary floating-point instruction, and emulate the execution of that instruction such that the application logic is unaware of the emulation. This emulation comes at a significant cost to application performance, and should only be used under special circumstances. For example, to run exactly the same executable object module on backup processors within your organization, or because you make incidental use of binary floating-point numbers.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

Note: The option FLOAT(AFP(VOLATILE)) is not supported by IPA. If the option FLOAT(AFP(VOLATILE)) is passed to the IPA Compile or Link phase, then the IPA phase will emit a severe diagnostic message.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can

be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same floating-point mode, and the same values for the FLOAT suboptions, and the ROUND and STRICT options:

- Floating-point mode (binary or hexadecimal)

  The floating-point mode for a partition is set to the floating-point mode (binary or hexadecimal) of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same floating-point mode; a binary floating-point mode subprogram is placed in a binary floating-point mode partition, and a hexadecimal mode subprogram is placed in a hexadecimal mode partition.

  If you specify FLOAT(HEX) or FLOAT(IEEE) during the IPA link step, the option is accepted, but ignored. This is because it is not possible to change the floating-point mode after source analysis has been performed.

  The Prolog and Partition Map sections of the IPA link step listing display the setting of the floating-point mode.

- AFP | NOAFP

  The value of AFP for a partition is set to the AFP value of the first subprogram that is placed in the partition. Subprograms that have the same AFP value are then placed in that partition.

  You can override the setting of AFP by specifying the suboption on the IPA link step. If you do so, all partitions will contain that value, and the Prolog section of the IPA link step listing will display the value.

  The Partition Map section of the IPA link step listing and the END information in the IPA object file display the current value of the AFP suboption.

- FOLD | NOFOLD

  Hexadecimal floating-point mode partitions are always set to FOLD.

  For binary floating-point partitions, the value of FOLD for a partition is set to the FOLD value of the first subprogram that is placed in the partition. Subprograms that have the same FOLD value are then placed in that partition. During IPA inlining, subprograms with different FOLD settings may be combined in the same partition. When this occurs, the resulting partition is always set to NOFOLD.

  You can override the setting of FOLD | NOFOLD by specifying the suboption on the IPA link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA link step listing displays the current value of the FOLD suboption.

- MAF | NOMAF

  For IPA object files generated with the FLOAT(IEEE) option, the value of MAF for a partition is set to the MAF value of the first subprogram that is placed in the partition. Subprograms that have the same MAF for this suboption are then placed in that partition.

  For IPA object files generated with the FLOAT(IEEE) option, you can override the setting of MAF | NOMAF by specifying the suboption on the IPA link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA link step listing displays the current value of the MAF suboption.

  Hexadecimal mode partitions are always set to NOMAF for ARCH(8) or lower. You cannot override this setting.

- RRM | NORRM

  For IPA object files generated with the FLOAT(IEEE) option, the value of RRM for a partition is set to the RRM value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different RRM settings may be combined in the same partition. When this occurs, the resulting partition is always set to RRM.

  For IPA object files generated with the FLOAT(IEEE) option, you can override the setting of RRM | NORRM by specifying the suboption on the IPA link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA link step listing displays the current value of the RRM suboption.

  Hexadecimal mode partitions are always set to NORRM. You cannot override this setting.
- ROUND option

  For IPA object files generated with the FLOAT(IEEE) option, the value of the ROUND option for a partition is set to the value of the first subprogram that is placed in the partition.

  You can override the setting of ROUND by specifying the option on the IPA link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA link step listing displays the current value of the ROUND suboption.

  Hexadecimal mode partitions are always set to round towards zero. You cannot override this setting.
- STRICT option

  The value of the STRICT option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different STRICT settings may be combined in the same partition. When this occurs, the resulting partition is always set to STRICT.

  If there are no compilation units with subprogram-specific STRICT options, all partitions will have the same STRICT value.

  If there are any compilation units with subprogram-specific STRICT options, separate partitions will continue to be generated for those subprograms with a STRICT option, which differs from the IPA Link option.

  The Partition Map sections of the IPA link step listing and the object module display the value of the STRICT option.

Note: The inlining of subprograms (C functions) is inhibited if the FLOAT suboptions (including the floating-point mode), and the ROUND and STRICT options are not all compatible between compilation units. Calls between incompatible compilation units result in reduced performance. For best performance, compile your applications with consistent options.

Predefined macros

__BFP__ is defined to 1 when you specify binary floating point (BFP) mode by using the FLOAT(IEEE) compiler option.

GOFF | NOGOFF

Category

Object code control

Pragma equivalent

None.

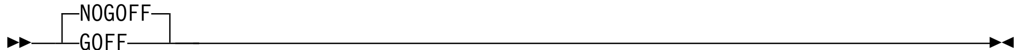
Purpose

Instructs the compiler to produce assembly in a form suitable for generation of a Generalized Object File Format (GOFF) object file.

When the GOFF option is in effect, the compiler produces an assembly file suitable for GOFF assembly.

When the NOGOFF option is in effect, the compiler produces the default assembly file.

Syntax



Defaults

- NOGOFF

Usage

The GOFF format supersedes the IBM S/370 Object Module and Extended Object Module formats. It removes various limitations of the previous format (for example, 16 MB section size) and provides a number of useful extensions, including native z/OS support for long names and attributes. GOFF incorporates some aspects of industry standards such as XCOFF and ELF.

When you specify the GOFF option, the compiler uses LONGNAME and CSECT() by default. You can override these default values by explicitly specifying the NOLONGNAME or the NOCSECT option.

When you specify the GOFF option, you must use the binder to bind the output object.

Note: When using GOFF and source files with duplicate file names, the linker may emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The GOFF option affects the object format of the code and data generated for each partition.

Information from non-IPA input files is processed and transformed based on the original format. GOFF format information remains in GOFF format.

Predefined macros

__GOFF__ is predefined to 1 when the GOFF compiler option is in effect.

Related information

For more information on related compiler options, see:
- "LONGNAME | NOLONGNAME" on page 88
- "CSECT | NOCSECT" on page 43

HALT

Category

Error checking and debugging

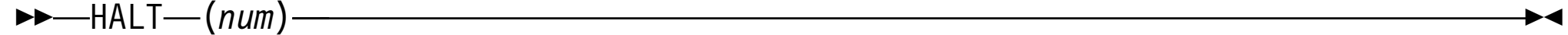
Pragma equivalent

None.

Purpose

Stops compilation process of a set of source code before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity specified for this option.

Syntax



Defaults

HALT(16)

Parameters

num
: Return code from the compiler. See *Enterprise Metal C for z/OS Messages* for a list of return codes.

Usage

If the return code from compiling a particular member is greater than or equal to the value *num* specified in the HALT option, no more members are compiled. This option only applies to the compilation of all members of a specified PDS or z/OS UNIX System Services file system directory.

IPA effects

The HALT option affects the IPA link step in a way similar to the way it affects the IPA compile step, but the message severity levels may be different.

Predefined macros

None.

HALTONMSG | NOHALTONMSG

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.

Syntax

Defaults

NOHALTON

Parameters

msg_number
Message number.

Note: The HALTONMSG option allows you to specify more than one message number by separating the message numbers with colons.

Usage

When the HALTONMSG compiler option is in effect, the compiler stops after the compilation phase when it encounters the specified message number.

When the compilation stops as a result of the HALTONMSG option, the compiler return code is nonzero.

Predefined macros

None.

HGPR | NOHGPR

Category

Optimization and tuning

Pragma equivalent

None.

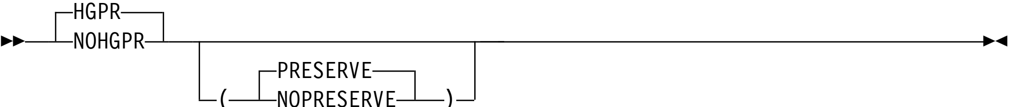
Purpose

Enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.

When the HGPR compiler option is in effect, the compiler can exploit 64-bit GPRs in the generated code. The compiler will take advantage of this permission when the code generation condition is appropriate.

When the NOHGPR compiler option is in effect, the compiler cannot exploit 64-bit GPRs in the generated code.

Syntax



Defaults

HGPR(PRESERVE)

Parameters

PRESERVE
Instructs the compiler to preserve the high halves of the 64-bit GPRs that a function is using, by saving them in the prolog for the function and restoring them in the epilog. The PRESERVE suboption is only necessary if the caller is not known to be Enterprise Metal C for z/OS compiler-generated code.

NOPRESERVE
Instructs the compiler not to preserve the high halves of the 64-bit GPRs that a function is using.

Usage

HGPR means "High-half of 64-bit GPR", which refers to the use of native 64-bit instructions. In particular, if the application has the use of `long long` types, it should benefit from the native 64-bit instructions.

The HGPR compiler option requires ARCH(5) or a higher level.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step will accept the HGPR option, but ignores it. The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The HGPR setting for a partition is determined by the first function that is imported into the partition. All other functions that are imported into the given partition must have the same HGPR option setting.

Predefined macros

None.

HOT | NOHOT

Category

Optimization and tuning

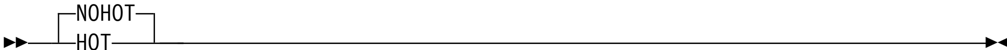
Pragma equivalent

None.

Purpose

Performs high-order loop analysis and transformations (HOT) during optimization.

Syntax



Defaults

NOHOT

Usage

If HOT is specified, the optimization level is forced to a level of at least 2. If a lower level is requested, it is increased to 2. If a higher level is requested, the requested value is used.

When the HOT option is used with DEBUG, the DEBUG(NOSYMBOL) suboption is forced.

If **-g** is specified, **-Wc,NOHOT** is forced.

The HOT option is independent of the UNROLL command line option or pragma. The HOT option setting will be listed in the compiler listing, the IPA Link phase listing, and the end card of the output object file.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

INCLUDE | NOINCLUDE

Category

Compiler input

Pragma equivalent

None.

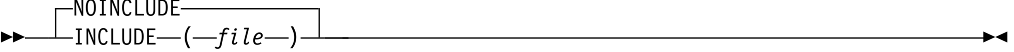
Purpose

Specifies additional header files to be included in a compilation unit, as though the files were named in consecutive `#include` "file" statements inserted before the first line of the source file.

The header files specified with the INCLUDE option are inserted before the first line of the source file.

This option simplifies the task of porting code across supported platforms by providing a way to affect the processing of the source code without having to change it.

Syntax



Defaults

NOINCLUDE, which ignores any INCLUDE options that were in effect prior to NOINCLUDE. The NOINCLUDE option does not have suboptions.

Parameters

file
: The name of a header file to be included at the beginning of the source file being compiled.

Usage

This option is applied only to the files specified in the same compilation as if it was specified for each individual file. It is not passed to any compilations that occur during the link step.

If you specify the option multiple times, the header files are included in order of appearance. If the same header file is specified multiple times with this option, the header is treated as if it was included multiple times by `#include` directives in the source file.

The file specified with the INCLUDE option is searched for first in the current directory when compiling in z/OS UNIX. If the file is not found in the current

directory or when compiling in batch, the file is searched for as if it was included by an `#include` directive in the source file.

The files specified with the INCLUDE option will be included as a dependency of the source file if the **-M** or **MAKEDEF** option is used to generate information to be included in a "make" description file.

When a dependency file is created as a result of a first build with the INCLUDE option, a subsequent build without the INCLUDE option will trigger recompile if the header file on the INCLUDE option was touched between the two builds.

The files specified with the INCLUDE option will show up in the "INCLUDES" section of the compiler listing file that is generated by the SOURCE or LIST option, similar to how they would as if they were included by `#include "file_name"` directives.

IPA effects

None.

Predefined macros

None.

Examples

The following file `t.h` is a predefined header file:

```
#define STRING "hello world"
```

The following source file `t.c` is to be compiled:

```
int main () {
    return strlen (STRING);
}
```

To compile `t.c` by specifying `t.h` with the INCLUDE option, enter:

```
metalc t.c -qinclude=string.h -qinclude=t.h
as t.s
ld t.o
./a.out
echo $?
```

The following output is produced:

```
11
```

INFO | NOINFO

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Produces groups of informational messages.

The compiler does not emit messages for files in the standard search paths for the compiler and system header files.

Syntax

Defaults

NOINFO

Parameters

subopts
Use *subopts* if you want to specify the type of warning messages.

A list of the applicable *subopts* is as follows:

ALS | NOALS
Emits report on possible violations of the ANSI aliasing rule in effect.

The traceback diagnostic messages refer to the character number as the column number.

CMP | NOCMP
Emits conditional expression check messages.

CND | NOCND
Emits messages on redundancies or problems in conditional expressions.

CNV | NOCNV
Emits messages about conversions.

CNS | NOCNS
Emits redundant operation on constants messages.

EFF | NOEFF
Emits information about statements with no effect.

ENU | NOENU
Emits information about ENUM checks.

EXT | NOEXT
Emits warnings about unused variables that have external declarations.

GEN | NOGEN
Emits messages if the compiler generates temporaries, and diagnoses variables that are used without being initialized.

LAN | NOLAN
Emits language level checks.

PAR | NOPAR
Emits warning messages on unused parameters.

POR | NOPOR
Emits warnings about non-portable constructs.

PPC | NOPPC
Emits messages on possible problems with using the preprocessor.

PPT | NOPPT
Emits trace of preprocessor actions.

PRO | NOPRO
Emits warnings about missing function prototypes.

REA | NOREA
Emits warnings about unreached statements.

RET | NORET
Emits warnings about return statement consistency.

TRD | NOTRD
Emits warnings about possible truncation of data.

USE | NOUSE
Emits information about usage of variables.

ALL Enables all of the suboptions except ALS and PPT. Suboptions ALS and PPT have to be turned on explicitly.

Usage

If you specify INFO with no suboptions, it is the same as specifying INFO(ALL).

IPA effects

The IPA link step merges and optimizes the application code; then the IPA link step divides it into sections for code generation. Each of these sections is a partition.

If you do not specify the option on the IPA link step, the value used for a partition depends on the value that you specified for the IPA compile step for each compilation unit that provided code for that partition.

The object module and the Partition Map section of the IPA link step listing display the final option value for each partition. If you override this option on the IPA link step, the Prolog section of the IPA link step listing displays the value of the option.

The Compiler Options Map section of the IPA link step listing displays the option value that you specified for each IPA object file during the IPA compile step.

Predefined macros

None.

INITAUTO | NOINITAUTO

Category

Error checking and debugging

Pragma equivalent

None.

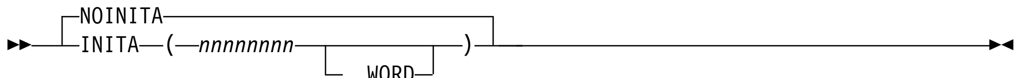
Purpose

Initializes automatic variables to a specific value for debugging purposes.

When the INITAUTO compiler option is in effect, the option instructs the compiler to generate extra code to initialize these variables with a user-defined value. This reduces the runtime performance of the program and should only be used for debugging.

When the NOINITAUTO compiler option is in effect, automatic variables without initializers are not implicitly initialized.

Syntax



Defaults

NOINITAUTO

Parameters

nnnnnnnn
The hexadecimal value you specify for *nnnnnnnn* represents the initial value for automatic storage in bytes. It can be two to eight hexadecimal digits in length. There is no default for this value.

WORD
The suboption WORD is optional, and can be abbreviated to W. If you specify WORD , *nnnnnnnnn* is a word initializer; otherwise it is a byte initializer. Only one initializer can be in effect for the compilation. If you specify INITAUTO more than once, the compiler uses the last setting.

Note: The word initializer is useful in checking uninitialized pointers.

Usage

Automatic variables require storage only while the block in which they are declared is active.

If you specify a byte initializer, and specify more than 2 digits for *nnnnnnnn*, the compiler uses the last 2 digits.

If you specify a word initializer, the compiler uses the last 2 digits to initialize a byte, and all digits to initialize a word.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

If you do not specify the INITAUTO option in the IPA link step, the setting in the IPA compile step will be used. The IPA link step merges and optimizes the application's code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same INITAUTO setting.

The IPA link step sets the INITAUTO setting for a partition to the specification of the first subprogram that is placed in the partition. It places subprograms that follow in partitions that have the same INITAUTO setting.

You can override the setting of INITAUTO by specifying the option on the IPA link step. If you do so, all partitions will use that value, and the Prolog section of the IPA link step listing will display the value.

The Partition Map sections of the IPA link step listing and the object module display the value of the INITAUTO option.

Predefined macros

- __INITAUTO__ is defined to the hexadecimal constant (0xnnU), including the parentheses, when the INITAUTO compiler option is in effect. Otherwise, it is not defined.
- __INITAUTO_W__ is defined to the hexadecimal constant (0xnnnnnnnnU), including the parentheses, when the INITAUTO compiler option is in effect. Otherwise, it is not defined.

INLINE | NOINLINE

Category

Optimization and tuning

Pragma equivalent

#pragma inline, **#pragma noinline**

#pragma options (inline), **#pragma options (noinline)**

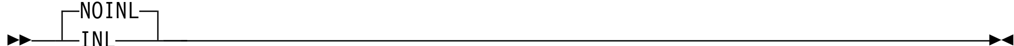
Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

When the INLINE compiler option is in effect, the compiler places the code for selected subprograms at the point of call; this is called *inlining.* It eliminates the linkage overhead and exposes the entire inlined subprogram for optimization by the global optimizer.

When the NOINLINE compiler option is in effect, the compiler generates calls to functions instead of inlining functions.

Syntax



Defaults

NOINLINE

If OPT is in effect, the default is INLINE.

Usage

The INLINE compiler option has the following effects:

- The compiler invokes the compilation unit inliner to perform inlining of functions within the current compilation unit.
- If the compiler inlines all invocations of a static subprogram, it removes the non-inlined instance of the subprogram.
- If the compiler inlines all invocations of an externally visible subprogram, it does not remove the non-inlined instance of the subprogram. This allows callers who are outside of the current compilation unit to invoke the non-inlined instance.

You can specify the INLINE | NOINLINE option on the invocation line and in the **#pragma options** preprocessor directive. When you use both methods at the same time, the compiler merges the options according to the following rules:

- If the NOINLINE option is specified on the invocation line and the **#pragma options(inline)** directive is used, the compiler will behave as if the INLINE option is specified.
- If the INLINE option is specified on the invocation line and the **#pragma options(noinline)** directive is used, the compiler will behave as if the INLINE option is specified.

For example, because you typically do not want to inline your subprograms when you are developing a program, you can use the **#pragma options(noinline)** directive. When you want to inline your subprograms, you can override the **#pragma options(noinline)** by specifying INLINE on the invocation line rather than by editing your source program. The following example illustrates these rules.

Source file:
: **#pragma options(noinline)**

Invocation line:
: INLINE

Result:
: INLINE

Notes:

1. When you specify the INLINE compiler option, a commentis generated in your object module to aid you in diagnosing your program.
2. Specify the LIST or SOURCE compiler option to redirect the output from the INLINE option.

3. If you specify NOINLINE, no subprograms will be inlined even if you have **#pragma inline** directives in your code.
4. If you specify INLINE, subprograms may not be inlined or inline other subprograms when COMPACT is specified (either directly or via **#pragma option_override**). Generate and check the inline report to determine the final status of inlining. The inlining may not occur when OPT(0) is specified via the **#pragma option_override**.

IPA effects

If you specify the INLINE option on the IPA link step, it has the following effects:
- The IPA link step invokes the IPA inliner, which inlines functions in the entire program.
- The IPA link step uses **#pragma inline | noinline** directive information and `inline` subprogram specifier information from the IPA compile step for source program inlining control. Specifying the INLINE option on the IPA compile step has no effect on IPA link step inlining processing.

  You can use the IPA Link control file `inline` and `noinline` directives to explicitly control the inlining of subprograms on the IPA link step. These directives override IPA compile step **#pragma inline | noinline** directives and `inline` subprogram specifiers.
- If the IPA link step inlines all invocations of a subprogram, it removes the non-inlined instance of the subprogram, unless the subprogram entry point was exported using a **#pragma export** directive, or was retained using the IPA Link control file `retain` directive. IPA Link processes static subprograms and externally visible subprograms in the same manner.

The IPA inliner has the inlining capabilities of the compilation unit inliner. In addition, the IPA inliner detects complex recursion, and may inline it.

Predefined macros

None.

IPA | NOIPA

Category

Optimization and tuning

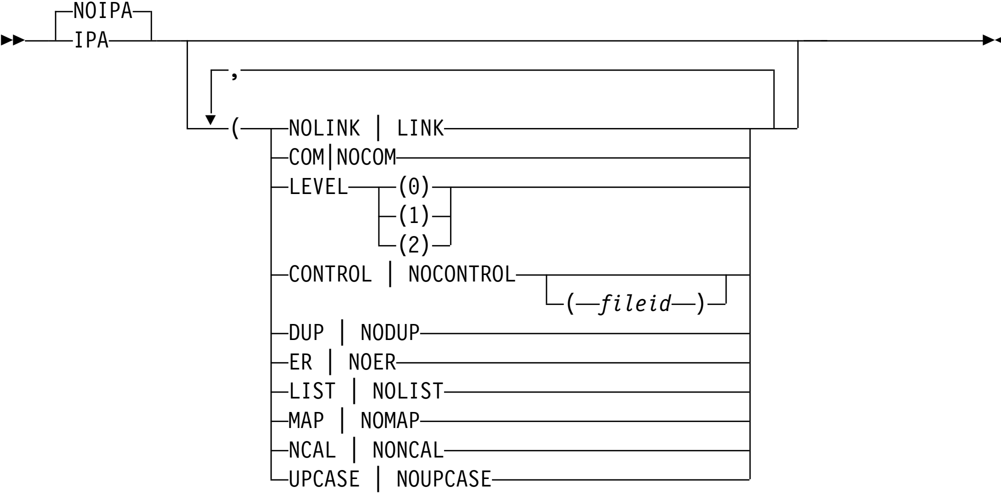
Pragma equivalent

None.

Purpose

Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

Syntax



Defaults

NOIPA

Parameters

IPA compile and link step suboptions

LEVEL(0|1|2)
Indicates the level of IPA optimization that the IPA link step should perform after it links the object files into the call graph.

If you specify LEVEL(0), IPA performs subprogram pruning and program partitioning only.

If you specify LEVEL(1), IPA performs all of the optimizations that it does at LEVEL(0), as well as subprogram inlining and global variable coalescing. IPA performs more precise alias analysis for pointer dereferences and subprogram calls.

Under IPA Level 1, many optimizations such as constant propagation and pointer analysis are performed at the intraprocedural (subprogram) level. If you specify LEVEL(2), IPA performs specific optimizations across the entire program, which can lead to significant improvement in the generated code.

The compiler option OPTIMIZE that you specify on the IPA link step controls subsequent optimization for each partition during code generation. Regardless of the optimization level you specified during the IPA compile step, you can modify the level of IPA optimization, regular code generation optimization, or both, on the IPA link step.

The default is IPA(LEVEL(1)).

IPA compile step suboptions

NOLINK
IPA(NOLINK) invokes the IPA compile step. (NOLINK is the default.)

COMPRESS | NOCOMPRESS
Indicates that the IPA object information is compressed to significantly reduce the size of the IPA object file.

The default is IPA(COMPRESS). The abbreviations are IPA(COM|NOCOM).

LIST | NOLIST
Indicates whether the compiler saves information about source line numbers in the IPA object file.

Refer to "LIST | NOLIST" on page 83 for information about the effect of this suboption on the IPA link step. Refer also to "IPA considerations" on page 7.

The default is IPA(NOLIST). The abbreviations are IPA(LIS|NOLIS). If you specify the LIST option, it overrides the IPA(NOLIST) option.

IPA link step suboptions

LINK
IPA(LINK) invokes the IPA link step.

Only the following IPA suboptions affect the IPA link step. If you specify other IPA suboptions, they are ignored.

CONTROL[(fileid)] | NOCONTROL[(fileid)]
Specifies whether a file that contains IPA directives is available for processing. You can specify an optional `fileid`. If you specify both IPA(NOCONTROL(*fileid*)) and IPA(CONTROL), in that order, the IPA link step resolves the option to IPA(CONTROL(*fileid*)).

The default `fileid` is `DD:IPACNTL` if you specify the IPA(CONTROL) option. The default is IPA(NOCONTROL).

For more information about the IPA link step control file, see "IPA link step control file" on page 186.

DUP | NODUP
Indicates whether the IPA link step writes a message and a list of duplicate symbols to the console.

The default is IPA(DUP).

ER | NOER
Indicates whether the IPA link step writes a message and a list of unresolved symbols to the console.

The default is IPA(NOER).

MAP | NOMAP
Specifies that the IPA link step will produce a listing. The listing contains a Prolog and the following sections:

- Object File Map
- Compiler Options Map
- Global Symbols Map (which may or may not appear, depending on how much global coalescence was done during optimization)
- Partition Map for each partition
- Source File Map

The default is IPA(NOMAP).

See "Using the IPA link step listing" on page 156 for more information.

NCAL | NONCAL
Indicates whether the IPA link step performs an automatic library search to resolve references in files that the IPA compile step produces. Also indicates whether the IPA link step performs library searches to locate an object file or files that satisfy unresolved symbol references within the current set of object information.

This suboption controls both explicit searches triggered by the LIBRARY IPA Link control statement, and the implicit SYSLIB search that occurs at the end of IPA Link input processing.

To help you remember the difference between NCAL and NONCAL, you may think of NCAL as "nocall" and NONCAL as "no nocall", (or "call").

The default is IPA(NONCAL).

UPCASE | NOUPCASE
Determines whether the IPA link step makes an additional automatic library call pass for SYSLIB if unresolved references remain at the end of standard IPA Link processing. Symbol matching is not case sensitive in this pass.

This suboption provides support for linking assembly language object routines, without forcing you to make source changes. The preferred approach is to add **#pragma map** definitions for these symbols, so that the correct symbols are found during normal IPA Link automatic library call processing.

The default is IPA(NOUPCASE). The abbreviations are IPA(UPC|NOUPC).

Predefined macros

None.

KEYWORD | NOKEYWORD

Category

Language element control

Pragma equivalent

None.

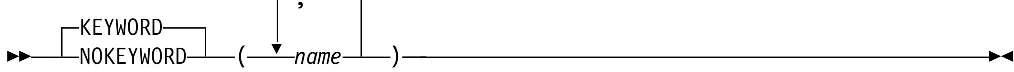
Purpose

Controls whether the specified *name* is treated as a keyword or an identifier whenever it is in your source code.

When the KEYWORD compiler option is in effect, the compiler treats the specified *name* as a keyword.

When the NOKEYWORD compiler option is in effect, the compiler treats the specified *name* as an identifier.

Syntax



Defaults

By default, all of the built-in keywords defined in the C language standard are reserved as keywords.

Parameters

name
The name of a keyword. This suboption is case-sensitive.

Usage

You can use NOKEYWORD(*name*) to disable built-in keywords, and use KEYWORD(*name*) to reinstate those keywords.

This option can be used with the following C keywords:
- asm
- typeof

Note: `asm` is not reserved as a keyword at the **stdc89** or **stdc99** language level.

Predefined macros

The following predefined macros are set using the KEYWORD | NOKEYWORD option:
- __BOOL__ is undefined when the NOKEYWORD(bool) is in effect.
- __IBM__TYPEOF__ is predefined to 1 when the KEYWORD(typeof) is in effect.

LANGLVL

Category

Language element control

Pragma equivalent

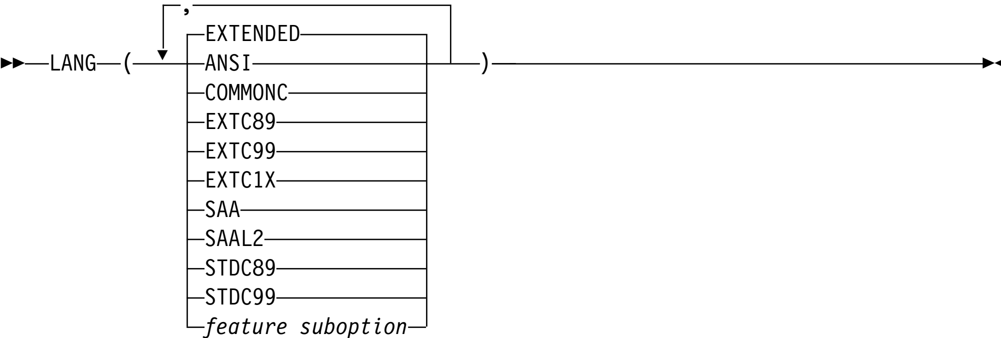
#pragma langlvl

Purpose

Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

Category: Language element control



Defaults

LANGLVL(EXTENDED)

Parameters

The following language levels are supported:

ANSI

Use it if you are compiling new or ported code that is ISO C compliant. It indicates language constructs that are defined by ISO. Some non-ANSI stub routines will exist even if you specify LANGLVL(ANSI), for compatibility with previous releases. The macro __ANSI__ is defined as 1. It is intended to ensure that the compilation conforms to the ISO C standard.

Note: When you specify LANGLVL(ANSI), the compiler can still read and analyze the `_Packed` keyword. If you want to make your code purely ANSI, you should redefine `_Packed` in a header file as follows:

```
#ifdef __ANSI__
  #define _Packed
#endif
```

The compiler will now see the `_Packed` attribute as a blank when LANGLVL(ANSI) is specified at compile time, and the language level of the code will be ANSI.

COMMONC

It indicates language constructs that are defined by XPG, many of which LANGLVL(EXTENDED) already supports. LANGLVL(ANSI) and LANGLVL(EXTENDED) do not support the following, but LANGLVL(COMMONC) does:

- Unsignedness is preserved for standard integral promotions (that is, `unsigned char` is promoted to `unsigned int`)
- Trigraphs within literals are not processed
- `sizeof` operator is permitted on bit fields
- Bit fields other than `int` are tolerated, and a warning message is issued
- Macro parameters within quotation marks are expanded
- The empty comment in a subprogram-like macro is equivalent to the ANSI/ISO token concatenation operator

The macro __COMMONC__ is defined as 1 when you specify LANGLVL(COMMONC).

If you specify LANGLVL(COMMONC), the ANSIALIAS option is automatically turned off. If you want ANSIALIAS turned on, you must explicitly specify it.

Note: The option ANSIALIAS assumes code that supports ANSI. Using LANGLVL(COMMONC) and ANSIALIAS together may have undesirable effects on your code at a high optimization level. See "ANSIALIAS | NOANSIALIAS" on page 24 for more information.

EXTC89

Indicates language constructs that are defined by the ISO C89 standard, plus additional orthogonal language extensions that do not alter the behavior of this standard.

Note: The unicode literals are enabled under the EXTC89 language level, and disabled under the strictly-conforming language levels. When the unicode literals are enabled, the macro __IBM_UTF_LITERAL is predefined to 1. Otherwise, this macro is not predefined.

EXTC99
Indicates language constructs that are defined by the ISO C99 standard, plus additional orthogonal language extensions that do not alter the behavior of the standard.

Note: The unicode literals are enabled under the EXTC99 language level, and disabled under the strictly-conforming language levels. When the unicode literals are enabled, the macro __IBM_UTF_LITERAL is predefined to 1. Otherwise, this macro is not predefined.

EXTC1X
Compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions. For more information about the currently supported C11 features, see Extensions for C11 compatibility in *Enterprise Metal C for z/OS Language Reference*.

Note: IBM supports selected features of C11. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C11 features is complete, including the support of a new C11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the C11 features.

EXTENDED
It indicates all language constructs are available with Enterprise Metal C for z/OS. It enables extensions to the ISO C standard. The macro __EXTENDED__ is defined as 1.

Note: Under Enterprise Metal C for z/OS, the unicode literals are enabled under the EXTENDED language level, and disabled under the strictly-conforming language levels. When the unicode literals are enabled, the macro __IBM_UTF_LITERAL is predefined to 1. Otherwise, this macro is not predefined.

SAA
Indicates language constructs that are defined by SAA.

SAAL2
Indicates language constructs that are defined by SAA Level 2.

STDC89
Indicates language constructs that are defined by the ISO C89 standard. This suboption is synonymous with LANGLVL(ANSI).

STDC99
Indicates language constructs that are defined by the ISO C99 standard.

The following feature suboptions are available:

LIBEXT | NOLIBEXT
Specifying this option affects the headers provided by the C runtime library, which in turn control the availability of general ISO runtime extensions. In addition, it also defines the following macros and sets their values to 1:

- _MI_BUILTIN (this macro controls the availability of machine built-in instructions.
- _EXT (this macro controls the availability of general ISO runtime extensions)

The default is LANG(LIBEXT). However, LANG(LIBEXT) is implicitly enabled by LANG(COMMONC | SAA | SAAL2 | EXTENDED | EXTC89 | EXTC99).

LONGLONG | NOLONGLONG
This option controls the availability of pre-C99 `long long` integer types for your compilation. The default is LANG(LONGLONG).

TEXTAFTERENDIF | NOTEXTAFTERENDIF
Specifies whether to suppress the warning message that is emitted when you are porting code from a compiler that allows extra text after `#endif` or `#else` to the Enterprise Metal C for z/OS compiler. The default option is LANGLVL(NOTEXTAFTERENDIF), indicating that a message is emitted if `#else` or `#endif` is followed by any extraneous text.

Usage

The LANGLVL option defines a macro that specifies a language level. You must then include this macro in your code to force conditional compilation; for example, with the use of `#ifdef` directives. You can write portable code if you correctly code the different parts of your program according to the language level. You use the macro in preprocessor directives in header files.

Note: The following list shows ISO C99 language constructs unavailable with LANGLVL(EXTENDED) or LANGLVL(EXTC89):
- `inline` keyword
- `restrict` keyword
- C++ style comments

Unsuffixed integer literals are handled differently under ISO C99 than they are for LANGLVL(EXTENDED) or LANGLVL(EXTC89). Unsuffixed integer literals with values greater than INT_MAX, have a `long long` type under ISO C99 and an `unsigned int` type under LANGVL(EXTENDED) or LANGLVL(EXTC89).

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

For the list of predefined macros related to language levels, see Macros related to language levels in *Enterprise Metal C for z/OS Language Reference*.

LIBANSI | NOLIBANSI

Category

Optimization and tuning

Pragma equivalent

`#pragma options (libansi)`, **`#pragma options (nolibansi)`**

Purpose

Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard.

When LIBANSI is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

Defaults

NOLIBANSI

Usage

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The LIBANSI option has the same effect on the IPA compile step as it does for normal compilation.

The LIBANSI option will be in effect for the IPA link step unless the NOLIBANSI option is specified.

The LIBANSI option that you specify on the IPA link step will override the LIBANSI option that you specify on the IPA compile step. The LIBANSI option that you specify on the IPA link step is shown in the IPA Link listing Compile Option Map for reference.

Predefined macros

None.

LIST | NOLIST

Category

Listings, messages and compiler information

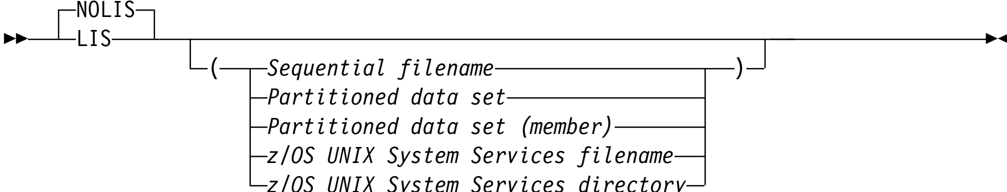
Pragma equivalent

None.

Purpose

Produces a compiler listing that includes a list of options and the compiler version.

Syntax



Defaults

NOLIST

Parameters

Sequential filename
Specifies the sequential data set file name for the compiler listing.

Partitioned data set
Specifies the partitioned data set for the compiler listing.

Partitioned data set (member)
Specifies the partitioned data set (member) for the compiler listing.

z/OS UNIX System Services filename
Specifies the z/OS UNIX System Services file name for the compiler listing.

z/OS UNIX System Services directory
Specifies the z/OS UNIX System Services directory for the compiler listing.

Usage

When the LIST compiler option is in effect, the compiler is instructed to generate a listing of options and the compiler version in the compiler listing.

LIST(*filename*) places the compiler listing in the specified file. If you do not specify a file name for the LIST option, the compiler uses the SYSCPRT ddname if you allocated one. Otherwise, the compiler generates a file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the listing data set. The high-level qualifier is replaced with the userid under which the compiler is running, and `.LIST` is appended as the low-level qualifier.
- If you are compiling a z/OS UNIX file, the compiler stores the listing in a file that has the name of the source file with a `.lst` extension. If you are linking with IPA and generating a z/OS UNIX executable, the name is instead based on the name of the executable.

The NOLIST option optionally takes a file name suboption. This file name then becomes the default. If you subsequently use the LIST option without a file name suboption, the compiler uses the file name that you specified in the earlier NOLIST. For example, the following specifications have the same effect:

```
metalc -Wc,"NOLIST(hello.list)" LIST
```

```
metalc -Wc,"LIST(hello.list)"
```

If you specify data set names in a C program, with the SOURCE or LIST options, all the listing sections are combined into the last data set name specified.

Notes:

1. If you use the following form of the command in a JES3 batch environment where *xxx* is an unallocated data set, you may get undefined results.

   LIST(*xxx*)

2. Statement line numbers exceeding 99999 will wrap back to 00000 for the generated assembly listing for the C source file. This may occur when the compiler LIST option is used.

IPA effects

If you specify the LIST option on the IPA compile step, the compiler saves information about the source file and line numbers in the IPA object file. This information is available during the IPA link step for use by the LIST option.

Predefined macros

None.

LOCALE | NOLOCALE

Category

Object code control

Pragma equivalent

None.

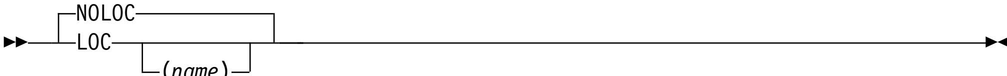
Purpose

Specifies the locale to be used by the compiler as the current locale throughout the compilation unit.

With the LOCALE compiler option, you can specify the locale you want to use.

When the NOLOCALE compiler option is in effect, the compiler uses the default code page, which is IBM-1047.

Syntax



Defaults

NOLOCALE

When compiling in the z/OS UNIX System Services Shell environment, the default is LOCALE(POSIX). The **metalc** utility picks up the locale value of the environment using `setlocale(LC_ALL,NULL))`. Because the compiler runs with the POSIX(OFF) option, categories that are set to C are changed to POSIX.

Parameters

name
: Indicates the name of the locale to be used by the compiler at compile time. If you omit *name*, the compiler uses the current default locale in the environment. If *name* does not represent a valid locale name, a warning message is emitted and NOLOCALE is used.

Usage

You can specify LOCALE on the command line or in the PARMS list in the JCL.

If you specify the LOCALE option, the locale name and the associated code set appear in the header of the listing. A locale name is also generated in the object module.

The LC_TIME category of the current locale controls the format of the time and the date in the compiler-generated listing file. The identifiers that appear in the tables in the listing file are sorted as specified by the LC_COLLATE category of the locale specified in the option.

Note: The formats of the predefined macros __DATE__, __TIME__, and __TIMESTAMP__ are not locale-sensitive.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The LOCALE option controls processing only for the IPA step for which you specify it.

During the IPA compile step, the compiler converts source code using the code page that is associated with the locale specified by the LOCALE compile-time option. As with non-IPA compilations, the conversion applies to identifiers, literals, and listings. The locale that you specify on the IPA compile step is recorded in the IPA object file.

You should use the same code page for IPA compile step processing for all of your program source files. This code page should match the code page of the runtime environment. Otherwise, your application may not run correctly.

The locale that you specify on the IPA compile step does not determine the locale that the IPA link step uses. The LOCALE option that you specify on the IPA link step is used for the following:

- The encoding of the message text and the listing text.
- Date and time formatting in the Source File Map section of the listing and in the text in the object comment string that records the date and time of IPA link step processing.
- Sorting of identifiers in listings. The IPA link step uses the sort order associated with the locale for the lists of symbols in the Inline Report (Summary), Global Symbols Map, and Partition Map listing sections.

If the code page you used for a compilation unit for the IPA compile step does not match the code page you used for the IPA link step, the IPA link step issues an informational message.

If you specify the IPA(MAP) option, the IPA link step displays information about the LOCALE option, as follows:

- The Prolog section of the listing displays the LOCALE or NOLOCALE option. If you specified the LOCALE option, the Prolog displays the locale and code set that are in effect.
- The Compiler Options Map listing section displays the LOCALE option active on the IPA compile step for each IPA object. If you specified conflicting code sets between the IPA Compile and IPA link steps, the listing includes a warning message after each Compiler Options Map entry that displays a conflict.
- The Partition Map listing section shows the current LOCALE option.

Predefined macros

- __CODESET__ is defined to the name of the compile-time code set. The compiler uses the runtime function `nl_langinfo(CODESET)` to determine the name of the compile-time code set. If you do not use the LOCALE compile option, the macro is undefined.
- __LOCALE__ is defined to the name of the compile-time locale. If you specified LOCALE(string literal), the compiler uses the runtime function `setlocale(LC_ALL,"string literal")` to determine the name of the compile-time locale. If you do not use the LOCALE compile option, the macro is undefined.

LONGLONG | NOLONGLONG

Category

Language element control

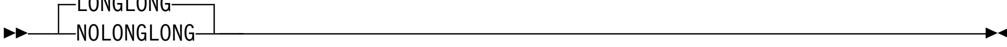
Pragma equivalent

`#pragma options [no]longlong`

Purpose

Controls whether to allow the pre-C99 `long long` integer types in your programs.

Syntax



Defaults

LONGLONG

Usage

This option takes effect when the **LANGLVL(EXTENDED | STDC89 | EXTC89)** option is in effect. It is not valid when the **LANGLVL(STDC99 | EXTC99)** option is in effect, because the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard.

IPA effects

None.

Predefined macros

_LONG_LONG is defined to 1 when `long long` data types are available; otherwise, it is undefined.

LONGNAME | NOLONGNAME

Category

Object code control

Pragma equivalent

#pragma longname, **#pragma nolongname** You can use the #pragma preprocessor directive to override the default values for compiler options. However, for LONGNAME | NOLONGNAME, the compiler options override the #pragma preprocessor directives.

Purpose

Provides support for external names of mixed case and up to 1024 characters long.

When the LONGNAME compiler option is in effect, the compiler generates untruncated and mixed case external names in the assembler source produced by the compiler.

When the NOLONGNAME compiler option is in effect:
- The compiler generates truncated and uppercase names in the assembler source.
- Functions that are given truncated and uppercase names.
- The compiler truncates all the external names to 8 characters.

Syntax

Defaults

NOLONGNAME

Usage

If you use **#pragma map** to associate an external name with an identifier, the compiler generates the external name in the assembler source. That is, **#pragma map** has the same behavior for the LONGNAME and NOLONGNAME compiler options. Also, **#pragma csect** has the same behavior for the LONGNAME and NOLONGNAME compiler options.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

LONGNAME is always in effect even if you specify NOLONGNAME. Either the LONGNAME compiler option or the **#pragma longname** preprocessor directive is required for the IPA compile step.

The IPA link step ignores this option if you specify it, and uses the LONGNAME option for all partitions it generates.

Predefined macros

__LONGNAME__ is predefined to 1 when the LONGNAME compiler option is in effect.

LP64 | ILP32

Category

Object code control

Pragma equivalent

None.

Purpose

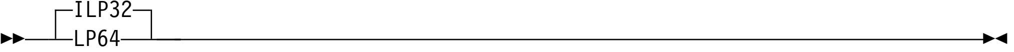
Selects either AMODE 64 or AMODE 31 mode.

When the LP64 compiler option is in effect, the compiler generates AMODE 64 code using the z/Architecture 64-bit instructions.

When the ILP32 compiler option is in effect, the compiler generates AMODE 31 code. This is the default.

Note: AMODE is the addressing mode of the program code generated by the compiler. In AMODE 64 and AMODE 31, 64 and 31 refer to the range of addresses that can be accessed (in other words 64-bits and 31-bits are used to form the address respectively). When there is no ambiguity, we will refer to these as 64-bit mode and 31-bit mode. Refer to the information that follows for further information on the data model.

Syntax



Defaults

ILP32

Usage

LP64 and ILP32 are mutually exclusive. If they are specified multiple times, the compiler will take the last one.

LP64 and ILP32 refer to the data model used by the language. "I" is an abbreviation that represents int type, "L" represents long type, and "P" represents the pointer type. 64 and 32 refer to the size of the data types. When the ILP32 option is used, `int`, `long` and pointers are 32-bit in size. When LP64 is used, `long` and pointer are 64-bit in size; `int` remains 32-bit. The addressing mode used by LP64 is AMODE 64, and by ILP32 is AMODE 31. In the latter case, only 31 bits within the pointer are taken to form the address. For the sake of conciseness, the terms 31-bit mode and ILP32, will be used interchangeably in this document when there is no ambiguity. The same applies to 64-bit mode and LP64.

The LP64 option requires ARCH(5) or higher. ARCH(5) and GOFF are the default settings for LP64 if you don't explicitly override them. If you explicitly specify NOGOFF or specify an architecture level lower than 5, the compiler issues a warning message, ignore NOGOFF, and raise the architecture level to 5.

Note: ARCH(5) specifies the 2064 hardware models.

In 31-bit mode, the size of `long` and pointers is 4 bytes and the size of `wchar_t` is 2 bytes. Under LP64, the size of `long` and pointer is 8 bytes and the size of `wchar_t` is 4 bytes. The size of other intrinsic datatypes remain the same between 31-bit mode and LP64. Under LP64, the type definition for `size_t` changes to `long`, and the type definition for `ptrdiff_t` changes to `unsigned long`. The following tables give the size of the intrinsic types:

Table 17. Size of intrinsic types in 64–bit mode

Type	Size (in bits)
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	64
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	64

Table 18. Size of intrinsic types in 31–bit mode

Type	Size (in bits)
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	32
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	32

The `__ptr32` pointer qualifier is intended to make the process of porting applications from ILP32 to LP64 easier. Use this qualifier in structure members to minimize the changes in the overall size of structures. Note that these pointers cannot refer to objects above the 31-bit address line (also known as "the bar"). In general, the program has no control over the address of a variable; the address is assigned by the implementation. It is up to the programmer to make sure that the use of `__ptr32` is appropriate within the context of the program's logic.

Note: The `long` and `wchar_t` data types also change in size.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step accepts the LP64 | ILP32 option, but ignores it.

The IPA link step will check that all objects have a consistent data model, either ILP32 or LP64. It checks both IPA object modules and non-IPA object modules. If the IPA link step finds a mixture of addressing modes among the object files, the compiler issues a diagnostic message and ends the compilation.

Predefined macros

Macros __64BIT__, _LP64, and __LP64__ are defined to 1 when the LP64 compiler option is in effect; otherwise, the macro _ILP32 is predefined to 1.

LSEARCH | NOLSEARCH

Category

Compiler input

Pragma equivalent

None.

Purpose

Specifies the directories or data sets to be searched for user include files.

When the LSEARCH compiler option is in effect, the preprocessor looks for the user include files in the specified directories or data sets.

When the NOLSEARCH compiler option is in effect, the preprocessor only searches those data sets that are specified in the USERLIB DD statement. A NOLSEARCH option cancels all previous LSEARCH specifications, and the compiler uses any LSEARCH options that follow it.

Syntax

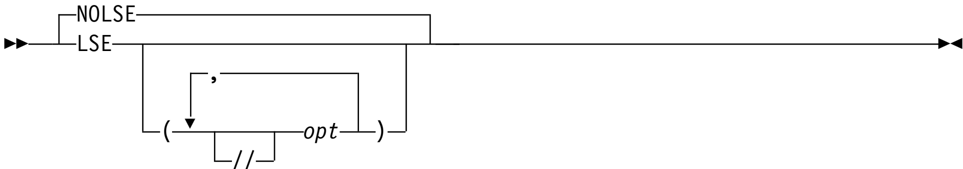
Defaults

NOLSEARCH

Parameters

path
Specifies any of the following:

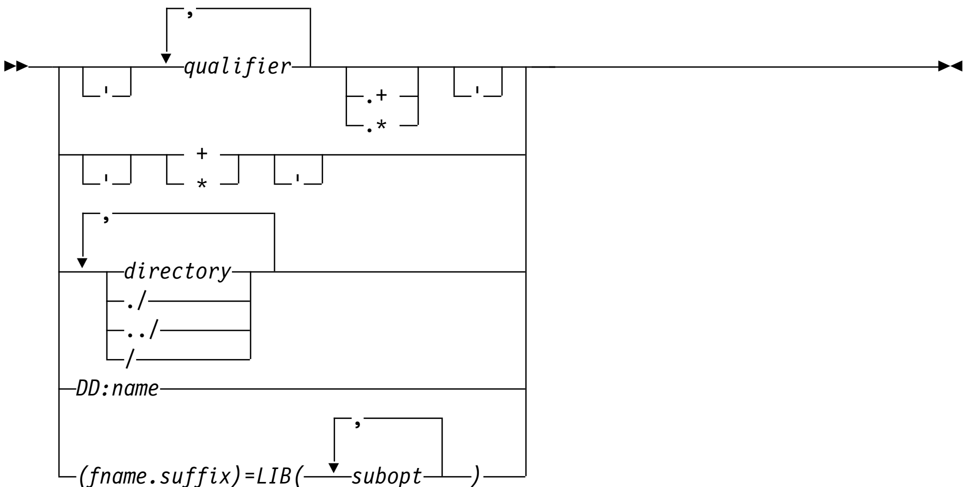
- The name of a partitioned or sequential data set that contains user include files.
- A z/OS UNIX System Services file system path that contains user include files.
- A search path that is more complex:



You must use the double slashes (//) to specify data set library searches when you specify the OE compiler option. (You may use them regardless of the OE option).

The USERLIB ddname is considered the last suboption for LSEARCH, so that specifying LSEARCH `(X)` is equivalent to specifying LSEARCH `(X,DD:USERLIB)`.

Parts of the `#include` *`filename`* are appended to each LSEARCH *opt* to search for the include file. *opt* has the format:



In this syntax diagram, *opt* specifies one of the following:

- The name of a partitioned or sequential data set that contains user include files
- A z/OS UNIX file system path name that should be searched for the include file. You can also use ./ to specify the current directory and ../ to specify the parent directory for your z/OS UNIX file.
- A DD statement for a sequential data set or a partitioned data set. When you specify a ddname in the search and the include file has a member name, the member name of the include file is used as the name for the `DD:` *name* search suboption, for example:

```
LSEARCH(DD:NEWLIB)
#include "a.b(c)"
```

  The resulting file name is `DD:NEWLIB(C)`.
- A specification of the form (*fname.suffix*) = (*subopt*,*subopt*,...) where:
  - *fname* is the name of the include file, or *
  - *suffix* is the suffix of the include file, or *
  - *subopt* indicates a subpath to be used in the search for the include files that match the pattern of *fname.suffix*. There should be at least one *subopt*. The possible values are:
    - LIB(*[pds,...]*) where each *pds* is a partitioned data set name. They are searched in the same order as they are specified.

      There is no effect on the search path if no *pds* is specified, but a warning is issued.
    - LIBs are cumulative; for example, LIB(A),LIB(B) is equivalent to LIB(A, B).
    - NOLIB specifies that all LIB(...) previously specified for this pattern should be ignored at this point.

When the `#include` *filename* matches the pattern of *fname.suffix*, the search continues according to the subopts in the order specified. An asterisk (*) in *fname* or *suffix* matches anything. If the compiler does not find the file, it attempts other searches according to the remaining options in LSEARCH.

Usage

When you specify more than one LSEARCH option, the compiler uses all the directories or data sets in these LSEARCH options to find the user include files.

The `#include "`*filename*`"` format of the `#include` preprocessor directive indicates user include files. See "Using include files" on page 173 for a description of the `#include` preprocessor directive.

Note: If the *filename* in the `#include` directive is in absolute form, the compiler does not perform a search. See "Determining whether the file name is in absolute form" on page 178 for more details on absolute `#include` *filename*.

For further information on search sequences, see "Search sequences for include files" on page 182.

When specifying z/OS UNIX library searches, do not put double slashes at the beginning of the LSEARCH *opt* . Use *pathnames* separated by slashes (/) in the LSEARCH *opt* for a z/OS UNIX library. When the LSEARCH *opt* does not start with double slashes, any single slash in the name indicates a z/OS UNIX library. If

you do not have path separators (/), then setting the OE compile option on indicates that this is a z/OS UNIX library; otherwise the library is interpreted as a data set. See "Using SEARCH and LSEARCH" on page 180 for additional information on z/OS UNIX files.

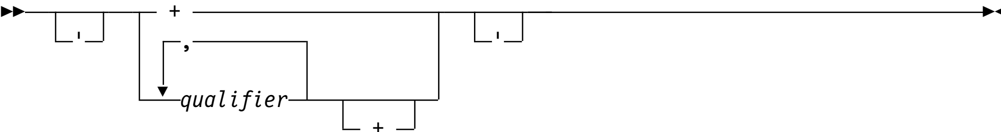
Example: The *opt* specified for LSEARCH is combined with the *filename* in `#include` to form the include file name:

```
LSEARCH(/u/mike/myfiles)
#include "new/headers.h"
```

The resulting z/OS UNIX file name is `/u/mike/myfiles/new/headers.h`.

Use an asterisk (*) or a plus sign (+) in the LSEARCH *opt* to specify whether the library is a sequential or partitioned data set.

When you want to specify a set of PDSs as the search path, you add a period followed by a plus sign (.+) at the end of the last qualifier in the *opt*. If you do not have any qualifier, specify a single plus sign (+) as the *opt*. The *opt* has the following syntax for specifying partitioned data set:



where *qualifier* is a data set qualifier.

Start and end the *opt* with single quotation marks (') to indicate that this is an absolute data set specification. Single quotation marks around a single plus sign (+) indicate that the *filename* that is specified in `#include` is an absolute partitioned data set.

When you do not specify a member name with the `#include` directive, for example, `#include "PR1.MIKE.H"`, the PDS name for the search is formed by replacing the plus sign with the following parts of the *filename* of the `#include` directive:

- For the PDS file name:
  1. All the *path*s and slashes (slashes are replaced by periods)
  2. All the periods and *qualifier*s after the left-most *qualifier*
- For the PDS member name, the left-most *qualifier* is used as the member name

See the first example in Table 19 on page 96.

However, if you specified a member name in the *filename* of the `#include` directive, for example, `#include "PR1.MIKE.H(M1)"`, the PDS name for the search is formed by replacing the plus sign with the qualified name of the PDS. See the second example in Table 19 on page 96.

See "Forming data set names with LSEARCH | SEARCH options" on page 175 for more information on forming PDS names.

Note: To specify a single PDS as the *opt*, do not specify a trailing asterisk (*) or plus sign (+). The library is then treated as a PDS but the PDS name is formed by just using the leftmost *qualifier* of the `#include` *filename* as the member name. For example:

```
LSEARCH(AAAA.BBBB)
#include "sys/ff.gg.hh"

Resulting PDS name is
userid.AAAA.BBBB(FF)
```

Also see the third example in Table 19 on page 96.

Predefined macros

None.

Examples

To search for PDS or PDSE files when you have coded your include files as follows:

```
#include "sub/fred.h"
#include "fred.inl"
```

You specified LSEARCH as follows:

```
LSEARCH(USER.+,'USERID.GENERAL.+')
```

The compiler uses the following search sequence to look for your include files:

1. First, the compiler looks for `sub/fred.h` in this data set:
   `USERID.USER.SUB.H(FRED)`
2. If that PDS member does not exist, the compiler looks in the data set:
   `USERID.GENERAL.SUB.H(FRED)`
3. If that PDS member does not exist, the compiler looks in DD:USERLIB, and then checks the system header files.
4. Next, the compiler looks for `fred.inl` in the data set:
   `USERID.USER.INL(FRED)`
5. If that PDS member does not exist, the compiler will look in the data set:
   `USERID.GENERAL.INL(FRED)`
6. If that PDS member does not exist, the compiler looks in DD:USERLIB, and then checks the system header files.

The compiler forms the search path for z/OS UNIX files by appending the path and name of the `#include` file to the path that you specified in the LSEARCH option.

Example 1: See the following example.

You code `#include "sub/fred.h"` and specify:

```
LSEARCH(/u/mike)
```

The compiler looks for the include file `/u/mike/sub/fred.h` .

Example 2: See the following example.

You specify your header file as `#include "fred.h"`, and your LSEARCH option as:

```
LSEARCH(/u/mike, ./sub)
```

The compiler uses the following search sequence to look for your include files:

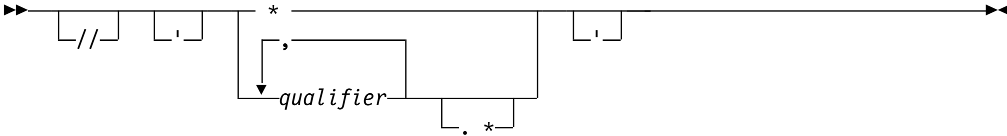
1. The compiler looks for `fred.h` in `/u/mike/fred.h`.
2. If that z/OS UNIX file does not exist, the compiler looks in `./sub/fred.h`.
3. If that z/OS UNIX file does not exist, the compiler looks in the libraries specified on the USERLIB DD statement.
4. If USERLIB DD is not allocated, the compiler follows the search order for system include files.

The following example shows you how to specify a PDS search path:

Table 19. Partitioned data set examples

include Directive	LSEARCH option	Result
#include "PR1.MIKE.H"	LSEARCH('CC.+')	'CC.MIKE.H(PR1)'
#include "PR.KE.H(M1)"	LSEARCH('CC.+')	'CC.PR.KE.H(M1)'
#include "A.B"	LSEARCH(CC)	*userid*.CC(A)
#include "A.B.D"	LSEARCH(CC.+)	*userid*.CC.B.D(A)
#include "a/b/dd.h"	LSEARCH('CC.+')	'CC.A.B.H(DD)'
#include "a/dd.ee.h"	LSEARCH('CC.+')	'CC.A.EE.H(DD)'
#include "a/b/dd.h"	LSEARCH('+')	'A.B.H(DD)'
#include "a/b/dd.h"	LSEARCH(+)	*userid*.A.B.H(DD)
#include "A.B(C)"	LSEARCH('D.+')	'D.A.B(C)'

When you want to specify a set of sequential data sets as the search path, you add a period followed by an asterisk (.*) at the end of the last qualifier in the *opt*. If you do not have any qualifiers, specify one asterisk (*) as the *opt*. The *opt* has the following syntax for specifying a sequential data set:



where *qualifier* is a data set qualifier.

Start and end the *opt* with single quotation marks (') to indicate that this is an absolute data set specification. Single quotation marks (') around a single asterisk (*) means that the file name that is specified in `#include` is an absolute sequential data set.

The asterisk is replaced by all of the qualifiers and periods in the `#include` *filename* to form the complete name for the search (as shown in the following table).

The following example shows you how to specify a search path for a sequential data set:

Table 20. Sequential data set examples

include Directive	LSEARCH option	Result
#include "A.B"	LSEARCH(CC.*)	*userid*.CC.A.B
#include "a/b/dd.h"	LSEARCH('CC.*')	'CC.DD.H'
#include "a/b/dd.h"	LSEARCH('*')	'DD.H'
#include "a/b/dd.h"	LSEARCH(*)	*userid*.DD.H

Note: If the trailing asterisk is not used in the LSEARCH *opt*, then the specified library is a PDS:

```
#include "A.B"
LSEARCH('CC')
```

Result is `'CC(A)'` which is a PDS.

MAKEDEP

Category

Compiler output

Pragma equivalent

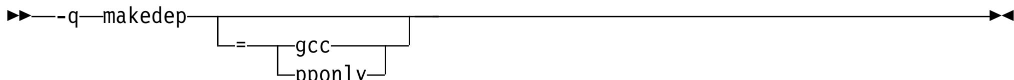
None.

Purpose

Produces the dependency files that are used by the make utility for each source file.

Note: This option is only supported using **-q** syntax. Specifying **-qmakedep** without suboptions is equivalent to the **-M** option, but it behaves differently when specified with a suboption. For more information about the **-M** option, see "Flag options syntax" on page 226.

Syntax



Defaults

Not applicable.

Parameters

gcc
: Instructs the compiler to produce make dependencies file format with a single make rule for all dependencies.

pponly
: Instructs the compiler to produce only the make dependencies file without generating an object file, with the same make file format as the format produced with the gcc suboption.

Usage

For each C source file specified on the command line, an output file is generated with the same name as the object file and the suffix replaced with the suffix for the make dependencies file. The default suffix for the make dependencies file is `.u`. It can be customized using the usuffix attribute in the xlc utility configuration file.

The option only applies to C sources in z/OS UNIX files, because MVS data sets do not have a time stamp required for make utility processing.

If the **-o** option is used to rename the object file, the output file uses the name you specified on the **-o** option.

When **-M** or **-qmakedep** without suboption is specified, the description file contains multiple make rules, one for each dependency. It has the general form:

```
file_name.o: file_name.suffix
file_name.o: include_file_name
```

When **-qmakedep=gcc** or **-qmakedep=pponly** is specified, the description file contains a single make rule for all dependencies. It has the form:

```
file_name.o: file_name.suffix \
 include_file_name
```

Include files are listed according to the search order rules for the `#include` preprocessor directive. If an include file is not found, it is not added to the `.u` file, but if the **-MG** flag option is used, it includes the missing file into the output file. Files with no include statements produce output files containing one line that lists only the input file name.

You can use the **-qmakedep** or **-M** option with the following flag options:

-MF <*file_name*>
: Sets the name of the make dependencies file, where *file_name* is the file name, full path, or partial path for the make dependencies file.

-MG
: When used with the **-qmakedep=pponly** option, **-MG** instructs the compiler to include missing header files into the make dependencies file and suppress diagnostic messages about missing header files.

-MT <*target_name*>
: Sets the target to the <*target_name*> rather than the object file name.

-MQ <*target_name*>
: **-MQ** is the same as **-MT** except that **-MQ** escapes any characters that have special meaning in make.

For more information about the **-MF**, **-MG**, **-MT**, and **-MQ** options, see "Flag options syntax" on page 226.

IPA effects

None.

Predefined macros

None.

Examples

To compile `mysource.c` and create an output file named `mysource.u`, enter:

```
metalc -c -qmakedep mysource.c
```

To compile `foo_src.c` and create an output file named `mysource.u`, enter:

```
metalc -c -qmakedep foo_src.c -MF mysource.u
```

To compile `foo_src.c` and create an output file named `mysource.u` in the `deps/` directory, enter:

```
metalc -c -qmakedep foo_src.c -MF deps/mysource.u
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `foo_obj.u`, enter:

```
metalc -c -qmakedep foo_src.c -o foo_obj.o
```

To compile `foo_src.c` and produce a dependency output file format with a single make rule for all dependencies, enter:

```
metalc -c -qmakedep=gcc foo_src.c
```

To compile `foo_src.c` and produce only the dependency output file without generating an object file, enter:

```
metalc -c -qmakedep=pponly foo_src.c
```

MARGINS | NOMARGINS

Category

Compiler input

Pragma equivalent

`#pragma margins`, **`#pragma nomargins`**

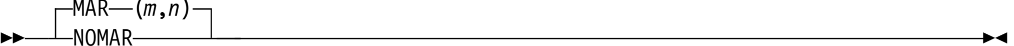
Purpose

Specifies, inclusively, the range of source column numbers that will be compiled.

When the MARGINS option is in effect, you can specify the columns in the input record that are to be scanned for input to the compiler. The compiler ignores text in the source input that does not fall within the range that is specified in the MARGINS option.

When the NOMARGINS options is in effect, the entire input source record will be scanned for input to the compiler.

Syntax



Defaults

- For fixed record format, the default option is MARGINS`(1,72)`.
- For z/OS UNIX file system, the default for a regular compile is NOMARGINS.

Parameters

m
: Specifies the first column of the source input that contains valid C code. The value of *m* must be greater than `0` and less than `32761`.

n
: Specifies the last column of the source input that contains valid C code. The value of *n* must be greater than *m* and less than `32761`. An asterisk (*) can be assigned to *n* to indicate the last column of the input record. If you specify MARGINS`(9,*)`, the compiler scans from column 9 to the end of the record for input source statements.

Usage

You can use the MARGINS and SEQUENCE compiler options together. The MARGINS option is applied first to determine which columns are to be scanned. The SEQUENCE option is then applied to determine which of these columns are not to be scanned. If the SEQUENCE settings do not fall within the MARGINS settings, the SEQUENCE option has no effect.

When a source (or include) file is opened, it initially gets the margins and sequence specified on the command line (or the defaults if none was specified). You can reset these settings by using **`#pragma margins`** or **`#pragma sequence`** at any point in the file. When an `#include` file returns, the previous file keeps the settings it had when it encountered the `#include` directive.

If the MARGINS option is specified along with the SOURCE option in a C program, only the range specified on the MARGINS option is shown in the compiler source listing.

Notes:
1. The MARGINS option does not reformat listings.
2. If your program uses the `#include` preprocessor directive to include Enterprise Metal C for z/OS library header files *and* you want to use the MARGINS option, you must ensure that the specifications on the MARGINS option does not exclude columns `20` through `50`. That is, the value of *m* must be less than `20`, and the value of *n* must be greater than `50`. If your program does not include any Enterprise Metal C for z/OS library header files, you can specify any setting you want on the MARGINS option when the setting is consistent with your own include files.

Predefined macros

None.

Related information

For more information on related compiler options, see
- "SEQUENCE | NOSEQUENCE" on page 127
- "SOURCE | NOSOURCE" on page 134

MAXMEM | NOMAXMEM

Category

Optimization and tuning

Pragma equivalent

#pragma options (maxmem), **#pragma options (nomaxmem)**

Purpose

Limits the amount of memory used for local tables, and that the compiler allocates while performing specific, memory-intensive optimizations, to the specified number of kilobytes.

Syntax

Defaults

MAXMEM(*)

Parameters

size
The valid range for *size* is 0 to 2097152. You can use asterisk as a value for *size* , MAXMEM(*), to indicate the highest possible value, which is also the default. NOMAXMEM is equivalent to MAXMEM(*). Use the MAXMEM *size* suboption if you want to specify a memory size of less value than the default.

Usage

If the memory specified by the MAXMEM option is insufficient for a particular optimization, the compilation is completed in such a way that the quality of the optimization is reduced, and a warning message is issued.

When a large *size* is specified for MAXMEM, compilation may be aborted because of insufficient virtual storage, depending on the source file being compiled, the size of the subprogram in the source, and the virtual storage available for the compilation.

The advantage of using the MAXMEM option is that, for large and complex applications, the compiler produces a slightly less-optimized object module and generates a warning message, instead of terminating the compilation with an error message of "insufficient virtual storage".

Notes:

1. The limit that is set by MAXMEM is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables that are required during the entire compilation process are not affected by or included in this limit.
2. Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
3. Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance.

4. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler may be able to find opportunities to increase performance.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The option value you specify on the IPA compile step for each IPA object file appears in the IPA link step Compiler Options Map listing section.

If you specify the MAXMEM option on the IPA link step, the value of the option is used. The IPA link step Prolog and Partition Map listing sections display the value of the option.

If you do not specify the option on the IPA link step, the value that it uses for a partition is the maximum MAXMEM value you specified for the IPA compile step for any compilation unit that provided code for that partition. The IPA link step Prolog listing section does not display the value of the MAXMEM option, but the Partition Map listing section does.

Predefined macros

None.

MEMORY | NOMEMORY

Category

Compiler customization

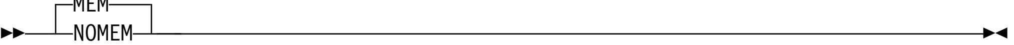
Pragma equivalent

None.

Purpose

Improves compile-time performance by using a memory file in place of a temporary work file, if possible.

Syntax



Defaults

MEMORY

Usage

This option generally increases compilation speed, but you may require additional memory to use it. If you use this option and the compilation fails because of a storage error, you must increase your storage size or recompile your program using the NOMEMORY option. For information on how to increase storage size, see Setting the region size for applications.

IPA effects

The MEMORY option has the same effect on the IPA link step as it does on a regular compilation. If the IPA link step fails due to an out-of-memory condition, provide additional virtual storage. If additional storage is unavailable, specify the NOMEMORY option.

Predefined macros

None.

METAL

Category

Object code control

Pragma equivalent

None.

Purpose

The METAL option is accepted and ignored to allow interoperability with the z/OS XL C compiler.

Syntax

```
►►──METAL──────────────────────────────────────────────────────►◄
```

Defaults

METAL

Usage

Enterprise Metal C for z/OS does not support the NOMETAL option. If NOMETAL is specified, the compiler issues an error for it.

Predefined macros

- __IBM_METAL__ is predefined to 1.
- __IBM_FAR_IS SUPPORTED__ is predefined to 1.

NESTINC | NONESTINC

Category

Compiler input

Pragma equivalent

None.

Purpose

Specifies the number of nested include files to be allowed in your source program.

When the NESTINC compiler option is in effect, you can specify the maximum limit of nested include files.

When the NONESTINC compiler option is in effect, you are specifying NESTINC(255).

Syntax

Defaults

NESTINC(255)

Parameters

num
: You can specify a limit of any integer from 0 to `SHRT_MAX`, which indicates the maximum limit, as defined in the header file `LIMITS.H`. To specify the maximum limit, use an asterisk (*). If you specify an invalid value, the compiler issues a warning message, and uses the default limit, which is 255.

Usage

If you use heavily nested include files, your program requires more storage to compile.

Predefined macros

None.

OE | NOOE

Category

Compiler input

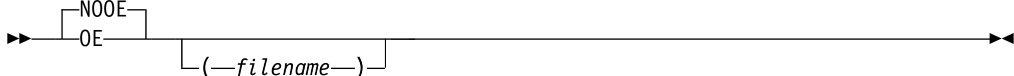
Pragma equivalent

None.

Purpose

Specifies the rules used when searching for files specified with `#include` directives.

Syntax



Defaults

NOOE

When compiling in the z/OS UNIX System Services Shell environment, the default is OE.

Parameters

filename
Specifies the path that is used when searching for files specified with `#include` directives.

Note: Diagnostics and listing information will refer to the file name that is specified for the OE option (in addition to the search information).

Usage

When the OE compiler option is in effect, the compiler uses the POSIX.2 standard rules when searching for files specified with `#include` directives. These rules state that the path of the file currently being processed is the path used as the starting point for searches of include files contained in that file.

The NOOE option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the OE option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOOE.

Example: The following specifications have the same result:

```
metalc hello.c -qnooe=./hello.c -qoe
```

```
metalc hello.c -qoe=./hello.c
```

If you specify OE and NOOE multiple times, the compiler uses the last specified option with the last specified suboption.

Example: The following specifications have the same result:

```
metalc hello.c -qnooe=./hello.c -qoe=./n1.c -qnooe=./test.c -qoe
```

```
metalc hello.c -qoe=./test.c
```

When the OE option is in effect and the main input file is a z/OS UNIX file, the path of *filename* is used instead of the path of the main input file name. If the file names indicated in other options appear ambiguous between z/OS and the z/OS UNIX file system, the presence of the OE option tells the compiler to interpret the ambiguous names as z/OS UNIX file names. User include files that are specified in the main input file are searched starting from the path of *filename*. If the main input file is not a z/OS UNIX file, *filename* is ignored.

For example, if the compiler is invoked to compile a z/OS UNIX file `/a/b/hello.c` it searches directory `/a/b/` for include files specified in `/a/b/hello.c`, in accordance with POSIX.2 rules . If the compiler is invoked with the `OE(/c/d/hello.c)` option for the same source file, the directory specified as the suboption for the OE option, `/c/d/`, is used to locate include files specified in `/a/b/hello.c`.

IPA effects

On the IPA link step, the OE option controls the display of file names.

Predefined macros

None.

OPTFILE | NOOPTFILE

Category

Compiler customization

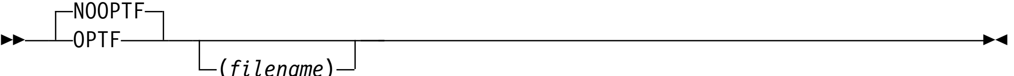
Pragma equivalent

None.

Purpose

Specifies where the compiler should look for additional compiler options.

Syntax



Defaults

NOOPTFILE

Parameters

filename
: Specifies an alternative file where the compiler should look for compiler options.

  You can specify any valid *filename*, including a DD name such as `(DD:MYOPTS)`. The DD name may refer to instream data in your JCL. If you do not specify *filename*, the compiler uses `DD:SYSOPTF`.

Usage

The NOOPTFILE option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the OPTFILE option without a *filename* suboption, the compiler uses the *filename* that you specified with NOOPTFILE earlier.

Example: The following specifications have the same result:

```
metalc -Wc,"NOOPTF(./hello.opt)" -Wc,OPTF hello.c
metalc -Wc,"OPTF(./hello.opt)" hello.c
```

The options are specified in a free format with the same syntax as they would have on the command line or in JCL. The code points for the special characters `\f`, `\v`, and `\t` are whitespace characters. Everything that is specified in the file is taken to be part of a compiler option (except for the continuation character), and unrecognized entries are flagged. Nothing on a line is ignored.

If the record format of the options file is fixed and the record length is greater than 72, columns 73 to the end-of-line are treated as sequence numbers and are ignored.

Notes:

1. Comments are supported in an option file used in the OPTFILE option. When a line begins with the # character, the entire line is ignored, including any continuation character. The option files are encoded in the IBM-1047 code page.
2. You cannot nest the OPTFILE option. If the OPTFILE option is also used in the file that is specified by another OPTFILE option, it is ignored.
3. If you specify NOOPTFILE after a valid OPTFILE, it does not undo the effect of the previous OPTFILE. This is because the compiler has already processed the options in the options file that you specified with OPTFILE. The only reason to use NOOPTFILE is to specify an option file name that a later specification of OPTFILE can use.
4. If the file cannot be opened or cannot be read, a warning message is issued and the OPTFILE option is ignored.
5. The options file can be an empty file.
6. Quotation marks on options (for example, '-O3') in the options file are not removed as they are when specified on the command line.
7. **Example:** You can use an option file only once in a compilation. If you use the following options:

   ```
   OPTFILE(DD:OF)     OPTFILE
   ```

   the compiler processes the option `OPTFILE(DD:OF)`, but the second option `OPTFILE` is not processed. A diagnostic message is produced, because the second specification of `OPTFILE` uses the same option file as the first.

   Example: You can specify OPTFILE more than once in a compilation, if you use a different options file with each specification:

   ```
   OPTFILE(DD:OF)   OPTFILE(DD:OF1)
   ```

IPA effects

The OPTFILE option has the same effect on the IPA link step as it does on a regular compilation.

Predefined macros

None.

Examples

1. Suppose that you use the following JCL:

   ```
   //   CPARM='SO OPTFILE(PROJ1OPT)'
   ```

If the file PROJ1OPT contains LONGNAME, the effect on the compiler is the same as if you specified the following:

```
//   CPARM='SO LONGNAME'
```

2. Suppose that you include the following in the JCL:

```
//   CPARM='OPTFILE(PROJ1OPT) LONGNAME OPTFILE(PROJ2OPT) LIST'
```

If the file PROJ1OPT contains SO LIST, the net effect to the compiler is the same as if you specified the following:

```
//   CPARM='SO LIST LONGNAME LIST'
```

3. The following example shows how to use the options file as an instream file in JCL:

```
//COMP  EXEC MTCC,
//      INFILE='<userid>.USER.CC(LNKLST)',
//      OUTFILE='<userid>.USER.ASM(LNKLST),DISP=SHR ',
//      CPARM='OPTFILE(DD:OPTION)'
//OPTION DD DATA,DLM=@@
   LIST
   MARGINS
   OPT

@@
```

OPTIMIZE | NOOPTIMIZE

Category

Optimization and tuning

Pragma equivalent

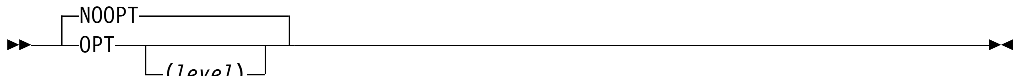
#pragma options (optimize), **#pragma options (nooptimize)**

#pragma option_override(subprogram_name, "OPT(LEVEL,n)")

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax



Defaults

NOOPTIMIZE

When compiling with HOT or IPA, the default is OPTIMIZE(2).

Parameters

level
: *level* can have the following values:

0 Indicates that no optimization is to be done; this is equivalent to NOOPTIMIZE. You should use this option in the early stages of your

application development since the compilation is efficient but the execution is not. This option also allows you to take full advantage of the debugger.

2
Indicates that global optimizations are to be performed. You should be aware that the size of your functions, the complexity of your code, the coding style, and support of the ISO standard may affect the global optimization of your program. You may need significant additional memory to compile at this optimization level.

3
Performs additional optimizations to those performed with OPTIMIZE(2). OPTIMIZE(3) is recommended when the need for runtime improvement outweighs the concern for minimizing compilation resources. Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization. Compilation may require more time and machine resources.

Use the STRICT option with OPTIMIZE(3) to turn off the aggressive optimizations that might change the semantics of a program. STRICT combined with OPTIMIZE(3) invokes all the optimizations performed at OPTIMIZE(2) as well as further loop optimizations. The STRICT compiler option must appear after the OPTIMIZE(3) option, otherwise it is ignored.

The aggressive optimizations performed when you specify OPTIMIZE(3) are:

- Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.
- Conformance to IEEE rules are relaxed. With OPTIMIZE(2), certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception. For example, `X + 0.0` is not folded to `X` because, under IEEE rules, `-0.0 + 0.0 = 0.0`, which is `-X`. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, `X - Y * Z` may result in a `-0.0` where the original computation would produce `0.0`. In most cases, the difference in the results is not important to an application and OPTIMIZE(3) allows these optimizations.
- Floating-point expressions may be rewritten. Computations such as `a*b*c` may be rewritten as `a*c*b` if, for example, an opportunity exits to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

no level
OPTIMIZE specified with no level defaults, depending on the compilation environment and IPA mode.

Usage

When the OPTIMIZE compiler option is in effect, the compiler is instructed to optimize the generated machine instructions to produce a faster running object module. This type of optimization can also reduce the amount of main storage that is required for the generated object module.

Note: When the compiler is invoked using the **metalc** command under z/OS UNIX System Services, the optimization level is specified by the compiler flag **-O** (the letter O). The OPTIMIZE option has no effect on the **metalc** command.

Using OPTIMIZE will increase compile time over NOOPTIMIZE and may have greater storage requirements. During optimization, the compiler may move code to increase runtime efficiency; as a result, statement numbers in the program listing may not correspond to the statement numbers used in runtime messages.

The OPTIMIZE option will control the overall optimization value. Any subprogram-specific optimization levels specified at compile time by **#pragma option_override(subprogram_name, "OPT(LEVEL,n)")** directives will be retained. Subprograms with an OPT(LEVEL,0) value will receive minimal code generation optimization. Subprograms may not be inlined or inline other subprograms. Generate and check the inline report to determine the final status of inlining.

Inlining of functions in conjunction with other optimizations provides optimal runtime performance. The option INLINE is automatically turned on when you specify OPTIMIZE, unless you have explicitly specified the NOINLINE option. See “INLINE | NOINLINE” on page 73 for more information about the INLINE option and the optimization information.

If you specify OPTIMIZE with DEBUG, you can only set breakpoints at function call, function entry, function exit, and function return points. See “DEBUG | NODEBUG” on page 46 for more information about the DEBUG option with optimization.

In the z/OS UNIX System Services environment, **-g** implies NOOPTIMIZE.

Information about the optimization level is inserted in the object file to aid you in diagnosing a problem with your program.

Effect of ANSIALIAS: When the ANSIALIAS option is specified, the optimizer assumes that pointers can point only to objects of the same type, and performs more aggressive optimization. However, if this assumption is not true and ANSIALIAS is specified, wrong program code could be generated. If you are not sure, use NOANSIALIAS.

IPA effects

During a compilation with IPA Compile-time optimizations active, any subprogram-specific optimization levels specified by **#pragma option_override(subprogram_name, "OPT(LEVEL,n)")** directives will be retained. Subprograms with an OPT(LEVEL,0) value will receive minimal IPA and code generation optimization. Subprograms may not be inlined or inline other subprograms. Generate and check the inline report to determine the final status of inlining.

On the IPA compile step, all values (except for (0)) of the OPTIMIZE compiler option and the OPT suboption of the IPA option have an equivalent effect.

OPTIMIZE(2) is the default for the IPA link step, but you can specify any level of optimization. The IPA link step Prolog listing section will display the value of this option.

This optimization level will control the overall optimization value. Any subprogram-specific optimization levels specified at IPA Compile time by **#pragma option_override(subprogram_name, "OPT(LEVEL,n)")** directives will be retained. Subprograms with an OPT(LEVEL,0)) value will receive minimal IPA and code generation optimization, and will not participate in IPA Inlining.

The IPA link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same OPTIMIZE setting.

The OPTIMIZE setting for a partition is set to that of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same OPTIMIZE setting. An OPTIMIZE(0) mode is placed in an OPTIMIZE(0) partition, and an OPTIMIZE(2) is placed in an OPTIMIZE(2) partition.

The option value that you specified for each IPA object file on the IPA compile step appears in the IPA link step Compiler Options Map listing section.

The Partition Map sections of the IPA link step listing and the object module END information section display the value of the OPTIMIZE option. The Partition Map also displays any subprogram-specific OPTIMIZE values.

If you specify OPTIMIZE(2) for the IPA link step, but only OPTIMIZE(0) for the IPA compile step, your program may be slower or larger than if you specified OPTIMIZE(2) for the IPA compile step. This situation occurs because the IPA compile step does not perform as many optimizations if you specify OPTIMIZE(0).

Refer to the descriptions for the OPTIMIZE and LEVEL suboptions of the IPA option in "IPA | NOIPA" on page 75 for information about using the OPTIMIZE option under IPA.

Predefined macros

__OPTIMIZE__ is defined to the value specified by the OPTIMIZE compiler option; it is undefined if NOOPTIMIZE is used.

Related information

For more information about related compiler options, see:

- "DEBUG | NODEBUG" on page 46
- "ANSIALIAS | NOANSIALIAS" on page 24

PHASEID | NOPHASEID

Category

Listings, messages and compiler information

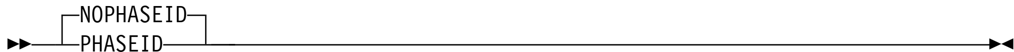
Pragma equivalent

None.

Purpose

Causes each compiler component (phase) to issue an informational message as each phase begins execution, which assists you with determining the maintenance level of each compiler component (phase). This message identifies the compiler phase module name, product identification, and build level.

Syntax



Defaults

NOPHASEID

Usage

The compiler issues a separate CJT0000(I) message each time compiler execution causes a given compiler component (phase) to be entered. This could happen many times for a given compilation.

The FLAG option has no effect on the PHASEID informational message.

In the z/OS UNIX System Services environment, **-qphsinfo** is synonymous with the PHASEID compiler option.

Note: The compiler saves phase ID information for all active compiler phases in an executable using the Saved Option String feature even if you don't specify the PHASEID compiler option.

Predefined macros

None.

PPONLY | NOPPONLY

Category

Compiler output

Pragma equivalent

None.

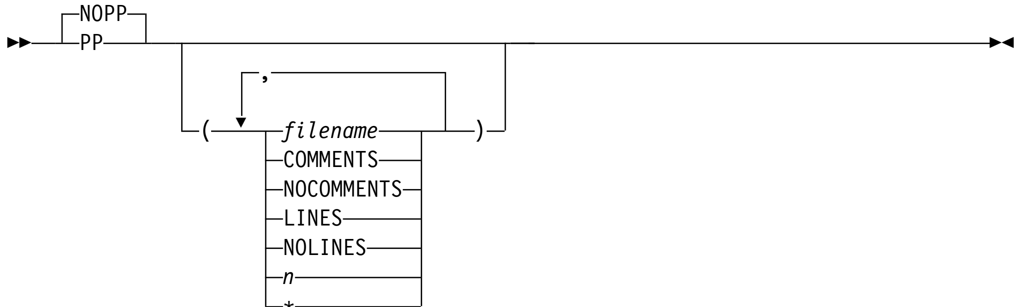
Purpose

Specifies that only the preprocessor is to be run and not the compiler.

When the PPONLY compiler option is in effect, the output of the preprocessor consists of the original source file with all the macros expanded and all the include files inserted. It is in a format that can be compiled.

When the NOPPONLY compiler option is in effect, both the preprocessor and the compiler are used to compile the source file.

Syntax



Defaults

NOPPONLY

For the z/OS UNIX System Services utilities, the default for a regular compile is NOPPONLY(NOCOMMENTS, NOLINES, /dev/fd1, 2048).

When using the **metalc** utility, this option can be turned on by specifying the -E or -P flag options, or by specifying the **-qpponly** compiler option in a manner similar to specifying the PPONLY option in JCL or TSO compiler invocations.

Parameters

COMMENTS | NOCOMMENTS
: The COMMENTS suboption preserves comments in the preprocessed output. The default is NOCOMMENTS.

LINES | NOLINES
: The LINES suboption issues `#line` directives at include file boundaries, block boundaries and where there are more than 3 blank lines. The default is NOLINES.

filename
: The name for the preprocessed output file. The *filename* may be a data set or a z/OS UNIX file. If you do not specify a file name for the PPONLY option, the `SYSUT10` ddname is used if it has been allocated. If `SYSUT10` has not been allocated, the file name is generated as follows:

- If a data set is being compiled, the name of the preprocessed output data set is formed using the source file name. The high-level qualifier is replaced with the userid under which the compiler is running, and `.EXPAND` is appended as the low-level qualifier.
- If the source file is a z/OS UNIX file, the preprocessed output is written to a z/OS UNIX file that has the source file name with `.i` extension.

Note: If you are using the xlc utility and you do not specify the file name, the preprocessed output goes to `stdout`. If -E or -P is also specified, the output file is determined by the -E option. The -E flag option maps to `PP(stdout)`. -P maps to PP(default_name). default_name is constructed using the source file name as the base and the suffix is replaced with the appropriate suffix, as defined by the `isuffix`, `isuffix_host`, `ixxsuffix`, and `ixxsuffix_host` configuration file attributes. See Chapter 14, “metalc — Compiler invocation using a customizable configuration file,” on page 219 for further information on the **metalc** utility.

n If a parameter *n*, which is an integer between 2 and 32752 inclusive, is specified, all lines are folded at column *n*. The default for *n* is 72.

Note: If the PPONLY output is directed into an existing file, and *n* is larger than the maximum record length of the file, then all lines are folded to fit into the output file, based on the record length of the output file.

* If an asterisk (*) is specified, the lines are folded at the maximum record length of 32752. Otherwise, all lines are folded to fit into the output file, based on the record length of the output file.

Usage

PPONLY output is typically requested when reporting a compiler problem to IBM using a Problem Management Record (PMR), so your build process should be able to produce a PPONLY file on request.

Note: For further information on the PMR process, refer to the Software Support Handbook (techsupport.services.ibm.com/guides/handbook.html).

PPONLY also removes conditional compilation constructs like `#if`, and `#ifdef`.

Note: If the PPONLY output is directed into an existing file, the record length of the file will be used to override the value of *n* if that value is bigger than the maximum record length.

The PPONLY suboptions are cumulative. If you specify suboptions in multiple instances of PPONLY and NOPPONLY, all the suboptions are combined and used for the last occurrence of the option.

Example: The following three specifications have the same result:

```
metalc -Wc,"NOPPONLY(./aa.exp)" -Wc,"PPONLY(LINES)" -Wc,"PPONLY(NOLINES)" hello.c
metalc -Wc,"PPONLY(./aa.exp,LINES,NOLINES)" hello.c
metalc -Wc,"PPONLY(./aa.exp,NOLINES)" hello.c
```

All `#line` and **`#pragma`** preprocessor directives (except for `margins` and `sequence` directives) remain. When you specify PPONLY(*), `#line` directives are generated to keep the line numbers generated for the output file from the preprocessor similar to the line numbers generated for the source file. All consecutive blank lines are suppressed.

If you specify the PPONLY option, the compiler turns on the TERMINAL option. If you specify the SHOWINC, AGGREGATE, or EXPMAC options with the PPONLY option, the compiler issues a warning, and ignores the options.

If you specify the PPONLY and LOCALE options, all the **`#pragma filetag`** directives in the source file are suppressed. The compiler generates its **`#pragma filetag`** directive at the first line in the preprocessed output file in the following format:

```
??=pragma filetag ("locale code page")
```

In this example, `??=` is a trigraph representation of the `#` character.

The code page in the pragma is the code set that is specified in the LOCALE option.

If you specify both PPONLY and NOPPONLY, the last one that is specified is used.

In the z/OS UNIX environment, the COMMENTS suboption can be requested by specifying the -C flag option. When using the **metalc** utility, the PPONLY option can be specified in addition to the -E, -P and -C flag options (for example, `-qpponly=myfunc.pp:comments:nolines:65`).

Note: **-Wc,PPONLY** syntax is not supported.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- “TERMINAL | NOTERMINAL” on page 144
- “SHOWINC | NOSHOWINC” on page 131
-
- “AGGREGATE | NOAGGREGATE” on page 23
- “EXPMAC | NOEXPMAC” on page 56

PREFETCH | NOPREFETCH

Category

Optimization and tuning

Pragma equivalent

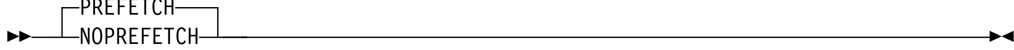
None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When PREFETCH is in effect, the compiler may insert prefetch instructions in compiled code. When NOPREFETCH is in effect, prefetch instructions are not inserted in compiled code.

Syntax



Defaults

PREFETCH

Usage

The compiler will attempt to generate prefetch instructions for ARCH(8) or above. The compiler will not issue a message if PREFETCH is active and the ARCH level is below 8.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

PROLOG

Category

Object code control

Pragma equivalent

#pragma prolog

Purpose

Enables you to provide your own function entry code for all functions that have `extern` scope, or for all extern and static functions.

Syntax

```
►►──PROLOG──(──┬─"text-string"──────────────────┬─)──────────────────────────►◄
               └─┬─EXTERN─┬─(─"text-string"─)─┘
                 └─ALL────┘
```

Defaults

The compiler generates default prolog code for the functions that do not have user-supplied prolog code.

Parameters

text-string

text-string is a C string, which must contain valid HLASM statements.

If the *text-string* consists of white-space characters only, or if the *text-string* is not provided, then the compiler ignores the option specification. If the *text-string* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the compiler will insert the *text-string* into the function prolog location of the generated assembler source. The compiler does not understand or validate the contents of the *text-string*. In order to satisfy the assembly step later, the given *text-string* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like `newline` and `quote` are shell (or command line) meta characters, and may be preprocessed before reaching the compiler. It is

advisable to avoid using them. The intended use of this option is to specify an assembler macro as the function prolog.

For information on valid HLASM statements, see #pragma prolog.

EXTERN
If the PROLOG option is specified with this suboption or without any suboption, the prolog applies to all functions that have external linkage in the compilation unit.

ALL
If the PROLOG option is specified with this suboption, the prolog also applies to static functions defined in the compilation unit.

Usage

Notes:

1. When the PROLOG option is specified multiple times with the same suboption **all** or **extern**, only the function entry code of the last suboption specified will be displayed.
2. The PROLOG option with the suboption **all** overwrites the one with **extern** suboption, or the one without any suboption.

IPA effects

See Building Metal C programs with IPA in *Enterprise Metal C for z/OS User's Guide*.

Predefined macros

None.

Related information

See "EPILOG" on page 53 for information on providing function exit code for system development.

RENT | NORENT

Category

Object code control

Pragma equivalent

#pragma options (rent), **#pragma options (norent)**

#pragma variable(rent), **#pragma variable(norent)**

Purpose

Generates reentrant code.

When the RENT compiler option is in effect, the compiler takes code that is not naturally reentrant and make it reentrant. Refer to *z/OS Language Environment Programming Guide* for a detailed description of reentrancy.

When the NORENT compiler option is in effect, the compiler does not generate reentrant code from non-reentrant code. Any naturally reentrant code remains reentrant.

Syntax

Defaults

NORENT.

Usage

If you use the RENT option, the linkage editor cannot directly process the object module that is produced. You must use the binder, which is described in Chapter 6, “Binding programs,” on page 193.

The RENT option can be enabled to support constructed reentrancy for C programs with writable static and external variables. The writable static area (WSA) can be managed by user provided initialization and termination functions.

Notes:

1. Enterprise Metal C for z/OS code always uses constructed reentrancy so the RENT option is always in effect.
2. RENT variables reside in the modifiable Writable Static Area (WSA) for Enterprise Metal C for z/OS programs.
3. NORENT variables reside in the code area (which might be write protected) for Enterprise Metal C for z/OS programs.
4. The RENT compiler option has implications on how the binder processes objects. See *z/OS MVS Program Management: User's Guide and Reference* for further information.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

If you specify RENT or use **#pragma strings(readonly)** or **#pragma variable(rent | norent)** during the IPA compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the RENT option on the IPA link step, it ignores the option. The reentrant/nonreentrant state of each symbol is maintained during IPA optimization and code generation. If any symbols within a partition are reentrant, the option section of the Partition Map displays the RENT compiler option.

If you generate an IPA Link listing by using the LIST or IPA(MAP) compiler option, the IPA link step generates a Partition Map listing section for each partition. If any symbols within a partition are reentrant, the options section of the Partition Map displays the RENT compiler option.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- "LIST | NOLIST" on page 83
- "IPA | NOIPA" on page 75

RESERVED_REG

Category

Object code control

Pragma equivalent

None.

Purpose

Instructs the compiler not to use the specified general purpose register (GPR) during the compilation.

Syntax

Defaults

Not specified.

Parameters

reg_name
: Only the general purpose registers 0-15 (written as r0, r1, ..., r15 or R0, R1, ...,R15) can be specified for the RESERVED_REG option. Any other name is rejected with a warning message. Some general purpose registers have designated roles in the compiler for generating program code, and reserving these registers may prevent the compiler from generating the correct code. See Table 21 on page 120 for further information on z/OS general purpose registers that have designated roles for the Enterprise Metal C for z/OS compiler.

Usage

A global register variable declaration reserves the register for the declared variable in the compilation unit where the declaration appears. The register is not reserved in other compilation units unless the global register declaration is placed in a common header file.

Notes:
1. Duplicate register names are ignored silently.

2. The RESERVED_REG option is cumulative, which means that, for example:

```
-qreserved_reg=r14 -qreserved_reg=r15
```

is equivalent to:

```
-qreserved_reg=r14:r15
```

Table 21. General purpose registers that have designated roles for the Enterprise Metal C for z/OS compiler

Register	Designated role
r0	volatile
r1	parameter list pointer
r3	designated by the compiler
r10	used by the C generated code for addressing data
r11	used by the C generated code for addressing data
r13	savearea pointer (C: stack pointer)
r14	function return address
r15	function entry point on entry, return code on exit. (C: integral type return value)

IPA effects

See Building Metal C programs with IPA in *Enterprise Metal C for z/OS User's Guide*.

Predefined macros

None.

RESTRICT | NORESTRICT

Category

Optimization and tuning

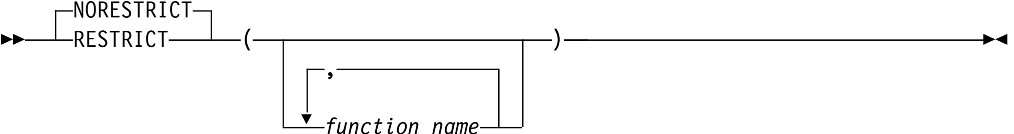
Pragma equivalent

None.

Purpose

Indicates to the compiler that no other pointers can access the same memory that has been addressed by function parameter pointers.

Syntax



Defaults

NORESTRICT

When NORESTRICT is in effect, no function parameter pointers are restricted unless the `restrict` attribute is specified in the source.

Parameters

function_name is a comma-separated list. If you do not specify the *function_name*, parameter pointers in all functions are treated as restrict. Otherwise, only those parameter pointers in the listed functions are treated as restrict.

Usage

The RESTRICT option indicates to the compiler that pointer parameters in all functions or in specified functions are disjoint. This is equivalent to adding the `restrict` keyword to the parameter pointers within the required functions, but without having to modify the source file. When RESTRICT is in effect, deeper pointer analysis is done by the compiler and performance of the application being compiled is improved.

Note that incorrectly asserting this pointer restriction might cause the compiler to generate incorrect code based on the false assumption. If the application works correctly when recompiled without the RESTRICT option, the assertion might be incorrect. In this case, this option should not be used.

Note: When RESTRICT and NORESTRICT are specified multiple times, the last option specified on the command line takes precedence over any previous specifications.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

ROCONST | NOROCONST

Category

Object code control

Pragma equivalent

```
#pragma variable(var_name, NORENT)
```

Purpose

Specifies the storage location for constant values.

When the ROCONST compiler option is in effect, the compiler places constants in read-only storage, even if the RENT option is in effect. Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access.

When the NOROCONST compiler option is in effect, constant values are placed in read/write storage.

Syntax

For C:

Defaults

The default option is NOROCONST.

Usage

The ROCONST option informs the compiler that the `const` qualifier is respected by the program. Variables defined with the `const` keyword will not be overridden by a casting operation.

Note that these `const` variables cannot be exported.

If the specification for a `const` variable in a **#pragma variable** directive is in conflict with the option, the **#pragma variable** takes precedence. The compiler issues an informational message.

If you set the ROCONST option, and if there is a **#pragma export** for a `const` variable, the pragma directive takes precedence. The compiler issues an informational message. The variable will still be exported and the variable will be reentrant.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

If you specify the ROCONST option during the IPA compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the ROCONST option on the IPA link step, it ignores the option. The reentrant or non-reentrant and `const` or non-`const` state of each symbol is maintained during IPA optimization and code generation.

The IPA link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same ROCONST setting.

The ROCONST setting for a partition is set to the specification of the first subprogram that is placed in the partition.

The option value that you specified for each IPA object file on the IPA compile step appears in the IPA link step Compiler Options Map listing section.

The RENT, ROCONST, and ROSTRING options both contribute to the re-entrant or non-reentrant state for each symbol.

The Partition Map sections of the IPA link step listing and the object module END information section display the value of the ROCONST option.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- "RENT | NORENT" on page 117
- "ROSTRING | NOROSTRING"

ROSTRING | NOROSTRING

Category

Object code control

Pragma equivalent

```
#pragma strings(readonly)
```

Purpose

Specifies the storage type for string literals.

When the ROSTRING compiler option is in effect, the compiler places string literals in read-only storage. Placing string literals in read-only memory can improve runtime performance and save storage.

When the NOROSTRING compiler option is in effect, string literals are placed in read/write storage.

Syntax

Defaults

ROSTRING

Usage

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

If you specify the ROSTRING option during the IPA compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the ROSTRING option on the IPA link step, it ignores the option. The reentrant or nonreentrant state of each symbol is maintained during IPA optimization and code generation.

The Partition Map section of the IPA link step listing and the object module do not display information about the ROSTRING option for that partition. The RENT, ROCONST, and ROSTRING options all contribute to the reentrant or nonreentrant state for each symbol. If any symbols within a partition are reentrant, the option section of the Partition Map displays the RENT compiler option.

Predefined macros

None.

Related information

For more information on related compiler options, see:
- "RENT | NORENT" on page 117
- "ROCONST | NOROCONST" on page 121

ROUND

Category

Floating-point and integer control

Pragma equivalent

None.

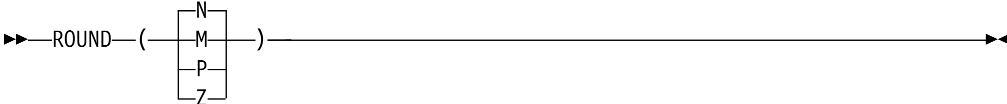
Purpose

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Syntax

The ROUND option is only valid when used with a base 2 IEEE-754 binary format (specified by the FLOAT(IEEE) compiler option) or base 16 z/Architecture hexadecimal format (specified by the FLOAT(HEX) compiler option).

When FLOAT(IEEE) is specified:



When FLOAT(HEX) is specified:

```
►►──ROUND──(──Z──)──────────────────────────────────────────►◄
```

Defaults

- For FLOAT(IEEE), the default option is ROUND(N).
- For FLOAT(HEX), the default option is ROUND(Z).

Parameters

When FLOAT(IEEE) is in effect, the following modes are valid:

N round to the nearest representable number (ties to even)

Note: A *tie* occurs when the number to be rounded is at the exact midpoint between two values towards which it can be rounded. For example, if we are rounding to the nearest representable whole number, and we are given the value 1.5, we are at the exact midpoint between the two nearest whole numbers (2 and 1). This is considered a tie. In this example, and using ties to even, we would round the value 1.5 to the value 2, as 2 is an even number.

M round towards minus infinity

P round towards positive infinity

Z round towards zero

Note: ROUND() is the same as ROUND(N).

Usage

You can specify a rounding *mode* only when you use IEEE floating-point mode. In hexadecimal mode, the rounding is always towards zero.

You must ensure that you are in the same rounding mode at compile time (specified by the ROUND(*mode*) option), as at run time. Entire compilation units will be compiled with the same rounding mode throughout the compilation. If you switch runtime rounding modes inside a function, your results may vary depending upon the optimization level used and other characteristics of your code; use caution if you switch rounding mode inside functions.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these section is a partition. The IPA link step uses information from the IPA compile step to ensure that an object is included in a compatible partition.

Predefined macros

None.

Related information

For information about related compiler options, see:

- “FLOAT” on page 58

SEARCH | NOSEARCH

Category

Compiler input

Pragma equivalent

None.

Purpose

Specifies the directories or data sets to be searched for system include files.

When the SEARCH compiler option is in effect, the preprocessor looks for system include files in the specified directories or data sets. System include files are those files that are associated with the `#include <filename>` form of the `#include` preprocessor directive. See "Using include files" on page 173 for a description of the `#include` preprocessor directive.

When the NOSEARCH compiler option is in effect, the preprocessor searches only those data sets that are specified in the SYSLIB statement.

Syntax

Defaults

SEARCH(/usr/include/metal)

Parameters

The suboptions for the SEARCH option are identical to those for the LSEARCH option. For information on the LSEARCH option, see "LSEARCH | NOLSEARCH" on page 91.

Usage

The SYSLIB ddname is considered the last suboption for SEARCH, so that specifying SEARCH `(X)` is equivalent to specifying SEARCH(`X,DD:SYSLIB`).

Any NOSEARCH option cancels all previous SEARCH specifications, and any new SEARCH options that follow it are used. When more than one SEARCH compile option is specified, all directories or data sets in the SEARCH options are used to find the system include files.

Notes:

1. SEARCH allows the compiler to distinguish between header files that have the same name but reside in different data sets. If NOSEARCH is in effect, the compiler searches for header files only in the data sets concatenated under the

SYSLIB DD statement. As the compiler includes the header files, it uses the first file it finds, which may not be the correct one. Thus the build may encounter unpredictable errors in the subsequent link-edit or bind, or may result in a malfunctioning application.
2. If the *filename* in the `#include` directive is in absolute form, searching is not performed. See "Determining whether the file name is in absolute form" on page 178 for more details on absolute `#include` *filename*.

IPA effects

The SEARCH option is used for source code searching, and has the same effect on an IPA compile step as it does on a regular compilation.

The IPA link step accepts the SEARCH option, but ignores it.

Predefined macros

None.

Related information

For further information on library search sequences, see "Search sequences for include files" on page 182.

SEQUENCE | NOSEQUENCE

Category

Compiler input

Pragma equivalent

`#pragma sequence`, **`#pragma nosequence`**

Purpose

Specifies the columns used for sequence numbers.

Syntax

For fixed record format, variable record format, and the z/OS UNIX file system:

```
     ┌─SEQ──(──m,n──)─┐
►►───┴─NOSEQ──────────┴──────────────────────────────────────────►◄
```

Defaults

- For variable record format and the z/OS UNIX file system, the default is NOSEQUENCE.
- For fixed record format, the default is SEQUENCE(`73,80`).

Parameters

m Specifies the column number of the left-hand margin. The value of *m* must be greater than `0` and less than `32760`.

n Specifies the column number of the right-hand margin. The value of *n* must be

greater than *m* and less than 32760. An asterisk (*) can be assigned to *n* to indicate the last column of the input record. Thus, SEQUENCE (74,*) shows that sequence numbers are between column 74 and the end of the input record.

Usage

When the SEQUENCE compiler option is in effect, it defines the section of the input record that is to contain sequence numbers. No attempt is made to sort the input lines or records into the specified sequence or to report records out of sequence.

You can use the MARGINS and SEQUENCE options together. The MARGINS option is applied first to determine which columns are to be scanned. The SEQUENCE option is then applied to determine which of these columns are not to be scanned. If the SEQUENCE settings do not fall within the MARGINS settings, the SEQUENCE option has no effect.

Note: If your program uses the `#include` preprocessor directive to include Enterprise Metal C for z/OS library header files *and* you want to use the SEQUENCE option, you must ensure that the specifications on the SEQUENCE option do not include any columns from 20 through 50. That is, both *m* and *n* must be less than 20, or both must be greater than 50. If your program does not include any Enterprise Metal C for z/OS library header files, you can specify any setting you want on the SEQUENCE option when the setting is consistent with your own include files.

Predefined macros

None.

Related information

"MARGINS | NOMARGINS" on page 99

SERVICE | NOSERVICE

Category

Error checking and debugging

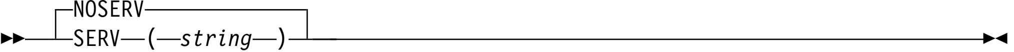
Pragma equivalent

#pragma options(service), **#pragma options(noservice)**

Purpose

Places a *string* in the object module, which is displayed in the traceback if the application fails abnormally.

Syntax



Defaults

NOSERVICE

Parameters

string
User-specified string of characters.

Usage

When the SERVICE compiler option is in effect, the *string* in the object module is loaded into memory when the program is executing. If the application fails abnormally, the *string* is displayed in the traceback.

If the SERVICE option is specified both on a **#pragma options** directive and on the command line, the option that is specified on the command line will be used.

You must enclose your *string* within opening and closing parentheses. You do not need to include the *string* in quotation marks.

The following restrictions apply to the *string* specified:

- The *string* cannot exceed 64 characters in length. If it does, excess characters are removed, and the *string* is truncated to 64 characters. Leading and trailing blanks are also truncated.

  Note: Leading and trailing spaces are removed first and then the excess characters are truncated.
- All quotation marks that are specified in the *string* are removed.
- All characters, including DBCS characters, are valid as part of the *string* provided they are within the opening and closing parentheses.
- Parentheses that are specified as part of the *string* must be balanced. That is, for each opening parentheses, there must be a closing one. The parentheses must match after truncation.
- When using the **#pragma options** directive, the text is converted according to the locale in effect.
- Only characters which belong to the invariant character set should be used, to ensure that the signature within the object module remains readable across locales.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

If you specify the SERVICE option on the IPA compile step, or specify **#pragma options(service)** in your code, it has no effect on the IPA link step. Only the SERVICE option you specify on the IPA link step affects the generation of the service string for that step.

Predefined macros

None.

SEVERITY | NOSEVERITY

Category

Listings, messages, and compiler information

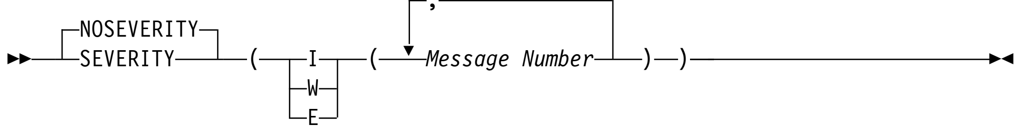
Pragma equivalent

None.

Purpose

Changes the default severities for certain user-specified messages, if these messages are generated by the compiler.

Syntax



Defaults

NOSEVERITY

When NOSEVERITY is in effect, all the previous message severity changes are cleared.

Parameters

I Specifies the message severity level of informational (I).

W Specifies the message severity level of warning (W).

E Specifies the message severity level of error (E).

Message Number
Represents a valid compiler message number, which must be in the following format:

`abc****`

Where:

- `abc` is the three-letter code prefix representing the message types.
- `****` is the four-digit message number.

Usage

The SEVERITY option allows you to set the severity for certain messages that you specified. The compiler will use the new severity if the specified messages are generated by the compiler. You can use this option to match your build process rules for cases which are known not to be problems.

The new severity can be higher or lower than the default compiler severity. When you decrease message severities, you can only decrease informational (I) and warning (W) messages. The (E) level messages cannot be decreased.

Note: When multiple severities are specified for one message, the last valid severity specified on the command line takes precedence over any previous valid specifications.

Predefined macros

None.

Examples

If your program `prototype.c` normally results in the following output:

```
WARNING CJT3304 ./prototype.c:2 No function prototype given for "malloc".
```

You can decrease the severity of the message to INFORMATIONAL by compiling with:

```
metalc prototype.c -qseverity=i=CJT3304
```

SHOWINC | NOSHOWINC

Category

Listings, messages and compiler information

Pragma equivalent

None.

Purpose

When used with SOURCE option to generate a listing file, selectively shows user and system header files in the source program section of the listing file.

Syntax

Defaults

NOSHOWINC

Usage

In the listing, the compiler replaces all `#include` preprocessor directives with the source that is contained in the include file.

The SHOWINC option has effect only if the SOURCE option is also in effect.

Predefined macros

None.

Related information

For more information on the SOURCE compiler option, see "SOURCE | NOSOURCE" on page 134.

SHOWMACROS | NOSHOWMACROS

Category

Compiler output

Pragma equivalent

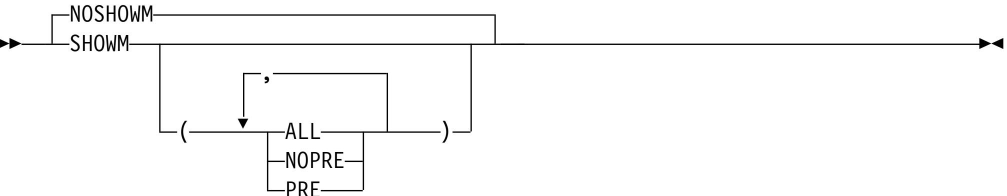
None.

Purpose

Displays macro definitions to preprocessed output.

Displaying macros to preprocessed output can help to determine the available functionality in the compiler. The macro listing may prove useful in debugging complex macro expansions.

Syntax



Defaults

NOSHOWMACROS

The SHOWMACROS option replaces the preprocessed output with the macro define directives.

Parameters

ALL
: Emits all macro definitions to preprocessed output. This is the same as specifying SHOWMACROS.

PRE
: Emits only predefined macro definitions to preprocessed output. This suboption has no impact on user macros.

NOPRE
: Suppresses appending predefined macro definitions to preprocessed output.

Usage

Specifying SHOWMACROS with no suboptions is equivalent to SHOWMACROS(ALL).

Specify SHOWMACROS(ALL,NOPRE) to emit only the user defined macros.

Note the following information when using this option:
- This option has no effect unless preprocessed output is generated; for example, using the **-qpponly** option in the **metalc** utility, or using the **PPONLY** option through JCL and TSO.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Predefined macros

None.

SKIPSRC

Category

Listings, messages, and compiler information

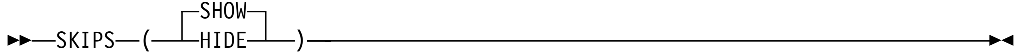
Pragma equivalent

None.

Purpose

When a listing file is generated using the SOURCE option, SKIPSRC option can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file.

Syntax



Defaults

SKIPSRC(SHOW)

Parameters

SHOW
Shows all source statements in the listing.

HIDE
Hides the source statements skipped by the compiler. This improves the readability of the listing file.

Usage

The SKIPSRC option has effect only if the SOURCE option is also in effect. For information on the SOURCE options, see "SOURCE | NOSOURCE" on page 134.

Predefined macros

None.

SOURCE | NOSOURCE

Category

Listings, messages and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes the source section of the listing.

Syntax

```
      ┌─NOSO─┐
►►────┴─SO───┴──┬─────────────────────────────────────────────────┬──────────────►◄
                └─(──┬─Sequential filename──────────────────┬─)─┘
                     ├─Partitioned data set─────────────────┤
                     ├─Partitioned data set (member)────────┤
                     ├─z/OS UNIX System Services filename───┤
                     └─z/OS UNIX System Services directory──┘
```

Defaults

NOSOURCE

Parameters

Sequential filename
Specifies the sequential data set file name for the compiler listing.

Partitioned data set
Specifies the partitioned data set for the compiler listing.

Partitioned data set (member)
Specifies the partitioned data set (member) for the compiler listing.

z/OS UNIX System Services filename
Specifies the z/OS UNIX System Services file name for the compiler listing.

z/OS UNIX System Services directory
Specifies the z/OS UNIX System Services directory for the compiler listing.

Usage

If you specify SOURCE(*filename*), the compiler places the listing in the file that you specified. If you do not specify a file name for the SOURCE option, the compiler uses the SYSCPRT ddname if you allocated one. Otherwise, the compiler constructs the file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the listing data set. The high-level qualifier is replaced with the userid under which the compiler is running, and `.LIST` is appended as the low-level qualifier.
- If the source file is a z/OS UNIX file, the listing is written to a file that has the name of the source file with a `.lst` extension in the current working directory.

The NOSOURCE option can optionally take a file name suboption. This file name then becomes the default. If you subsequently use the SOURCE option without a file name suboption, the compiler uses the file name that you specified in the earlier NOSOURCE.

Example: The following specifications have the same result:

```
metalc -Wc,"NOSO(./hello.lis)" -Wc,SO hello.c
```

```
metalc -Wc,"SO(./hello.lis)"
```

If you specify SOURCE and NOSOURCE multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
metalc -Wc,"NOSO(./hello.lis)" -Wc,"SO(./n1.lis)" -Wc,"NOSO(./test.lis)" -Wc,SO hello.c
```

```
metalc -Wc,"SO(./test.lis)" hello.c
```

Notes:

- If you use the following form of the command in a JES3 batch environment where xxx is an unallocated data set, you may get undefined results.

  ```
  SOURCE(xxx)
  ```

- If you specify data set names with the SOURCE or LIST option, the compiler combines all the listing sections into the last data set name specified.

Predefined macros

None.

SPLITLIST | NOSPLITLIST

Category

Listings, messages, and compiler information

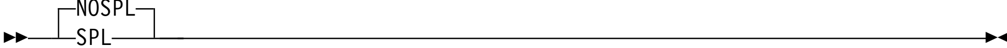
Pragma equivalent

None.

Purpose

Enables the compiler to write the IPA Link phase listing to multiple PDS members, PDSE members, or z/OS UNIX files. The SPLITLIST compiler option has no effect unless the LIST compiler option is also specified.

Syntax



Defaults

NOSPLITLIST

Usage

Normally, the default listing location is `stdout` or SYSCPRT. You can instruct the compiler to output listing contents into a file by using the LIST option. This method can be useful when the source file is large and there is a large amount of detail in the listing. Writing the listing contents to a file, allows you to use an editor or a search utility to browse through the file. However, for the IPA Link phase, which processes the whole application instead of just one source file, there are situations when the listing file itself becomes too large, which can cause difficulties for an editor or search utility. The SPLITLIST option is designed to split a listing into multiple files so that it will be easier for you to browse and edit large listings.

The SPLITLIST option is used only in the IPA Link phase, and the location of the files, which must be a PDS, PDSE, or z/OS UNIX file system directory, must be specified by the LIST option. If the LIST option is not used to specify a location, you will receive an error message.

Table 22 shows the names given to the generated listing sections if a z/OS UNIX file system directory name is specified. In the table, we assume the location is a directory called `listing`, and there are three partitions generated by the IPA Link phase.

Table 22. Listing section names comparison for a specified z/OS UNIX file system directory

Listing section names generated with SPLITLIST	Listing section names generated with NOSPLITLIST
listing/part0	Partition 0 listing
listing/part1	Partition 1 listing
listing/part2	Partition 2 listing
listing/objmap	Object File Map
listing/srcmap	Source File Map
listing/inlrpt	Inline Report
listing/options	IPA Link Options
listing/cuopts	Compiler Options Map
listing/globsym	Global Symbols Map
listing/messages	Messages and Summary

Table 23 shows the names given to the generated listing sections if a PDS or PDSE name is specified. In the table, we assume the PDS or PDSE name is `ACCNTING.LISTING`, and that three partitions are generated by the IPA Link phase.

Table 23. Listing section names comparison for a specified PDS name

Listing section names generated with SPLITLIST	Listing section names generated with NOSPLITLIST
ACCNTING.LISTING(PART0)	Partition 0 listing
ACCNTING.LISTING(PART1)	Partition 1 listing
ACCNTING.LISTING(PART2)	Partition 2 listing

Table 23. Listing section names comparison for a specified PDS name (continued)

Listing section names generated with SPLITLIST	Listing section names generated with NOSPLITLIST
ACCNTING.LISTING(OBJMAP)	Object File Map
ACCNTING.LISTING(SRCMAP)	Source File Map
ACCNTING.LISTING(INLRPT)	Inline Report
ACCNTING.LISTING(OPTIONS)	IPA Link Options
ACCNTING.LISTING(CUOPTS)	Compiler Options Map
ACCNTING.LISTING(GLOBSYM)	Global Symbols Map
ACCNTING.LISTING(MESSAGES)	Messages and Summary

Notes:

1. The SPLITLIST option can only be specified in the IPA Link phase.
2. Repeating a SPLITLIST option is equivalent to specifying it once. The last one specified is the effective setting.
3. If the SPLITLIST option is specified but the effective location of the listing is not a z/OS UNIX file system directory, PDS data set, or PDSE data set, then a diagnostic message will be issued and the IPA Link phase return code will be at least 8.
4. A z/OS UNIX file system directory name must denote a z/OS UNIX directory which exists and is accessible by the user prior to the IPA Link. Otherwise, a diagnostic message will be issued and the minimum return code will be raised to 16.
5. The PDS name must denote a PDS or PDSE data set which exists and is accessible by the user prior to the IPA Link. Otherwise, a diagnostic message will be generated and the minimum return code will be raised to 16.

IPA effects

The SPLITLIST option will be ignored by the IPA Compile phase (since it does not generate a listing). If **-Wc,SPLITLIST** is used, the IPA compile step will ignore it.

Predefined macros

None.

Examples

The following examples show how to use SPLITLIST.

Example 1

```
# list must exist prior to executing the IPA link
#
mkdir list

# Generate listing sections corresponding to LIST
#
metalc -Wc,"LIST(./list)" -Wc,SPLITLIST -o a.out hello.o
```

Example 2

```
# list must exist prior to executing the IPA link
#
mkdir list

# Since NOLIST is specified, only IPA(MAP)  sections are generated
#  However, the destination directory is the one specified in the NOLIST option
#
metalc -Wc,SPLITLIST -Wc,'NOLIST(./list)' -WI,MAP -o a.out hello.o
```

The following provides a JCL example for SPLITLIST:

```
//USRID1A  JOB (359B,2326),'USRID1',
//          MSGLEVEL=(1,1),MSGCLASS=S,CLASS=A,NOTIFY=USRID1
/*JOBPARM   T=1,L=300
//ORDER     JCLLIB ORDER=(CJT.SCJTPRC)
//*--------------------------------------------------------------------
//* Compile
//*--------------------------------------------------------------------
//C0011L01  EXEC MTCC,
//          OUTFILE='USRID1.PASS1.OBJECT(SPLLIST),DISP=SHR',
// PARM.COMPILE=('IPA(NOLINK) OPT',
//   'RENT LO ')
//SYSIN    DD *,DLM='/>'
int main()
{
    return 0;
}
/>
//*--------------------------------------------------------------------
//* IPA LINK
//*--------------------------------------------------------------------
//C0011L02  EXEC MTCI,
//          OUTFILE='USRID1.PASS2.OBJECT(SPLLIST),DISP=SHR',
// PARM.COMPILE=('LIST(USRID1.LISTPDS) IPA(LINK,MAP) OPT',
//   'RENT LO SPLITLIST')
//OBJECT  DD  DSN=USRID1.PASS1.OBJECT,DISP=SHR
//SYSIN    DD *,DLM='/>'
  INCLUDE OBJECT(SPLLIST)
/>
//
```

Related information

"LIST | NOLIST" on page 83

SSCOMM | NOSSCOMM

Category

Language element control

Pragma equivalent

None.

Purpose

Allows comments to be specified by two slashes (//), which supports C++ style comments in C code.

When the SSCOMM option is in effect, it instructs the C compiler to recognize two slashes (//) as the beginning of a comment, which terminates at the end of the line. It will continue to recognize /*   */ as comments.

When the NOSSCOMM compiler option is in effect, /* */ is the only valid comment format.

Syntax

Defaults

NOSSCOMM

For LANGLVL(STDC99) and LANGLVL(EXTC99), the default is SSCOMM.

Usage

When using the **metalc** command in z/OS UNIX System Services, the equivalent option for SSCOMM is **-qcpluscmt**.

Predefined macros

None.

Examples

If you include your C program in your JCL stream, be sure to change the delimiters so that your comments are recognized as Enterprise Metal C for z/OS comments and not as JCL statements:

```
//COMPILE.SYSIN DD DATA,DLM=@@
void main(){
// Enterprise Metal C for z/OS comment
   char *string = "hello world";
// A nested Enterprise Metal C for z/OS /*  */ comment
}
@@
//* JCL comment
```

STRICT | NOSTRICT

Category

Optimization and tuning

Pragma equivalent

#pragma option_override(subprogram_name, "OPT(STRICT)")

Purpose

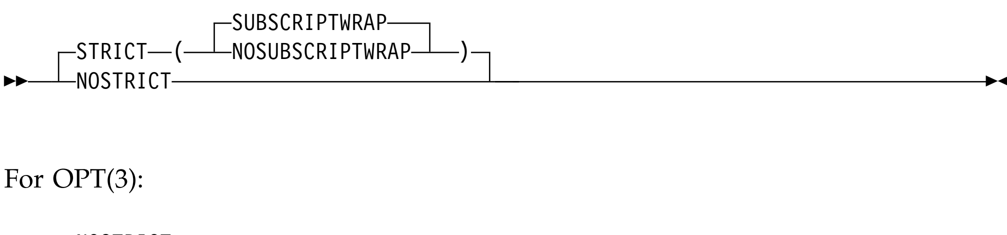
Used to prevent optimizations done by default at optimization levels OPT(3), and, optionally at OPT(2), from re-ordering instructions that could introduce rounding errors.

When the STRICT option is in effect, the compiler performs computational operations in a rigidly-defined order such that the results are always determinable and recreatable.

When the NOSTRICT compiler option is in effect, the compiler can reorder certain computations for better performance. However, the end result may differ from the result obtained when STRICT is specified.

Syntax

For NOOPT and OPT(2):



For OPT(3):

```
      ┌─NOSTRICT──────────────────────────┐
      │          ┌─NOSUBSCRIPTWRAP─┐      │
►►────┴─STRICT──(──┴─SUBSCRIPTWRAP───┴──)─┴──────────────────────────────►◄
```

Defaults

For NOOPT and OPT(2), the default option is STRICT. For OPT(3), the default option is NOSTRICT.

Usage

STRICT disables the following optimizations:

- Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception.
- Relaxing conformance to IEEE rules.
- Reassociating floating-point expressions.

In IEEE floating-point mode, NOSTRICT sets FLOAT(MAF). To avoid this behavior, explicitly specify FLOAT(NOMAF).

STRICT(SUBSCRIPTWRAP) prevents the compiler from assuming that array subscript expressions will never overflow.

When the NOSTRICT or STRICT(NOSUBSCRIPTWRAP) option is in effect, the compiler is free to perform operations which might be unsafe when there are integer overflow operations involving array subscript expressions.

The [NO]STRICT_INDUCTION setting supersedes STRICT([NO]SUBSCRIPTWRAP) or NOSTRICT, when induction variables are present in the array subscript expressions.

When STRICT settings in source level pragmas conflict with compilation unit STRICT settings, the settings in the source level pragmas are applied.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The STRICT settings for each compilation unit and procedure are preserved from the IPA compile step and respected during the IPA link step. You cannot override the setting of STRICT by specifying the option on the IPA link step. If you specify the STRICT option on the IPA link step, the compiler issues a warning message and ignores the STRICT option. For more information about the IPA link processing of the STRICT option, see "FLOAT" on page 58.

Predefined macros

None.

STRICT_INDUCTION | NOSTRICT_INDUCTION

Category

Optimization and tuning

Pragma equivalent

None.

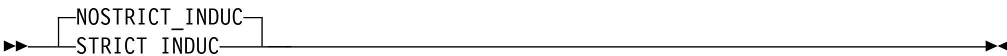
Purpose

Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.

When the STRICT_INDUCTION option is in effect, the compiler disables loop induction variable optimizations.

When the NOSTRICT_INDUCTION compiler option is in effect, the compiler permits loop induction variable optimizations.

Syntax



Defaults

NOSTRICT_INDUCTION

Usage

Loop induction variable optimizations can change the result of a program if truncation or sign extension of a loop induction variable occurs as a result of variable overflow or wrap-around.

The STRICT_INDUCTION option only affects loops which have an induction (loop counter) variable declared as a different size than a register. Unless you intend such variables to overflow or wrap-around, use NOSTRICT_INDUCTION.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step merges and optimizes your application's code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA link step uses information from the IPA compile step to ensure that an object is included in a compatible partition.

The compiler sets the value of the STRICT_INDUCTION option for a partition to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different STRICT_INDUCTION settings may be combined in the same partition. When this occurs, the resulting partition is always set to STRICT_INDUCTION.

You can override the setting of STRICT_INDUCTION by specifying the option on the IPA link step. If you do so, all partitions will contain that value, and the prolog section of the IPA link step listing will display the value.

Predefined macros

None.

SUPPRESS | NOSUPPRESS

Category

Listings, messages and compiler information

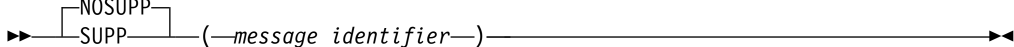
Pragma equivalent

None.

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax



Defaults

For C, the default is NOSUPPRESS.

Parameters

message_identifier
Comma separated list of message IDs.

Usage

The message ID range that is affected is CJT3000 through CJT4399.

Note that this option has no effect on linker or operating system messages. Compiler messages that cause compilation to stop, such as (S) and (U) level messages cannot be suppressed.

If a compilation has no (S) and (U) level messages and all the informational or warning messages are suppressed by the SUPPRESS option, the compilation return code is 0.

When you specify NOSUPPRESS with specific message identifiers, the previous SUPPRESS instances with the same message identifiers lose effect. When you specify NOSUPPRESS without specific message identifiers, all previous SUPPRESS instances lose effect. If you specify two or three of the following options, the last option has precedence:

SUPPRESS(*message_identifier*)
NOSUPPRESS(*message_identifier*)
NOSUPPRESS

IPA effects

The SUPPRESS option has the same effect on the IPA link step that it does on a regular compilation.

Predefined macros

None.

SYSSTATE

Category

Object code control

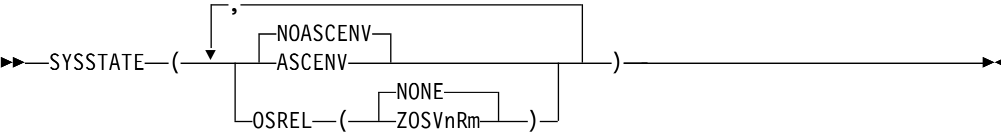
Pragma equivalent

None.

Purpose

Provides additional SYSSTATE macro parameters to the SYSSTATE macro that is generated by the compiler.

Syntax



Defaults

SYSSTATE(NOASCENV, OSREL(NONE))

Parameters

ASCENV | NOASCENV

Instructs the compiler to automatically generate additional SYSSTATE macros with the ASCENV parameter to reflect the ASC mode of the function.

The default is NOASCENV, with which no ASCENV parameter appears on the SYSSTATE macro.

OSREL (NONE | ZOSVnRm)

Provides z/OS release value for the OSREL parameter on the SYSSTATE macro.

The z/OS release value must be in the form of ZOSVnRm as described in *z/OS MVS Programming: Assembler Services Reference*. Valid values for the OSREL parameter include ZOSV1R6 or later z/OS releases.

The default is NONE, with which no OSREL parameter appears on the SYSSTATE macro.

Usage

You can specify the SYSSTATE compiler option to enhance the SYSSTATE macro that is generated by the compiler. With the SYSSTATE option, you can include the OSREL parameter in the SYSSTATE macro, or have the ASCENV parameter automatically set, or both.

The effect of the SYSSTATE macro depends on whether you use other system macros and whether those system macros rely on system variables that are set by the SYSSTATE macro. For example, if a system macro checks for the OSREL setting, you might need to include the OSREL parameter; if a system macro used in an AR mode function checks for the ASCENV setting, you might need to add the ASCENV parameter. With the SYSSTATE compiler option, you can control how these parameters can be added to the SYSSTATE macro.

IPA effects

If you specify different SYSSTATE suboptions for compilation units during the IPA compile step, different SYSSTATE values will be isolated in different partitions during the IPA link step.

If the SYSSTATE option is specified during the IPA link step, it overrides all other SYSSTATE settings during the IPA compile step.

Predefined macros

None.

TERMINAL | NOTERMINAL

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Directs diagnostic messages to be displayed on the terminal.

Syntax

Defaults

TERMINAL

Usage

When the TERMINAL compiler option is in effect, it directs all of the diagnostic messages of the compiler to `stderr`.

Under z/OS batch, the default for `stderr` is SYSPRINT.

If you specify the PPONLY option, the compiler turns on TERM.

IPA effects

The TERMINAL compiler option has the same effect on the IPA link step as it does on a regular compile step.

Predefined macros

None.

Related information

For more information on the PPONLY compiler option, see "PPONLY | NOPPONLY" on page 112.

TUNE

Category

Optimization and tuning

Pragma equivalent

`#pragma options(tune)`

Purpose

Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture.

Syntax

```
►►—TUN—(—n—)————————————————————————————————————————►◄
```

Defaults

TUNE(10)

Parameters

n Specifies the group to which a model number belongs as a sub-parameter. If you specify a model which does not exist or is not supported, a warning message is issued stating that the suboption is invalid and that the default will be used. Current models that are supported include:

0 This option generates code that is executable on all models, but it will not be able to take advantage of architectural differences on the models specified in the following information.

1 This option generates code that is executable on all models but that is optimized for the following models:

- 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900
- 9021-xx1, 9021-xx2, and 9672-Rx2 (G1)

2 This option generates code that is executable on all models but that is optimized for the following models:

- 9672-Rx3 (G2), 9672-Rx4 (G3), and 2003
- 9672-Rx1, 9672-Exx, and 9672-Pxx

3 This option generates code that is executable on all models but that is optimized for the following and follow-on models: 9672-Rx5 (G4), 9672-xx6 (G5), and 9672-xx7 (G6).

4 This option generates code that is executable on all models but that is optimized for the model 2064-100 (z900).

5 This option generates code that is executable on all models but that is optimized for the model 2064-100 (z900) in z/Architecture mode.

6 This option generates code that is executable on all models, but is optimized for the 2084-xxx (z990) models.

7 This option generates code that is executable on all models, but is optimized for the 2094-xxx (IBM System z9 Enterprise Class) and 2096-xxx (IBM System z9 Business Class) models.

8 This option is the default. This option generates code that is executable on all models, but is optimized for the 2097-xxx (IBM System z10 Enterprise Class) and 2098-xxx (IBM System z10 Business Class) models.

9 This option generates code that is executable on all models, but is optimized for the 2817-xxx (IBM zEnterprise 196 (z196)) and 2818-xxx (IBM zEnterprise 114 (z114)) models.

10 This option generates code that is executable on all models, but is optimized for the 2827-xxx (IBM zEnterprise EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models.

11 This option generates code that is executable on all models, but is optimized for the 2964-xxx (IBM z13™ (z13)) and the 2965-xxx (IBM z13s (z13s)) models.

12 This option generates code that is executable on all models, but is optimized for the 3906-xxx (IBM z14) and 3907-xxx (IBM z14 ZR1) models.

Note: For these system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RY4 and 9672-R14 through to 9672-R94 (the entire range of G3 processors), not just 9672-RX4.

Usage

The TUNE option specifies the architecture for which the executable program will be optimized. The TUNE level controls how the compiler selects and orders the available machine instructions, while staying within the restrictions of the ARCH level in effect. It does so in order to provide the highest performance possible on the given TUNE architecture from those that are allowed in the generated code. It also controls instruction scheduling (the order in which instructions are generated to perform a particular operation). Note that TUNE impacts performance only; it does not impact the processor model on which you will be able to run your application.

Select TUNE to match the architecture of the machine where your application will run most often. Use TUNE in cooperation with ARCH. TUNE must always be greater or equal to ARCH because you will want to tune an application for a machine on which it can run. The compiler enforces this by adjusting TUNE up rather than ARCH down. TUNE does not specify where an application can run. It is primarily an optimization option. For many models, the best TUNE level is not the best ARCH level. For example, the correct choices for model 9672-Rx5 (G4) are ARCH(2) and TUNE(3). For more information on the interaction between TUNE and ARCH see "ARCHITECTURE" on page 27.

Note: If the TUNE level is lower than the specified ARCH level, the compiler forces TUNE to match the ARCH level or uses the default TUNE level, whichever is greater.

Information on the level of the TUNE option will be generated in your object module to aid you in diagnosing your program.

IPA effects

The IPA compile step generates information for the IPA link step.

The IPA link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition.

If you specify the TUNE option for the IPA link step, it uses the value of the option you specify. The value you specify appears in the IPA link step Prolog listing section and all Partition Map listing sections.

If you do not specify the option on the IPA link step, the value it uses for a partition depends upon the TUNE option you specified during the IPA compile step for any compilation unit that provided code for that partition. If you specified

the same TUNE value for all compilation units, the IPA link step uses that value. If you specified different TUNE values, the IPA link step uses the highest value of TUNE.

If the resulting level of TUNE is lower than the level of ARCH, TUNE is set to the level of ARCH.

The Partition Map section of the IPA link step listing, and the object module display the final option value for each partition. If you override this option on the IPA link step, the Prolog section of the IPA link step listing displays the value of the option.

The Compiler Options Map section of the IPA link step listing displays the value of the TUNE option that you specified on the IPA compile step for each object file.

Predefined macros

__TUNE__ is predefined to the value specified by the TUNE compiler option.

UNDEFINE

Category

Language element control

Pragma equivalent

None.

Purpose

Undefines preprocessor macro names.

Syntax

Defaults

Not applicable.

Parameters

name
Specifies a preprocessor macro name.

Usage

UNDEFINE(*name*) removes any value that *name* may have and makes its value undefined. For example, if you set `OS2` to 1 with `DEF(OS2=1)`, you can use the UNDEF(OS2) option to remove that value. **metalc** passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 14, “metalc — Compiler invocation using a customizable configuration file,” on page 219.

Predefined macros

None.

UNROLL | NOUNROLL

Category

Optimization and tuning

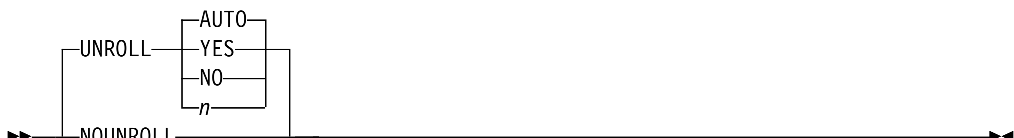
Pragma equivalent

#pragma unroll

Purpose

Controls loop unrolling, for improved performance.

Syntax



Defaults

UNROLL(AUTO)

Parameters

YES
Allows the compiler to unroll loops that are annotated (for example, using a pragma), unless it is overridden by **#pragma nounroll**.

NO Means that the compiler is not permitted to unroll loops in the compilation unit, unless **unroll** or **unroll(n)** pragmas are specified for particular loops.

AUTO
This option is the default. It enables the compiler to unroll loops that are annotated (for example, using a pragma) and loops which the compiler has decided (via heuristics) are appropriate for unrolling. AUTO should only be specified if you have specified OPTIMIZE(2) or greater and COMPACT is not specified.

n

Instructs the compiler to unroll loops by a factor of *n*. In other words, the body of a loop is replicated to create *n* copies, and the number of iterations is reduced by a factor of *1/n*. The UNROLL(n) option specifies a global unroll factor that affects all loops that do not have an unroll pragma already. The value of *n* must be a positive integer.

Specifying **#pragma unroll(1)** or UNROLL(1) option disables loop unrolling, and is equivalent to specifying **#pragma nounroll** or UNROLL option.

Usage

The UNROLL compiler option instructs the compiler to perform loop unrolling, which is an optimization that replicates a loop body multiple times, and adjusts the loop control code accordingly. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve a program's performance. It also increases code size in the new loop body, which may increase pressure on register allocation, cause register spilling, and therefore cause a loss in performance. Before applying unrolling to a loop, you must evaluate these tradeoffs. In order to check if the unroll option improves performance of a particular application, you should compile your program with the usual options, run it with a representative workload, recompile it with the UNROLL option and/or unroll pragmas, and rerun it under the same conditions to see if the UNROLL option leads to a performance improvement.

Specifying UNROLL without any suboptions is equivalent to specifying UNROLL(YES).

Specifying NOUNROLL is equivalent to specifying UNROLL(NO).

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

UPCONV | NOUPCONV

Category

Portability and migration

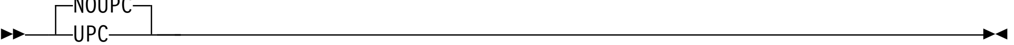
Pragma equivalent

#pragma options(upconv), **#pragma options(noupconv)**

Purpose

Specifies whether the `unsigned` specification is preserved when integral promotions are performed.

Syntax



Defaults

NOUPCONV

Usage

The UPCONV option causes the compiler to follow unsignedness preserving rules when doing C type conversions; that is, when widening all integral types (`char`, `short`, `int`, `long`). Use this option when compiling older C programs that depend on the *K&R C* conversion rules.

Note: This document uses the term *K&R C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie that were in use prior to the ISO standardization of C.

Whenever the UPCONV compiler option is in effect, the usage status of the UPCONV option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

VECTOR | NOVECTOR

Category

Language element control

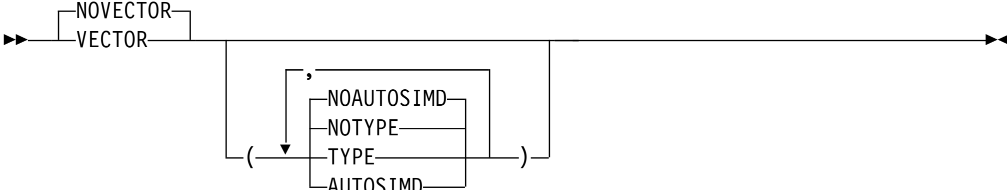
Pragma equivalent

None.

Purpose

For a runtime environment that supports vector instructions, this option can be specified to control whether the compiler enables the vector programming support and automatically takes advantage of vector instructions.

Syntax



Defaults

NOVECTOR(NOTYPE, NOAUTOSIMD)

When running on z/OS V2R3 system, the default is as follows, if neither LANGLVL(STRICT98) nor LANGLVL(ANSI) is in effect:

- VECTOR(NOTYPE, AUTOSIMD) when all of the following options are in effect: ARCH(11) or higher levels, FLOAT(AFP(NOVOLATILE)), and HOT.
- VECTOR(NOTYPE, NOAUTOSIMD) when all of the following options are in effect: ARCH(12), FLOAT(AFP(NOVOLATILE)), and OPT(3).

Note:
- Specifying VECTOR without suboptions is equivalent to VECTOR(TYPE).

Parameters

TYPE | NOTYPE
Enables the support for `vector` data types, in addition to `__vector` data types. The default is NOTYPE.

AUTOSIMD | NOAUTOSIMD
Enables the automatic SIMDization or automatic vectorization optimization that uses Single Instruction Multiple Data (SIMD) instructions where possible, which calculate several results at one time and is faster than calculating each result sequentially. This optimization is available only when HOT is in effect. The default is NOAUTOSIMD.

Usage

The VECTOR option is effective only when ARCH(11) or higher levels and FLOAT(AFP(NOVOLATILE)) are in effect.

IBM z13 (z13) and IBM z13s (z13s) hardware introduced the support for vector instructions under the Vector Facility for z/Architecture. The newest generation of the hardware with the vector enhancements facility 1 and vector packed decimal facility further enhances the support for vector instructions.

The VECTOR option enables the `__vector` data types for vector programming support.

The VECTOR option provides potential performance improvements in the following aspects: fixed point decimal operations, built-in library functions, operations on binary floating-point `double`, `float`, and `long double` data types, and SIMD instructions.

The vector or SIMD code must run in the following runtime environments that support vector instructions and vector context switching:
- z/OS V2.1 with PTF for APAR PI12281 or later.
- z/OS image running on z/VM® V6.3 with PTF for APAR VM65733 or later.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

IPA effects

If you specify the AUTOSIMD suboption on the IPA link step, it uses this suboption for all partitions. The IPA link step Prolog and all Partition Map sections of the IPA link step listing display this suboption.

If you do not specify the AUTOSIMD suboption on the IPA link step, the value used for a partition depends on the value that you specified for the IPA compile step for each compilation unit that provided code for that partition.

If you specify the NOVECTOR option, or the TYPE, NOTYPE, or NOAUTOSIMD suboption on the IPA link step, the compiler ignores them.

Predefined macros

__VEC__ is defined to 10402 when VECTOR is in effect.

Related information

- "ARCHITECTURE" on page 27
- "FLOAT" on page 58
- "LANGLVL" on page 79

WARN64 | NOWARN64

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates diagnostic messages, which enable checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Syntax

Defaults

NOWARN64

Usage

Use the FLAG(I) option to display any informational messages.

WARN64 warns you about any code fragments that have the following types of portability errors:

- A constant that selected an `unsigned long int` data type in 31-bit mode may fit within a `long int` data type in 64-bit mode
- A constant larger than UINT_MAX, but smaller than ULONGLONG_MAX will overflow in 31-bit mode, but will be acceptable in an `unsigned long` or `signed long` in 64-bit mode

It also warns you about the following types of possible portability errors:

- Loss of digits when you assign a `long` type to an `int` type
- Change in the result when you assign an `int` to a `long` type
- Loss of high-order bytes of a pointer when a pointer type is assigned to an `int` type
- Incorrect pointer when an `int` type is assigned to a pointer type
- Change of a constant value when the constant is assigned to a `long` type

Predefined macros

None.

WSIZEOF | NOWSIZEOF

Category

Object code control

Pragma equivalent

#pragma wsizeof(on)

Purpose

Causes the `sizeof` operator to return the widened size for function return types.

When the WSIZEOF compiler option is in effect, `sizeof` returns the size of the widened type for function return types instead of the size of the original return type.

When the NOWSIZEOF compiler option is in effect, `sizeof` returns the size of the original return type.

Syntax

```
      ┌─NOWSIZEOF─┐
►►────┴─WSIZEOF───┴──────────────────────────────────────────────────►◄
```

Defaults

NOWSIZEOF

Usage

When the `sizeof` operator was applied to a function return type using the WSIZEOF compiler option the compiler returns the size of the widened type instead of the original type. For example, if the following code fragment, was compiled with an earlier compiler, `i` would have a value of `4`.

```
char myfunc();
i = sizeof myfunc();
```

Using the Enterprise Metal C for z/OS compiler, `i` has a value of `1`, which is the size of the original type `char`.

The usage status of this option is inserted in the object file to aid you in diagnosing a problem with your program.

Predefined macros

None.

Using compiler listing

If you select the SOURCE or LIST option, the compiler creates a listing that contains information about the source program and the compilation. If the compilation terminates before reaching a particular stage of processing, the compiler does not generate corresponding parts of the listing. The listing contains standard information that always appears, together with optional information that is supplied by default or specified through compiler options.

In an interactive environment you can also use the TERMINAL option to direct all compiler diagnostic messages to your terminal. The TERMINAL option directs only the diagnostic messages part of the compiler listing to your terminal.

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

IPA considerations

The listings that the IPA compile step produces are basically the same as those that a regular compilation produces. Any differences are noted throughout this section.

The IPA link step listing has a separate format from the other compiler listings. Many listing sections are similar to those that are produced by a regular compilation. Refer to "Using the IPA link step listing" on page 156 for information about IPA link step listings.

Compiler listing components

The following information describes the components of a compiler listing. These are available for regular and IPA compilations. Differences in the IPA versions of the listings are noted. "Using the IPA link step listing" on page 156 describes IPA-specific listings.

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the name of the data set or z/OS UNIX file containing the source code, the date and time compilation began (formatted according to the current locale), and the page number.

Note: If the name of the data set or z/OS UNIX file that contains the source code is greater than 32 characters, it is truncated. Only the right-most 30 characters appear in the listing.

Prolog section

The Prolog section provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the compiler was invoked.

All options except those with no default are shown in the listing. Any problems with the compiler options appear after the body of the Prolog section.

IPA considerations: If you specify IPA suboptions that are irrelevant to the IPA compile step, the Prolog does not display them. If IPA processing is not active, IPA suboptions do not appear in the Prolog. The following information describes the optional parts of the listing and the compiler options that generate them.

Source program

If you specify the SOURCE option, the listing file includes input to the compiler.

Note: If you specify the SHOWINC option, the source listing shows the included text after the `#include` directives.

Includes section

The compiler generates the Includes section when you use include files, and specify the option SOURCE or LIST.

Cross-Reference Listing

The XREF option generates a cross-reference table that contains a list of the identifiers from the source program and the line numbers in which they appear.

Structure and Union Maps

You obtain structure and union maps by using the AGGREGATE option. The table shows how each structure and union in the program is mapped. It contains the following:

- Name of the structure or union and the elements within the structure or union
- Byte offset of each element from the beginning of the structure or union, and the bit offset for unaligned bit data
- Length of each element
- Total length of each structure, union, and substructure

Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, it generates messages. If you specify the SOURCE compiler option, preprocessor error messages appear immediately after the source statement in error. You can generate your own messages in the preprocessing stage by using the **`#error`** preprocessor directive.

If you specify the compiler option INFO, the compiler will generate informational diagnostic messages.

For more information on the compiler messages, see “FLAG | NOFLAG” on page 57, and *Enterprise Metal C for z/OS Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

Using the IPA link step listing

The IPA link step generates a listing file if you specify any of the following options:

- IPA(MAP)
- LIST

IPA link step listing components

The following information describes the components of an IPA link step listing.

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the central title area, the date and time compilation began (formatted according to the current locale), and the page number.

In the following listing sections, the central title area will contain the primary input file identifier:

- Prolog
- Object File Map
- Source File Map
- Compiler Options Map
- Global Symbols Map
- Messages
- Message Summary

In the following listing sections, the central title area will contain the phrase Partition `nnnn`, where `nnnn` specifies the partition number:

- Partition Map

Prolog section

The Prolog section of the listing provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the IPA link step was invoked.

The listing displays all compiler options except those with no default. If you specify IPA suboptions that are irrelevant to the IPA link step, the Prolog does not display them. Any problems with compiler options appear after the body of the Prolog section and before the End of Prolog section.

Object File Map

The Object File Map displays the names of the object files that were used as input to the IPA link step. Specify IPA(MAP) or LIST to generate the Object File Map.

Other listing sections, such as the Source File Map, use the File ID numbers that appear in this listing section.

z/OS UNIX file names that are too long to fit into a single listing record continue on subsequent listing records.

Source File Map

The Source File Map listing section identifies the source files that are included in the object files. The IPA link step generates this section if you specify the IPA(MAP) option.

The IPA link step formats the compilation date and time according to the locale you specify with the LOCALE option in the IPA link step. If you do not specify the LOCALE option, it uses the default locale.

This section appears near the end of the IPA link step listing. If the IPA link step terminates early due to errors, it does not generate this section.

Compiler Options Map

The Compiler Options Map listing section identifies the compiler options that were specified during the IPA compile step for each compilation unit that is encountered when the object file is processed. For each compilation unit, it displays the final options that are relevant to IPA link step processing. You may have specified these options through a compiler option or `#pragma` directive, or you may have picked them up as defaults.

The IPA link step generates this listing section if you specify the IPA(MAP) option.

Global Symbols Map

The Global Symbols Map listing section shows how global symbols are mapped into members of global data structures by the global variable coalescing optimization process.

Each global data structure is limited to 16 MB by the z/OS object architecture. If an application has more than 16 MB of data, IPA Link must generate multiple global data structures for the application. Each global data structure is assigned a unique name.

The Global Symbols Map includes symbol information and file name information (file name information may be approximate).

The IPA link step generates this listing section if you specify the IPA(MAP) option and the IPA link step causes global symbols to be coalesced. The Global Symbols Map is only added to the IPA link step listing if the IPA Link phase optimization changes the structure and/or layout of the global symbols used by the final module. If no changes are made, then the Global Symbols Map is not included in the listing.

Partition Map

The Partition Map listing section describes each of the object code partitions the IPA link step creates. It provides the following information:
- The reason for generating each partition
- How the code is packaged (the CSECTs)
- The options used to generate the object code
- The function and global data included in the partition
- The source files that were used to create the partition

The IPA link step generates this listing section if you specify the IPA(MAP) option.

Messages

If the IPA link step detects an error, or the possibility of an error, it issues one or more diagnostic messages, and generates the Messages listing section. This listing section contains a summary of the messages that are issued during IPA link step processing.

The IPA link step listing sorts the messages by severity. The Messages listing section displays the listing page number where each message was originally shown. It also displays the message text, and optionally, information relating the error to a file name, line (if known), and column (if known).

For more information on compiler messages, see “FLAG | NOFLAG” on page 57 and *Enterprise Metal C for z/OS Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

The following table shows the components that are included in the listing depending on which option is specified during the IPA link phase:

Table 24. IPA link step listing components

Listing Component	`-Wc, IPA (MAP)`	`-Wc, LIST (destination)`	`-V`
Compiler Options Map	✓		✓
Global Symbols Map **	✓		✓
Message Summary	✓	✓	✓
Messages *	✓	✓	✓
Object File Map	✓		✓
Partition Map	✓		✓
Prolog	✓	✓	✓
Source File Map	✓		✓

* This section is only generated if diagnostic messages are issued.

** This section is only generated if the IPA Link phase coalesces global variables.

Chapter 3. Compiling

This information describes how to compile your program with the Enterprise Metal C for z/OS compiler. For specific information about compiler options, see Chapter 2, “Compiler options,” on page 7.

The Enterprise Metal C for z/OS compiler analyzes the source program and translates the source code into machine instructions that are known as *object code*.

You can perform compilations under z/OS batch, TSO, or the z/OS UNIX System Services environment.

Input to the compiler

The following information describes how to specify input to the Enterprise Metal C for z/OS compiler for a regular compilation, or the IPA compile step. For more information about input for IPA, refer to Chapter 4, “Using IPA link step with programs,” on page 185.

If you are compiling a C program, input for the compiler consists of the following:

- Your Enterprise Metal C for z/OS source program
- The Enterprise Metal C for z/OS supported standard header files
- Your header files

When you invoke the Enterprise Metal C for z/OS compiler, the operating system locates and runs the compiler. To run the compiler, you need the default data set CJT.SCJTCMP, which is supplied by IBM. The locations of the compiler and the runtime library were determined by the system programmer who installed the product. The compiler and library should be in the STEPLIB, JOBLIB, LPA, or LNKLST concatenations. LPA can be from either specific modules (`IEALPAxx`) or a list (`LPALSTxx`). See the cataloged procedures shipped with the product in Chapter 9, “Cataloged procedures,” on page 199.

Note: For z/OS UNIX System Services file names, unless they appear in JCL, file names, which contain the special characters blank, backslash, and double quotation mark, must escape these characters. The escape character is backslash (\).

Primary input

For a C program, the primary input to the compiler is the data set that contains your source program. If you are running the compiler in batch, identify the input source program with the SYSIN DD statement. You can do this by either defining the data set that contains the source code or by placing your source code directly in the JCL stream. In TSO or in z/OS UNIX System Services, identify the input source program by name as a command line argument. The primary input source file can be any one of the following:

- A sequential data set
- A member of a partitioned data set
- All members of a partitioned data set
- A z/OS UNIX file
- All files in a z/OS UNIX directory

© Copyright IBM Corp. 2018

Secondary input

For a C program, secondary input to the compiler consists of data sets or directories that contain include files. Use the LSEARCH and SEARCH compiler options, or the SYSLIB DD statement when compiling in batch, to specify the location of the include files.

Related information

- "LSEARCH | NOLSEARCH" on page 91
- "SEARCH | NOSEARCH" on page 126
- "Specifying include file names" on page 173
- "Search sequences for include files" on page 182
- "Using include files" on page 173

Output from the compiler

You can specify compiler output files as one or more of the following:
- A sequential data set
- A member of a partitioned data set
- A partitioned data set
- A z/OS UNIX file
- A z/OS UNIX directory

For valid combinations of input file types and output file types, refer to Table 27 on page 164.

Specifying output files

You can use compile options to specify compilation output files as follows:

Table 25. Compile options that provide output file names

Output File Type	Compiler Option
Listing File	SOURCE (*filename*), LIST(filename) **Note:** All listings must go to the same file. The last given location is used.
Preprocessor Output	PPONLY(*filename*)
Events File	EVENTS(*filename*)

When compiler options that generate output files are specified without suboptions to identify the output files, and, in the case of a batch job, the designated ddnames are not allocated, the output file names are generated based on the name of the source file. For data sets, the compiler generates a low-level qualifier by appending a suffix to the data set name of the source, as Table 26 on page 163 shows.

If you compile source from z/OS UNIX files without specifying output file names in the compiler options, the compiler writes the output files to the current working directory. The compiler does the following to generate the output file names:
- Appends a suffix, if it does not exist
- Replaces the suffix, if it exists

The following default suffixes are used:

Table 26. Defaults for output file types

Output File Type	z/OS File	z/OS UNIX File
Asembler source file	ASM	s
Listing file	LIST	lst
Preprocessor Output	EXPAND	i

Notes:

1. Output files default to the z/OS UNIX directory if the source resides in the z/OS UNIX file system, or to an MVS data set if the source resides in a data set.
2. If you have specified the OE option, see "OE | NOOE" on page 104 for a description of the default naming convention.
3. If you supply inline source in your JCL, the compiler will not generate an output file name automatically. You can specify a file name on a ddname in your JCL.
4. If you are using **#pragma options** to specify a compile-time option that generates an output file, you must use a ddname to specify the output file name when compiling under batch. The compiler will not automatically generate file names for output that is created by **#pragma options**.

Listing output

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

To create a listing file that contains source , use the SOURCE or LIST compile option. The listing includes the results of the default or specified options of the CPARM parameter (that is, the diagnostic messages and the object code listing). If you specify *filename* with two or more of these compile options, the compiler combines the listings and writes them to the last file specified in the compile options. If you did not specify *filename*, the listing will go to the SYSCPRT DD name, if you allocated it. Otherwise, the compiler generates a default file name as described in "LIST | NOLIST" on page 83.

Preprocessor output

If you specify *filename* with the PPONLY compile option, the compiler writes the preprocessor output to that file. If you do not specify *filename* with the PPONLY option, the compiler stores the preprocessor output in the file that you define in the SYSUT10 DD statement. If you did not allocate SYSUT10, the compiler generates a default file name, as described in "PPONLY | NOPPONLY" on page 112.

Valid input/output file types

Depending on the type of file that is used as primary input, certain output file types are allowed. The following table describes these combinations of input and output files:

Table 27. Valid combinations of source and output file types

Input Source File	Output Data Set Specified Without (member) Name, for example `A.B.C`	Output Data Set Specified as filename(member), for example `A.B.C(D)`	Output Specified as a z/OS UNIX file, for example `a/b/c.o`	Output Specified as a z/OS UNIX directory, for example `a/b`
Sequential Data Set, for example `A.B`	1. If the file exists as a sequential data set, overwrites it 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file	Not supported
A member of a PDS using (member), for example `A.B(C)`	1. If the file exists as a sequential data set, overwrites it 2. If the file exists as a PDS, creates or overwrites member 3. If the file does not exist, creates PDS and member	1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the specified file name does not exist, creates file 3. If the directory exists and the file exists, overwrites file	1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the file name *MEMBER.ext* does not exist, creates file 3. If the directory exists and the file with the file name *MEMBER.ext* also exists, overwrite file
All members of a PDS, for example `A.B`	1. If the file exists as a PDS, creates or overwrites members 2. If the file does not exist, creates PDS and members 3. Otherwise compilation fails	Not Supported	Not Supported	1. If the directory does not exist, compilation fails 2. If the directory exists and the files with the file names *`MEMBER.ext`* do not exist, creates files 3. If the directory exists and the files with the file names *`MEMBER.ext`* exist, overwrites files

Table 27. Valid combinations of source and output file types (continued)

Input Source File	Output Data Set Specified Without (member) Name, for example `A.B.C`	Output Data Set Specified as filename(member), for example `A.B.C(D)`	Output Specified as a z/OS UNIX file, for example `a/b/c.o`	Output Specified as a z/OS UNIX directory, for example `a/b`
z/OS UNIX file, for example `/a/b/d.c`	1. If the file exists as a sequential data set, overwrites file 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If the PDS does not exist, creates the PDS and stores a member into the data set 2. If the PDS exists and member does not exist, then adds the member in the PDS 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file	1. If the directory does not exist, compilation fails 2. If the directory exists and the file does not exist, creates file 3. If the directory exists and the file exists, overwrites file
z/OS UNIX directory, for example `a/b/`	Not supported	Not supported	Not supported	1. If the directory does not exist, compilation fails 2. If the directory exists and the files to be written do not exist, creates files 3. If the directory exists and the files to be written already exist, overwrites files

Compiling under z/OS batch

To compile your C source program under batch, you can either use cataloged procedures that IBM supplies, or write your own JCL statements.

Using cataloged procedures

You can use one of the following IBM-supplied cataloged procedures. Each procedure includes a compilation step to compile your program.

Cataloged procedures	Task Description
MTCC	Compile a program
MTCCA	Compile and assemble a program
MTCI	Compile a program with IPA link
MTCIA	Compile a program with IPA link and assemble

IPA considerations

The MTCC procedure should be used for the IPA compile step. Only the MTCI procedure applies to the IPA link step.

To run the IPA compile step, use the MTCC procedure, and ensure that you specify the IPA(NOLINK) or IPA compiler option. Note that you must also specify the LONGNAME compiler option or the **#pragma longname** directive.

To create an IPA-optimized object module, take the following steps:
1. Run the IPA compile step for each source file in your program to generate IPA objects.
2. Run the IPA link step once for the entire program to create the assembly source file.
3. Run the assemble step to create the IPA-optimized object module.
4. Bind the generated IPA-optimized object module to create the final executable.

For further information on IPA, see Chapter 4, “Using IPA link step with programs,” on page 185.

Using special characters

When invoking the compiler directly, if a string contains a single quotation mark (') it should be written as two single quotation marks ('') as in:

```
//COMPILE EXEC PGM=CJTDRVR,PARM='OPTFILE(''USERID.OPTS'')'
```

If you are using the same string to pass a parameter to a cataloged procedure, use four single quotation marks (''''), as follows:

```
//COMPILE EXEC MTCC,CPARM='OPTFILE(''''USERID.OPTS'''')'
```

A backslash need not precede special characters in z/OS UNIX System Services file names that you use in DD cards. For example:

```
//SYSLIN DD PATH='/u/user1/obj 1.o'
```

A backslash must precede special characters in z/OS UNIX file names that you use in the PARM statement. For example:

```
//STEP1 EXEC PGM=CJTDRVR,PARM='OPTFILE(/u/user1/opt\ file)'
```

Specifying source files

For non-z/OS UNIX files, use this format of the SYSIN DD statement:

```
//SYSIN DD DSNAME=dsname,DISP=SHR
```

If you specify a PDS without a member name, all members of that PDS are compiled.

Note: If you specify a PDS as your primary input, you must specify either a PDS or a z/OS UNIX directory for your output files.

For z/OS UNIX files, use this format of the SYSIN DD statement:

```
//SYSIN DD PATH='pathname'
```

You can specify compilation for a single file or all source files in a z/OS UNIX directory, for example:

```
//SYSIN DD PATH='/u/david'
//* All files in the directory /u/david are compiled
```

Note: If you specify a z/OS UNIX directory as your primary input, you must specify a z/OS UNIX directory for your output files.

When you place your source code directly in the input stream, use the following form of the SYSIN DD statement:

```
//SYSIN DD DATA,DLM=
```

rather than:

```
//SYSIN DD *
```

When you use the DD * convention, the first comment statement that starts in column 1 will terminate the input to the compiler. This is because /*, the beginning of a C comment, is also the default delimiter.

Note: To treat columns 73 through 80 as sequence numbers, use the SEQUENCE compiler option.

For more information about the DD * convention, refer to the publications that are listed in *z/OS Information Roadmap*.

Specifying include files

Example: Use the SEARCH option to search data sets `userid.AA.**`.

```
//C EXEC PGM=CJTDRVR,PARM='SEARCH(''AA.+'')'
```

You can also use the SYSLIB and USERLIB DD statements (note that the SYSLIB DD statement has a different use if you are running the IPA link step). To specify more than one library, concatenate multiple DD statements as follows:

```
//SYSLIB     DD     DSNAME=USERLIB,DISP=SHR
//           DD     DSNAME=DUPX,DISP=SHR
```

Note: If the concatenated data sets have different block sizes, either specify the data set with the largest block size first, or use the `DCB=`*dsname* subparameter on the first DD statement. For example:

```
//USERLIB DD DSNAME=TINYLIB,DISP=SHR,DCB=BIGLIB
//        DD DSNAME=BIGLIB,DISP=SHR
```

where `BIGLIB` has the largest block size.

Specifying output files

You can specify output file names as suboptions to the compiler. You can direct the output to a PDS member as follows:

```
//    CPARM='SOURCE(MY.LISTINGS(MEMBER1))'
```

You can direct the output to a z/OS UNIX file as follows:

```
//    CPARM='SOURCE(./listings/member1.lst)'
```

You can also use DD statements to specify output file names.

To specify non-z/OS UNIX files, use DD statements with the DSNAME parameter. For example:

```
//SYSLIN DD DSN=USERID.TEST.OBJ(HELLO),DISP=SHR
```

To specify z/OS UNIX directories or z/OS UNIX files, use DD statements with the PATH parameter.

```
//SYSLIN DD PATH='/u/david/test.o',PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
```

Note: Use the PATH and PATHOPTs parameters when specifying z/OS UNIX files in the DD statements.

If you do not specify the output *filename* as a suboption, and do not allocate the associated ddname, the compiler generates a default output file name. There are two situations when the compiler will not generate a default file name:

- You supply instream source in your JCL.
- You are using **`#pragma options`** to specify a compile-time option that generates an output file.

Compiling in the z/OS UNIX System Services environment

z/OS UNIX C programs with source code in z/OS UNIX files or data sets must be compiled to create output object files residing either in z/OS UNIX files or data sets.

You can compile application source code to generate an HLASM source file using the **`metalc`** command.

You can invoke the compiler by using the **`metalc`** utility, which uses an external configuration file to control the invocation of the compiler.

- The **`metalc`** utility supports **`-q`** options syntax as the primary method of specifying options on the command line.
- The **`metalc`** utility is unaffected by the value assigned to the STEPLIB environment variable in the z/OS UNIX Systems Services session; it obtains the STEPLIB from the configuration file.
- The PATH environment variable must contain the path to the **`metalc`** `bin` directory.
- The **`metalc`** utility uses -O4 and -O5 or **`-qipa`** as the mechanism for invoking IPA.

Note: For more information on the **`metalc`** utility, see Chapter 14, “metalc — Compiler invocation using a customizable configuration file,” on page 219.

For information on customizing your environment to compile in the z/OS UNIX System Services environment, see “Setting up a configuration file” on page 221.

Use the **`metalc`** utility to compile a C application program from the z/OS shell. The syntax is:

```
metalc [-options ...] [file.c ...] [file.a ...] [file.o ...] [-l libname]
```

where:

options
are **`metalc`** options.

file.c
is a source file. Note that C source files have a file extension of lowercase c.

file.o **1**
is an object file.

file.a **1**
is an archive file.

libname **1**
is an archive library.

Note:

1. This file type is supported only when it contains valid input for IPA link and the IPA option is specified, otherwise it will not produce any output.

The **metalc** utility supports IPA. For information on how to invoke the IPA compile step using **metalc**, refer to "Invoking IPA using metalc utility."

Note: You can compile application program source and objects from within the shell using the **metalc** utility. You must keep track of and maintain all the source and object files for the application program. You can use the **make** utility to maintain your z/OS UNIX System Services application source files and object files automatically when you update individual modules. The **make** utility will only compile files that have changed since the last make run.

Related information

Chapter 8, "Building Enterprise Metal C for z/OS programs," on page 197

Building a 64-bit application using metalc utility

To build a 64-bit application using the metalc utility, you must explicitly specify the **-q64** or **-Wc,lp64** compiler option on the command line to ensure 64-bit compiles.

Invoking IPA using metalc utility

You can invoke the IPA compile step, the IPA link step, or both using the **metalc** utility. The step that you invoke depends upon the invocation parameters and type of files specified. You must use the **-qipa**, **-O4**, or **-O5** options. You can specify IPA suboptions as colon-separated keywords.

If you invoke the **metalc** utility by specifying the **-c** compiler option and at least one source file, **metalc** automatically specifies **IPA(NOLINK)** and automatically invokes the IPA compile step.

The following **metalc** command invokes the IPA compile step for the source file `hello.c`:

```
metalc -c -qipa hello.c
```

If you invoke **metalc** with at least one source file for compilation and any number of object files, and do not specify the **-c** option, **metalc** invokes the IPA compile step once for each compilation unit. It then invokes the IPA link step once for the entire program.

Example: The following **metalc** command invokes the IPA compile step and the IPA link step, and produces the assembly output in `myfunc.s`:

```
metalc -o myfunc.s -qipa myfunc.c
```

See Chapter 14, "metalc — Compiler invocation using a customizable configuration file," on page 219 for more information about the **metalc** utility.

Specifying options for the IPA compile step

You can pass options to the IPA compile step, as follows:

- You can pass IPA compiler option suboptions by specifying **-qipa=** for **metalc**, followed by the suboptions.

- You can pass compiler options by specifying **-q** for **metalc**, followed by the options.

Compiling with IPA

If you request Interprocedural Analysis (IPA) through the IPA compiler option, the compilation process changes significantly. IPA instructs the compiler to optimize your program across compilation units, and to perform optimizations that are not otherwise available with the compiler.

Differences between the IPA compilation process and the regular compilation process are noted throughout this topic.

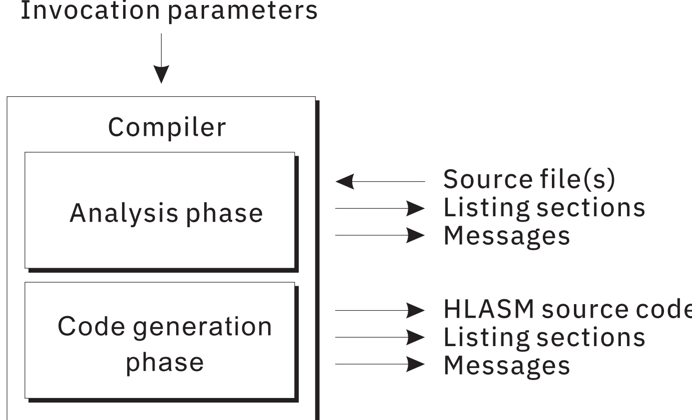


Figure 1. Flow of regular compiler processing

Figure 1 shows the flow of processing for a regular compilation:

IPA processing consists of two separate steps, called the *IPA compile step* and the *IPA link step.*

IPA compile step

The IPA compile step is similar to a regular compilation.

You invoke the IPA compile step for each source file in your application by specifying the IPA(NOLINK) compiler option or by specifying **-Wc,IPA** in z/OS UNIX System Services. The output of the IPA compile step is an object file that contains IPA information. The IPA information is an encoded form of the compilation unit with additional IPA-specific compile-time optimizations.

Figure 2 on page 171 shows the flow of IPA compile step processing.

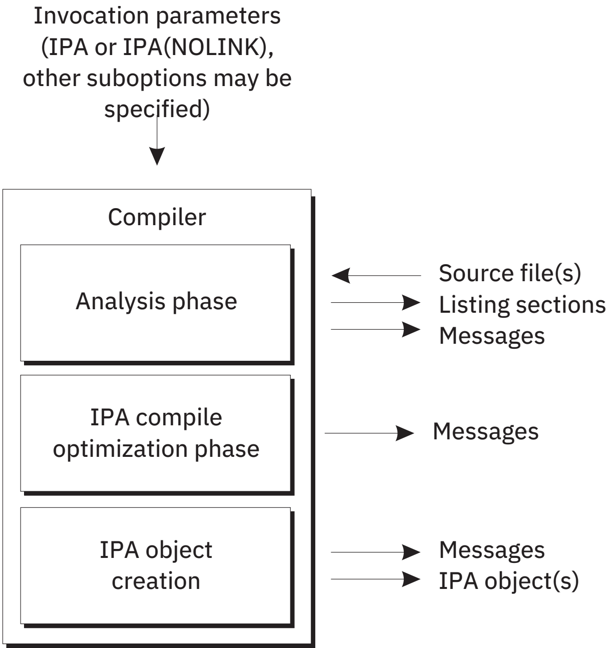


Figure 2. IPA compile step processing

The same environments that support a regular compilation also support the IPA compile step.

IPA link step

The IPA link step is similar to the binding process.

You invoke the IPA link step by specifying the IPA(LINK) compiler option in z/OS UNIX System Services. This step links the user application program together by combining object files with IPA information. It merges IPA information, performs IPA Link-time optimizations, and generates the final assembly code.

Each application program module must be built with a single invocation of the IPA link step. All parts must be available during the IPA link step; missing parts may result in termination of IPA Link processing.

Figure 3 on page 172 shows the flow of IPA link step processing:

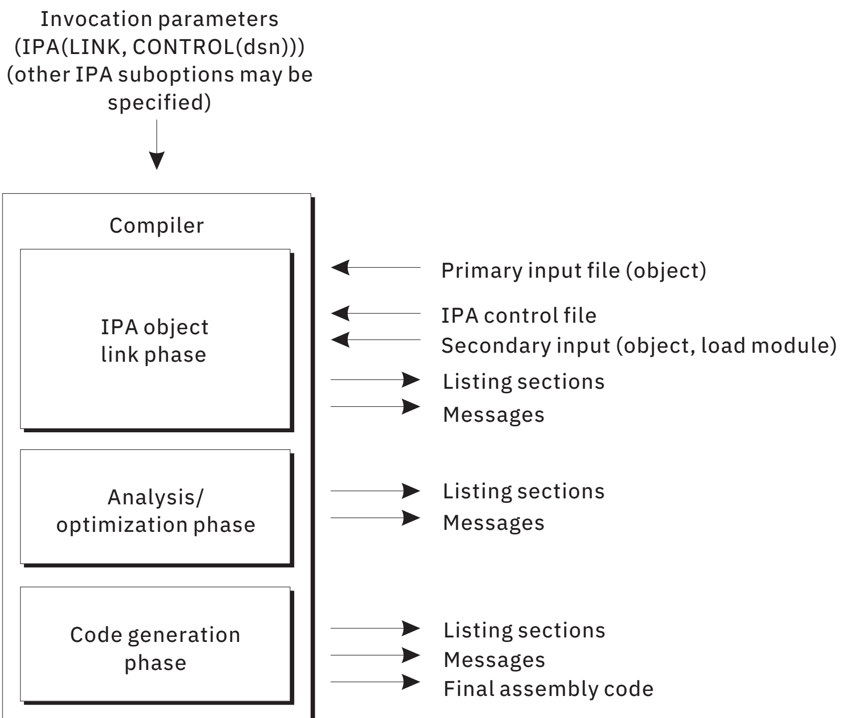


Figure 3. IPA link step processing

Refer to Chapter 4, "Using IPA link step with programs," on page 185 for information about the IPA link step.

Working with object files

z/OS object files are composed of a stream of 80 byte records. These may be binary object records, or link control statements. It is useful to be able to browse the contents of an object file, so that some basic information can be determined.

Browsing object files

Object files, which are sequential data sets or are members of a PDS or PDSE object library, can be browsed directly using the Program Development Facility (PDF) edit and browse options.

Object files, which are z/OS UNIX files, can be browsed using the PDF **obrowse** command. z/OS UNIX files can be browsed using the TSO ISHELL command, and then using the V (View) action (V on the Command line, or equivalently **Browse records** from the File pull-down menu). This will result in a pop-up window for entering a record length. To force display in F 80 record mode, one would issue the following sequence of operations:

1. Enter the command: **obrowse file.o**

   Note: The file name is deliberately typed with an extra character. This will result in the display of an obrowse dialog panel with an error message that the file is not found. After pressing Enter, a second obrowse dialog is displayed to allow the file name to be corrected. This panel has an entry field for the record length.
2. Correct the file name and enter 80 in the record length entry field.
3. Browse the object records as you would a F 80 data set.

The hex display mode (enabled by the HEX ON primary command) allows the value of each byte to be displayed.

Identifying object file variations

Browse the object file and scroll to the end of the file. The last few records contain a character string, which lists the options used during compilation.

In addition, it is possible to identify the compiler mode used to generate the object file. For example, option text "NOIPA" indicates **NOIPA** is effect.

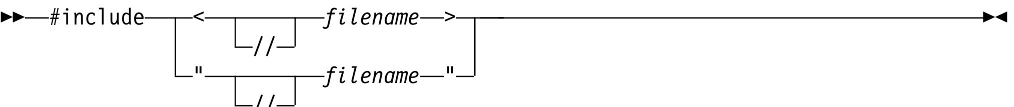
Using feature test macros

The compiler predefines feature test macros when certain features are available. For example, the _LONG_LONG macro is predefined if the compiler supports the `long long` data type.

Using include files

The `#include` preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your C program.

The syntax is:



The angle brackets specify system include files, and double quotation marks specify user include files.

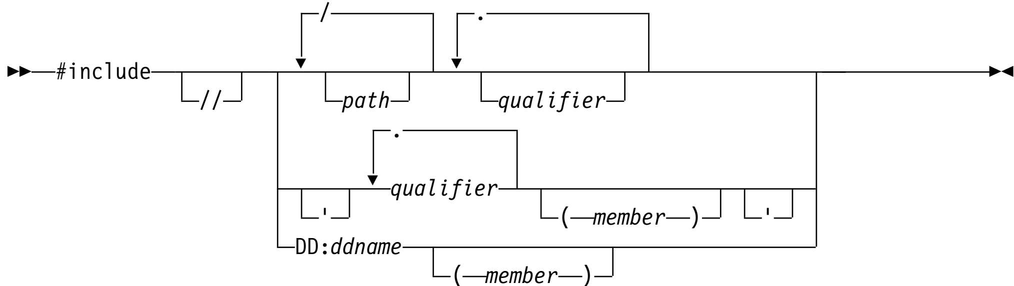
When you use the `#include` directive, you must be aware of the following:

- The *library search sequence*, the search order that the compiler uses to locate the file. See "Search sequences for include files" on page 182 for more information on the library search sequence.
- The file-naming conversions that the compiler performs.
- The area of the input record that contains sequence numbers when you are including files with different record formats. See *Enterprise Metal C for z/OS Language Reference* for more information on #pragma sequence.

Specifying include file names

You can use the SEARCH and LSEARCH compiler options to specify search paths for system include files and user include files. For more information on these options, see "LSEARCH | NOLSEARCH" on page 91 and "SEARCH | NOSEARCH" on page 126.

You can specify *filename* of the `#include` directive in the following format:



The leading double slashes (//) not followed by a slash (in the first character of *filename*) indicate that the file is to be treated as a non-z/OS UNIX file, hereafter called a data set.

Note:

1. *filename* immediately follows the double slashes (//) without spaces.
2. Absolute data set names are specified by putting single quotation marks (') around the name. Refer to the syntax diagram in this topic for this specification.
3. Absolute z/OS UNIX file names are specified by putting a leading slash (/) as the first character in the file name.
4. ddnames are always considered absolute.

Forming file names

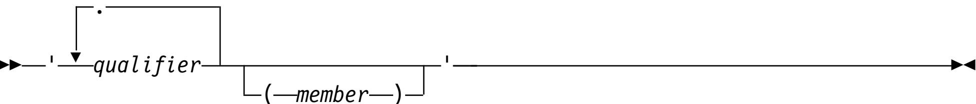
Refer to "Determining whether the file name is in absolute form" on page 178 for information on absolute file names. When the compiler performs a library search, it treats *filename* as either a z/OS UNIX System Services file name or a data set name. This depends on whether the library being searched is a z/OS UNIX library or MVS library. If the compiler treats *filename* as a z/OS UNIX file name, it does not perform any conversions on it. If it treats *filename* as a data set name (DSN), it performs the following conversion:

- For the first DSN format:

  The compiler:

  1. Uppercases *qualifier* and *path*
  2. Truncates each *qualifier* and *path* to 8 characters
  3. Converts the underscore character (which is invalid for a DSN) to the '@' character (hex 7c)

- For the second DSN format:

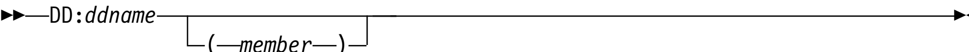


  The compiler:

  1. Uppercases the *qualifier* and *member*

2. Converts the underscore character (which is invalid for a DSN) to the '@' character (hex 7c)

- For the third DSN format:



The compiler:

1. Uppercases the `DD:`, *ddname*, and *member*
2. Converts the underscore character (which is invalid for a DSN) to the '@' character (hex 7c)

Forming data set names with LSEARCH | SEARCH options

When the *filename* specified in the `#include` directive is not in absolute form, the compiler combines it with different types of libraries to form complete data set specifications. These libraries may be specified by the LSEARCH or SEARCH compiler options. When the LSEARCH or SEARCH option indicates a data set then, depending on whether it is a ddname, sequential data set, or PDS, different parts of *filename* are used to form the ddname or data set name.

Forming DDname

Example: The leftmost qualifier of the *filename* in the `#include` directive is used when the *filename* is to be a ddname:

Invocation:
```
SEARCH(DD:SYSLIB)
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting ddname:
```
DD:SYSLIB(AFILE)
```

In this example, if your header file includes an underscore (_), for example, `#include "sys/afile_1.g.h"`, the resulting ddname is `DD:SYSLIB(AFILE@1)`.

Forming sequential data set names

Example: You specify libraries in the SEARCH | LSEARCH options as sequential data sets by using a trailing period followed by an asterisk (.*), or by a single asterisk (*). See "LSEARCH | NOLSEARCH" on page 91 to understand how to specify sequential data sets. All *qualifier*s and periods (.) in *filename* are used for sequential data set specification.

Invocation:
```
SEARCH(AA.*)
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:
```
userid.AA.AFILE.G.H
```

Forming PDS name with LSEARCH | SEARCH + specification

Example: To specify libraries in the SEARCH and LSEARCH options as PDSs, use a period that is followed by a plus sign (.+), or a single plus sign (+). See "LSEARCH | NOLSEARCH" on page 91 to understand how PDSs are specified. When this is the case then all the *path*s, slashes (replaced by periods), and any *qualifier*s following the leftmost *qualifier* of the *filename* are appended to form the data set name. The leftmost *qualifier* is then used as the member name.

Invocation:

```
SEARCH('AA.+')
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:

```
AA.SYS.G.H(AFILE)
```

and

Invocation:

```
SEARCH('AA.+')
```

Include directive:

```
#include "sys/bfile"
```

Resulting fully qualified data set name:

```
AA.SYS(BFILE)
```

Forming PDS with LSEARCH | SEARCH Options without +

Example: When the LSEARCH or SEARCH option specifies a library but it neither ends with an asterisk (*) nor a plus sign (+), it is treated as a PDS. The leftmost qualifier of the *filename* in the `#include` directive is used as the member name.

Invocation:

```
SEARCH('AA')
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:

```
AA(AFILE)
```

Examples of forming data set names

The following table gives the original format of the *filename* and the resulting converted name when you specify the NOOE option:

Table 28. Include filename conversions when NOOE is specified

`#include` Directive	Converted Name
Example 1. This *filename* is absolute because single quotation marks (') are used. It is a sequential data set. A library search is not performed. LSEARCH is ignored.	
#include "'USER1.SRC.MYINCS'"	USER1.SRC.MYINCS
Example 2. This *filename* is absolute because single quotation marks (') are used. The compiler attempts to open data set COMIC/BOOK.OLDIES.K and fails because it is not a valid data set name. A library search is not performed when *filename* is in absolute form. SEARCH is ignored.	
#include <'COMIC/BOOK.OLDIES.K'>	COMIC/BOOK.OLDIES.K

Table 28. Include filename conversions when NOOE is specified (continued)

#include Directive	Converted Name
Example 3.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include "sys/abc/xx"	• first *opt* in SEARCH SEQUENTIAL FILE = *userid*.LIB1.XX • second *opt* in SEARCH PDS = *userid*.LIB2.SYS.ABC(XX) • third *opt* in SEARCH PDS = *userid*.LIB3(XX)
Example 4.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include "Sys/ABC/xx.x"	• first *opt* in SEARCH SEQUENTIAL FILE = *userid*.LIB1.XX.X • second *opt* in SEARCH PDS = *userid*.LIB2.SYS.ABC.X(XX) • third *opt* in SEARCH PDS = *userid*.LIB3(XX)
Example 5.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include <sys/name_1>	• first *opt* in SEARCH SEQUENTIAL FILE = *userid*.LIB1.NAME@1 • second *opt* in SEARCH PDS = *userid*.LIB2.SYS(NAME@1) • third *opt* in SEARCH PDS = *userid*.LIB3(NAME@1)
Example 6.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include <Name2/App1.App2.H>	• first *opt* in SEARCH SEQUENTIAL FILE = *userid*.LIB1.APP1.APP2.H • second *opt* in SEARCH PDS = *userid*.LIB2.NAME2.APP2.H(APP1) • third *opt* in SEARCH PDS = *userid*.LIB3(APP1)
Example 7. The PDS member named YEAREND of the library associated with the ddname PLANLIB is used. A library search is not performed when *filename* in the #include directive is in absolute form (ddname is used). SEARCH is ignored.	
#include <dd:planlib(YEAREND)>	DD:PLANLIB(YEAREND)

Search sequence

The following diagram describes the compiler file searching sequence:

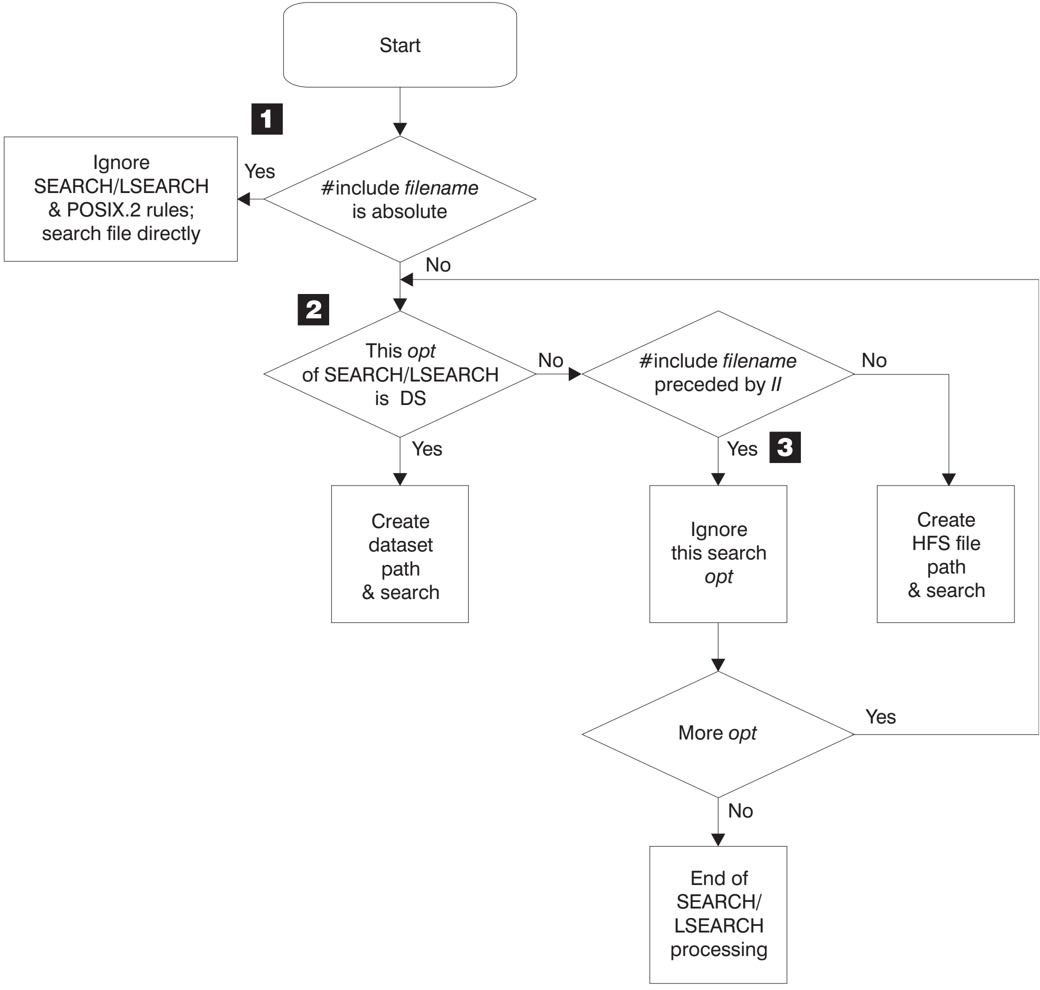


Figure 4. Overview of include file searching

1 The compiler opens the file without library search when the file name that is specified in `#include` is in absolute form. This also means that it bypasses the rules for the SEARCH and LSEARCH compiler options, and for POSIX.2. See Figure 5 on page 179 for more information on absolute file testing.

2 When the file name is not in absolute form, the compiler evaluates each option in SEARCH and LSEARCH to determine whether to treat the file as a data set or a z/OS UNIX System Services file search. The LSEARCH/SEARCH *opt* testing here is described in Figure 6 on page 181.

3 When the `#include` file name is not absolute, and is preceded by exactly two slashes (//), the compiler treats the file as a data set. It then bypasses all z/OS UNIX file options of the SEARCH and LSEARCH options in the search.

Determining whether the file name is in absolute form

The compiler determines if the file name that is specified in `#include` is in absolute form as follows:

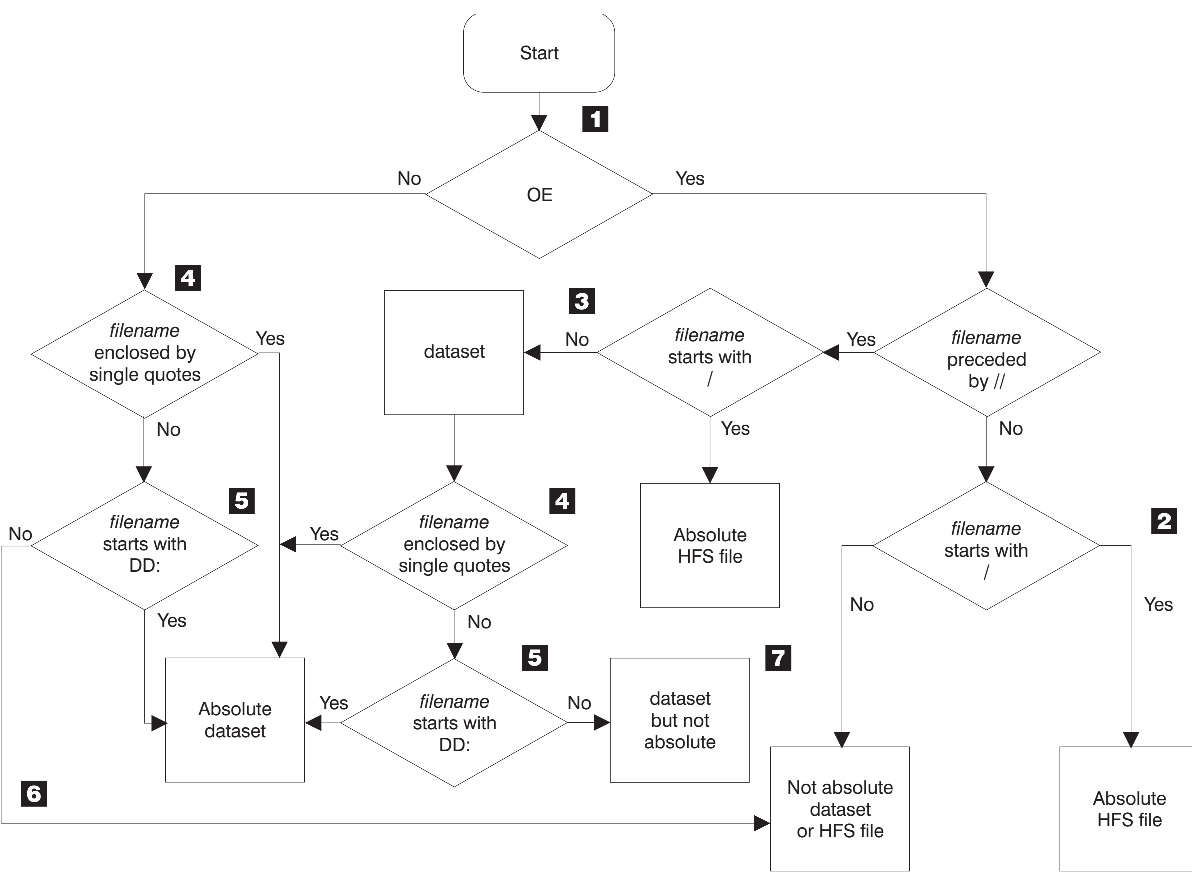


Figure 5. Testing if filename is in absolute form

1 The compiler first checks whether you specified OE.

2 When you specify OE, if double slashes (//) do not precede *filename*, and the file name starts with a slash (/), then *filename* is in absolute form and the compiler opens the file directly as a z/OS UNIX file. Otherwise, the file is not an absolute file and each *opt* in the SEARCH or LSEARCH compiler option determines if the file is treated as a z/OS UNIX file or data set in the search for the include file.

3 When OE is specified, if double slashes (//) precede *filename*, and the file name starts with a slash (/), then *filename* is in absolute form and the compiler opens the file directly as a z/OS UNIX file. Otherwise, the file is a data set, and more testing is done to see if the file is absolute.

4 If *filename* is enclosed in single quotation marks ('), then it is an absolute data set. The compiler directly opens the file and ignores the libraries that are specified in the LSEARCH or SEARCH options. If there are any invalid characters in *filename*, the compiler converts the invalid characters to at signs (@, hex 7c).

5 If you used the ddname format of the `#include` directive, the compiler uses the file associated with the ddname and directly opens the file as a data set. The libraries that are specified in the LSEARCH or SEARCH options are ignored.

6 If none of the conditions are true then *filename* is not in absolute format

and each *opt* in the SEARCH or LSEARCH compiler option determines if the file is a z/OS UNIX file or a data set and then searches for the include file.

7 If none of the conditions are true, then *filename* is a data set, but it is not in absolute form. Only *opt*s in the SEARCH or LSEARCH compiler option that are in data set format are used in the search for include file.

For example:

```
 Options specified:

  OE

Include Directive:

  #include "apath/afile.h"       NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "/apath/afile.h"      absolute z/OS UNIX file,
                                 (starts with 1 slash)
  #include "//apath/afile.h.c"   NOT absolute, MVS (starts with 2 slashes)
  #include "a.b.c"               NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "///apath/afile.h"    absolute z/OS UNIX file,
                                 (starts with 3 slashes)
  #include "DD:SYSLIB"           NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "//DD:SYSLIB"         absolute, MVS (DD name)
  #include "a.b(c)"              NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "//a.b(c)"            NOT absolute, OS/MVS (PDS member name)
```

Using SEARCH and LSEARCH

When the file name in the `#include` directive is not in absolute form, the *opt*s in SEARCH are used to find system include files and the *opt*s in LSEARCH are used to find user include files. Each *opt* is a library path and its format determines if it is a z/OS UNIX System Services path or a data set path:

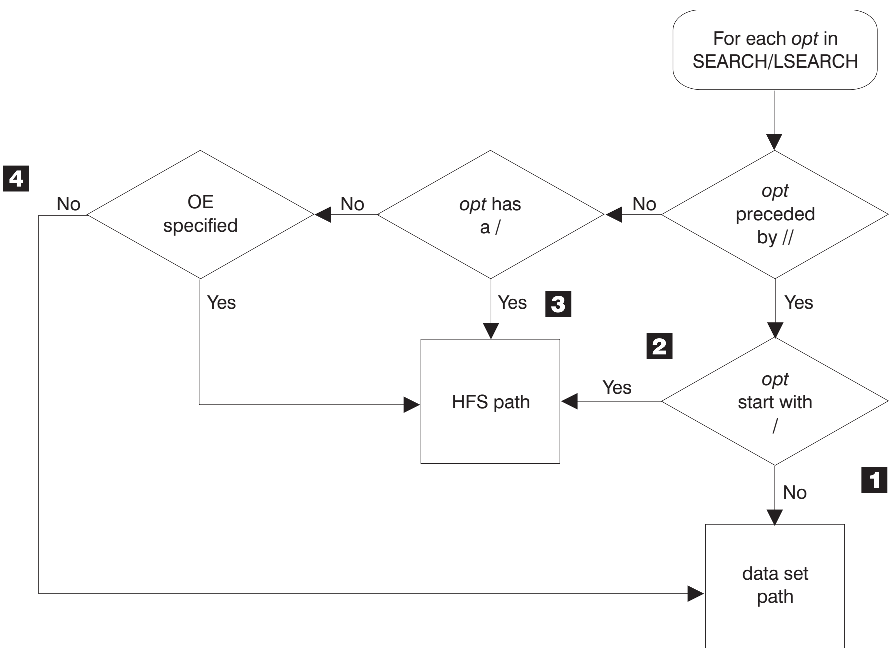


Figure 6. Determining if the SEARCH/LSEARCH opt is a z/OS UNIX path

Note:

1. If *opt* is preceded by double slashes (//) and *opt* does not start with a slash (/), then this path is a data set path.
2. If *opt* is preceded by double slashes (//) and *opt* starts with a slash (/), then this path is a z/OS UNIX path.
3. If *opt* is **not** preceded by double slashes (//) and *opt* starts with a slash (/), then this path is a z/OS UNIX path.
4. If *opt* is **not** preceded by double slashes (//), *opt* does not start with a slash (/) and NOOE is specified then this path is a data set path.

For example:

`SEARCH(./PATH)`	is an explicit z/OS UNIX path
`OE SEARCH(PATH)`	is treated as a z/OS UNIX path
`NOOE SEARCH(PATH)`	is treated as a non-z/OS UNIX path
`NOOE SEARCH(//PATH)`	is an explicit non-z/OS UNIX path

Example: When combining the library with the file name specified on the `#include` directive, it is the form of the library that determines how the include file name is to be transformed:

```
Options specified:

  NOOE LSEARCH(Z, /u/myincs, (*.h)=(LIB(mac1)))

Include Directive:

  #include "apath/afile.h"

Resulting fully qualified include names:

1. userid.Z(AFILE)  (Z is non-z/OS UNIX file so file name is treated
as non-z/OS UNIX file)
```

```
2. /u/myincs/apath/afile.h   (/u/myincs is z/OS UNIX file so
file name is treated as z/OS UNIX file)
3. userid.MAC1.H(AFILE)           (afile.h matches *.h)
```

Example: A z/OS UNIX path specified on a SEARCH or LSEARCH option only combines with the file name specified on an `#include` directive if the file name is not explicitly stated as being MVS only. A file name is explicitly stated as being MVS only if two slashes (//) precede it, and *filename* does not start with a slash (/).

```
Options specified:

  OE LSEARCH(/u/myincs, q, //w)

Include Directive:

  #include "//file.h"

Resulting fully qualified include names

 userid.W(FILE)
```

`/u/myincs` and `q` would not be combined with `//file.h` because both paths are z/OS UNIX paths and `//file.h` is explicitly MVS.

The order in which options on the LSEARCH or SEARCH option are specified is the order that is searched.

See "LSEARCH | NOLSEARCH" on page 91 and "SEARCH | NOSEARCH" on page 126 for more information on these compiler options.

Search sequences for include files

The search path is a list of include paths, each of which may form the start of a fully qualified file name. The include path can be specified through the -I option. For the compiler, it can also be specified through the SEARCH and LSEARCH options.

If the same z/OS UNIX System Services directory is specified in the search path multiple times, then only the first one is used. For example, `/usr/include` and `/usr/include/sys/..` resolve to the same z/OS UNIX System Services directory, therefore only the first path will be used in the final search path.

The status of the OE option affects the search sequence.

With the NOOE option

Search sequences for include files are used when the include file is not in absolute form. "Determining whether the file name is in absolute form" on page 178 describes the absolute form of include files.

If the include filename is not absolute, the compiler performs the library search as follows:

- For system include files:
  1. The search order as specified on the SEARCH option, if any
  2. The libraries specified on the SYSLIB DD statement
- For user include files:
  1. The libraries specified on the USERLIB DD statement

2. The search order for system include files

Example: This example shows an excerpt from a JCL stream, that compiles a C program for a user whose user prefix is JONES:

```
//COMPILE  EXEC PROC=MTCC,
//              CPARM='SEARCH(''''BB.D'''',BB.F),LSEARCH(CC.X)'
//SYSLIB   DD DSN=JONES.ABC.A,DISP=SHR
//         DD DSN=ABC.B,DISP=SHR
//USERLIB  DD DSN=JONES.XYZ.A,DISP=SHR
//         DD DSN=XYZ.B,DISP=SHR
//SYSIN    DD DSN=JONES.ABC.C(D),DISP=SHR
   .
   .
   .
```

The search sequence that results from the preceding JCL statements is:

Table 29. Order of search for include files

Order of Search	For System Include Files	For User Include Files
First	BB.D	JONES.CC.X
Second	JONES.BB.F	JONES.XYZ.A
Third	JONES.ABC.A	XYZ.B
Fourth	ABC.B	BB.D
Fifth		JONES.BB.F
Sixth		JONES.ABC.A
Seventh		ABC.B

With the OE option

Search sequences for include files are used when the include file is not in absolute form. “Determining whether the file name is in absolute form” on page 178 describes the absolute form of an include file.

If the include filename is not absolute, the compiler performs the library search as follows:

- For system include files:
  1. The search order as specified on the SEARCH option, if any
  2. The libraries specified on the SYSLIB DD statement
- For user include files:
  1. If you specified OE with a file name and the including file is a z/OS UNIX file and a main source file, the directory of the file name specified with the OE option; otherwise, the directory of the including file
  2. The search order as specified by the LSEARCH option, if any
  3. The libraries specified on the USERLIB DD statement
  4. The search order for system include files

Example: The following shows an example where you are given a file /r/you/cproc.c that contains the following #include directives:

```
#include "/u/usr/header1.h"
#include "//aa/bb/header2.x"
#include "common/header3.h"
#include <header4.h>
```

And the following options:

```
OE(/u/crossi/myincs/cproc)
SEARCH(//V.+, /new/inc1, /new/inc2)
LSEARCH(//(*.x)=(lib(AAA)), /c/c1, /c/c2)
```

The include files would be searched as follows:

Table 30. Examples of search order for z/OS UNIX

`#include` Directive Filename	**Files in Search Order**
Example 1. This is an absolute pathname, so no search is performed.	
`#include "/u/usr/header1.h"`	1. /u/usr/header.h
Example 2. This is a data set (starts with //) and is treated as such.	
"//aa/bb/header2.x"	1. *userid*.AAA(HEADER2) 2. DD:USERLIB(HEADER2) 3. *userid*.V.AA.BB.X(HEADER2) 4. DD:SYSLIB(HEADER2)
Example 3. This is a user include file with a relative path name. The search starts with the directory of the parent file or the name specified on the OE option if the parent is the main source file (in this case the parent file is the main source file so the OE suboption is chosen i.e. /u/crossi/myincs).	
"common/header3.h"	1. /u/crossi/myincs/common/header3.h 2. /c/c1/common/header3.h 3. /c/c2/common/header3.h 4. DD:USERLIB(HEADER3) 5. *userid*.V.COMMON.H(HEADER3) 6. /new/inc1/common/header3.h 7. /new/inc2/common/header3.h 8. DD:SYSLIB(HEADER3)
Example 4. This is a system include file with a relative path name. The search follows the order of suboptions of the SEARCH option.	
<header4.h>	1. *userid*.V.H(HEADER4) 2. /new/inc1/common/header4.h 3. /new/inc2/common/header4.h 4. DD:SYSLIB(HEADER4)

Chapter 4. Using IPA link step with programs

Traditional optimizers only have the ability to optimize within a function (*intra-procedural optimization*) or at most within a compilation unit (a single source file and its included header files). This is because traditional optimizers are only given one compilation unit at a time.

Interprocedural optimizations are a class of optimizations that operate across function boundaries. IBM's Interprocedural Analysis (IPA) optimizer is designed to optimize complete modules at a time. This allows for increased optimization. By seeing more of the application at once, IPA is able to find more opportunities for optimization and this can result in much faster code.

In order to get a global module view of the application, IPA uses the following two pass process:

- The first pass is called an *IPA Compile*. During this pass, IPA collects all of the relevant information about the compilation unit and stores it in the object file. This collected information is referred to as an IPA Object.
- The second pass is called the *IPA Link*. During this step, IPA acts like a traditional linker, and all object files, object libraries and side decks are fed to IPA so that it can optimize the entire module. The IPA link step involves two separate optimizers. The IPA optimizer is run first and focuses optimizations across the module. IPA then breaks down the module into logical chunks called partitions and invokes the traditional optimizer with these partitions.

Whenever a compiler attempts to perform more optimizations, or looks at a larger portion of an application, more time, and more memory are required. Since IPA does more optimizations than either OPT(2) or OPT(3) and has a global view of the module, the compile time and memory used by the IPA Compile or Link process is more than that used by a traditional OPT(2) or OPT(3) compilation.

Invoking IPA using metalc utility

You can invoke the IPA compile step, the IPA link step, or both. The step that **metalc** invokes depends upon the invocation parameters and type of files you specify.

The following command invokes the IPA compile step for the source file `hello.c`:

```
metalc -c -qipa hello.c
```

The following command invokes the IPA link step and generate the assembly source file:

```
metalc -qipa hello.o
```

Specifying options

The following example shows how to pass the IPA and the SOURCE options to the IPA compile step, and the MAXMEM(2048) option to both the IPA compile and the IPA link step.

```
metalc -O2 -qipa -qsource -qmaxmem=2048 hello.c
```

© Copyright IBM Corp. 2018

Other considerations

The `metalc` utility automatically generates all INCLUDE and LIBRARY IPA Link control statements.

IPA under `metalc` supports the following types of files:
- MVS PDS members
- Sequential data sets
- z/OS UNIX files
- z/OS UNIX archive (.a) files

Compiling under z/OS batch

To compile your C source program under batch, you can either use the cataloged procedures that IBM supplies, or write your own JCL statements.

Using cataloged procedures for IPA Link

You can use the following IBM-supplied cataloged procedure.

MTCI Run the IPA link step for a program.

Related information

Chapter 8, "Building Enterprise Metal C for z/OS programs," on page 197

Reference Information

The following topic provides reference information concerning the IPA link step control file, and object file directives understood by IPA.

IPA link step control file

The IPA link step control file is a fixed-length or variable-length format file that contains additional IPA processing directives. The CONTROL suboption of the IPA compiler option identifies this file.

The IPA link step issues an error message if any of the following conditions exist in the control file:
- The control file directives have invalid syntax.
- There are no entries in the control file.
- Duplicate names exist in the control file.

You can specify the following directives in the control file.

csect=*csect_names_prefix*
: Supplies information that the IPA link step uses to name the CSECTs for each partition that it creates. The *csect_names_prefix* parameter is a comma-separated list of tokens that is used to construct CSECT names.

  The behavior of the IPA link steps varies depending upon whether you specify the CSECT option with a qualifier.
  - If you do not specify the CSECT option with a qualifier, the IPA link step does the following:
    - Truncates each name prefix or pads it at the end with @ symbols, if necessary, to create a 7 character token

– Uppercases the token
– Adds a suffix to specify the type of CSECT, as follows:

C code

S static data

T test

- If you specify the CSECT option with a non-null qualifier, the IPA link step does the following:
  – Uppercases the token
  – Adds a suffix to specify the type of CSECT, as follows where *qualifier* is the qualifier you specified for CSECT and *nameprefix* is the name you specified in the IPA link step Control File:

  qualifier#*nameprefix*#**C**
  code

  qualifier#*nameprefix*#**S**
  static data

  qualifier#*nameprefix*#**T**
  test

- If you specify the CSECT option with a null qualifier, the IPA link step does the following:
  – Uppercases the token
  – Adds a suffix to specify the type of CSECT, as follows where *nameprefix* is the name you specified in the IPA link step Control File:

  nameprefix#**C**
  code

  nameprefix#**S**
  static data

  nameprefix#**T**
  test

The IPA link step issues an error message if you specify the CSECT option but no control file, or did not specify any `csect` directives in the control file. In this situation, IPA generates a CSECT name and an error message for each partition.

The IPA link step issues a warning or error message (depending upon the presence of the CSECT option) if you specify CSECT name prefixes, but the number of entries in the `csect_names` list is fewer than the number of partitions that IPA generated. In this situation, for each unnamed partition, the IPA link step generates a CSECT name prefix with format `@CSnnnn`, where `nnnn` is the partition number. If you specify the CSECT option, the IPA link step also generates an error message for each unnamed partition. Otherwise, the IPA link step generates a warning message for each unnamed partition.

inline=*name[,name]*
Specifies a list of functions that are desirable for the compiler to inline. The functions may or may not be inlined.

inline=*name[,name]* **from** *name[,name]*
Specifies a list of functions that are desirable for the compiler to inline, if the functions are called from a particular function or list of functions. The functions may or may not be inlined.

noinline=*name[,name]*
: Specifies a list of functions that the compiler will not inline.

noinline=*name[,name]* **from** *name[,name]*
: Specifies a list of functions that the compiler will not inline, if the functions are called from a particular function or list of functions.

exits=*name[,name]*
: Specifies names of functions that represent program exits. Program exits are calls that can never return, and can never call any procedure that was compiled with the IPA compile step.

lowfreq=*name[,name]*
: Specifies names of functions that are expected to be called infrequently. These functions are typically error handling or trace functions.

partition=small|*medium***|large|unsigned-integer**
: Specifies the size of each program partition that the IPA link step creates. When partition sizes are large, it usually takes longer to complete the code generation, but the quality of the generated code is usually better.

For a finer degree of control, you can use an *unsigned-integer* value to specify the partition size. The integer is in ACUs (Abstract Code Units), and its meaning may change between releases. You should only use this integer for very short term tuning efforts, or when the number of partitions (and therefore the number of CSECTs in the output object module) must remain constant.

The size of a CSECT cannot exceed 16 MB with the XOBJ format. Large CSECTs require the GOFF option.

The default for this directive is `medium`.

safe=*name[,name]*
: Specifies a list of *safe functions* that are not compiled as IPA objects. These are functions that do not call a visible (not missing) function either through a direct call or a function pointer. Safe functions can modify global variables, but may not call functions that are not compiled as IPA objects.

isolated=*name[,name]*
: Specifies a list of *isolated functions* that are not compiled as IPA objects. Neither isolated functions nor functions within their call chain can refer to global variables. IPA assumes that functions that are bound from shared libraries are isolated.

unknown=*name[,name]*
: Specifies a list of *unknown functions* that are not compiled as IPA objects. These are functions that are not safe or isolated. This is the default for all functions defined within non-IPA objects. Any function specified as unknown can make calls to other parts of the program compiled as IPA objects and modify global variables and dummy arguments. This option greatly restricts the amount of interprocedural optimization for calls to unknown functions.

missing=*attribute*

: Specifies the characteristics of *missing functions*, which are statically available but not compiled with the IPA option. IPA has no visibility to the code within these functions. You must ensure that all user references are resolved at IPA Link time with user libraries or runtime libraries.

The default setting for this directive is `unknown`. This instructs IPA to make pessimistic assumptions about the data that may be used and modified through a call to such a missing function, and about the functions that may be called indirectly through it.

You can specify the following attributes for this directive:

safe
Specifies that the missing functions are safe. See the description for the `safe` directive in this topic.

isolated
Specifies that the missing functions are isolated. See the description for the `isolated` directive in this topic.

unknown
Specifies that the missing functions are unknown. See the description for the `unknown` directive in this topic. This is the default attribute.

retain=*symbol-list*
Specifies a list of exported functions or variables that the IPA link step retains in the final object module. The IPA link step does not prune these functions or variables during optimization.

Note: In the listed directives, *name* can be a regular expression. Thus, *name* can match multiple symbols in your application through pattern matching. The regular expression syntax supported by the IPA control file processor is as follows:

Table 31. Syntax rules for specifying regular expressions

Expression	Description
string	Matches any of the characters specified in *string*. For example, `test` will match `testimony`, `latest`, and `intestine`.
^string	Matches the pattern specified by *string* only if it occurs at the beginning of a line.
string$	Matches the pattern specified by *string* only if it occurs at the end of a line.
str.ing	The period (.) matches any single character. For example, `t.st` will match `test`, `tast`, `tZst`, and `t1st`.
*string**special_char*	The backslash (\ ) can be used to escape special characters. For example, assume that you want to find lines ending with a period. Simply specifying the expression `.$` would show all lines that had at least one character of any kind in it. Specifying `\.$` escapes the period (.), and treats it as an ordinary character for matching purposes.
[*string*]	Matches any of the characters specified in *string*. For example, `t[a-g123]st` matches tast and test, but not `t-st` or `tAst`.
[^*string*]	Does not match any of the characters specified in *string*. For example, `t[^a-zA-Z]` st matches `t1st`, `t-st`, and `t,st` but not `test` or `tYst`.
*string**	Matches zero or more occurrences of the pattern specified by *string*. For example, `te*st` will match `tst`, `test`, and `teeeeeest`.
string+	Matches one or more occurrences of the pattern specified by *string*. For example, `t(es)+t` matches `test`, `tesest`, but not `tt`.
string?	Matches zero or one occurrences of the pattern specified by *string*. For example, `te?st` matches either `tst` or `test`.
string{*m*,*n*}	Matches between *m* and *n* occurrence(s) of the pattern specified by *string*. For example, `a{2}` matches `aa`, and `b{1,4}` matches `b`, `bb`, `bbb`, and `bbbb`.

Table 31. Syntax rules for specifying regular expressions (continued)

Expression	Description
string1 | *string2*	Matches the pattern specified by either *string1* or *string2*. For example, s | o matches both characters s and o.

Object file directives understood by IPA

IPA recognizes and acts on the following binder object control directives:

- INCLUDE
- LIBRARY
- IMPORT

Some other linkage control statements (such as NAME, RENAME and ALIAS) are accepted and passed through to the linker.

Troubleshooting

It is strongly recommended that you resolve all warnings that occur during the IPA link step. Resolution of these warnings often removes seemingly unrelated problems.

The following list provides frequently asked questions (Q) and their respective answers (A):

- Q - I am running out of memory while using IPA. Are there any options for reducing its use of memory and increasing the system-defined limits?

  A - IPA reacts to the NOMEMORY option, and the code generator will react to the MAXMEM option. If this does not give you sufficient memory, consider running IPA from batch where more memory can be accessed. Before switching to batch, verify with your system programmer that you have access to the maximum possible memory (both in batch and in z/OS UNIX System Services). You could also reduce the level of IPA processing via the IPA LEVEL suboption.
- Q - I am receiving a "partition too large" warning. How do I fix it?

  A - Use the IPA Control file to specify a different partition size.
- Q - My IPA Compile time is too long. Are there any options?

  A - Using a lower IPA compilation level (0 or 1 instead of 2) will reduce the compile time. A smaller partition size, specified in the control file, may minimize the amount of time spent in the code generator. Limiting inlining, may improve your compile time, but it will decrease your performance gain significantly and should only be done selectively using the IPA control file. Use the IPA control file to specify little used functions as low frequency so that IPA does not spend too much time trying to optimize them.
- Q - Can I tune the IPA automatic inlining like I can for the regular inliner?

  A - Yes. Use the INLINE option for the IPA link step.

Chapter 5. Assembling

The Enterprise Metal C for z/OS compiler produces the output in HLASM source code format. You need to assemble the HLASM source file to produce the output file.

Generating an object file from the HLASM source using the z/OS UNIX System Services as command

The generated object file does not have to be a z/OS UNIX file. The **as** command can write the object file directly to an MVS data set. The **-o** flag can be used to name the output file, where it can be a UNIX file or an MVS data set. For example:

```
as mycode.s
```

A successful assemble will produce `mycode.o`.

If the C source file was compiled with the LONGNAME compiler option, the generated HLASM source file will contain symbols that are more than eight characters in length. In that case, the HLASM GOFF option must be specified. Use the **as** utility **-m** flag to specify HLASM options. For example:

```
as -mgoff mycodelong.s
```

A successful assemble will produce `mycodelong.o`.

Generating an object file from the HLASM source under batch

To assemble the assembly source generated by the Enterprise Metal C for z/OS under batch, you can either use the cataloged procedures that IBM supplies or write your own JCL statements.

Using cataloged procedures for assembling

You can use one of the following IBM®-supplied cataloged procedures.

Cataloged procedures	Task Description
MTCCA	Compile and assemble a program
MTCIA	Compile a program with IPA link and assemble

Related information

Chapter 8, “Building Enterprise Metal C for z/OS programs,” on page 197

Chapter 6. Binding programs

This information describes how to bind your programs using the program management binder in the z/OS batch and z/OS UNIX System Services environments.

Binding under z/OS UNIX

You can use the z/OS UNIX System Services `ld` command to bind the object files.

Related information

Chapter 8, "Building Enterprise Metal C for z/OS programs," on page 197

Binding under z/OS batch

You can either use the cataloged procedure CEEWL that resides in CEE.SCEEPROC or write your own JCL statements.

Related information

Chapter 8, "Building Enterprise Metal C for z/OS programs," on page 197

© Copyright IBM Corp. 2018

Chapter 7. Running a C application

You can run an application that contains a main entry point under z/OS batch, TSO, and z/OS UNIX environment.

Related information

Chapter 8, “Building Enterprise Metal C for z/OS programs,” on page 197

© Copyright IBM Corp. 2018

Chapter 8. Building Enterprise Metal C for z/OS programs

The following two examples show how to compile, assemble, link, and run an Enterprise Metal C for z/OS program under z/OS UNIX and z/OS batch environment.

Under z/OS UNIX

The following example shows how to build a reentrant Enterprise Metal C for z/OS application for `mycode.c`.

```
/* mycode.c */
extern int b = 45;

int main(void) {
   int a=10;
   __asm (" AR %0,%1 " :"+r"(a) :"r"(b));
   return a;
}
```

1. Compile the program and generate `mycode.s`:

   ```
   metalc -qrent mycode.c
   ```

2. Assemble `mycode.s`:

   ```
   as -mgoff mycode.s     1
   ```

3. Link `mycode.o`:

   ```
   export _LD_SYSLIB="//'CJT.SCJTOBJ'"     2
   ld -o mycode -e main  mycode.o
   ```

4. Run `mycode`:

   ```
   mycode
   ```

Notes:

1. The HLASM GOFF option is required to assemble the compiler generated code for RENT.
2. CJT.SCJTOBJ data set is required for the RENT program to resolve CCNZINIT and CCNZTERM.

Under z/OS batch

The following JCL example shows how to compile, assemble, link, and run an Enterprise Metal C for z/OS program under z/OS batch.

```
//ORDER JCLLIB ORDER=(CJT.SCJTPRC)
//*******************************************************************
//* COMPILE AND ASSEMBLE STEP:
//*******************************************************************
//COMP EXEC MTCCA,
// LNGPRFX='CJT',CPARM='SO'
//SYSLIB DD DSN=CEE.SCEEMAC,DISP=SHR
// DD DSN=SYS1.MACLIB,DISP=SHR
//SYSIN DD *
int main(void) {
int a=10, b=45 ;
__asm (" AR %0,%1 " :"+r"(a) :"r"(b));
return a;
}
/*
//*
```

© Copyright IBM Corp. 2018

```
//********************************************************************
//* LINKEDIT STEP:
//********************************************************************
//LKED EXEC PGM=HEWL,
// REGION=1024K,PARM='AMODE=31 '
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLIN DD DSN=*.COMP.ASSEMBLE.SYSLIN,DISP=(OLD,DELETE)
//SYSLMOD DD DSN=&&GSET(GO),DISP=(NEW,PASS),
// SPACE=(TRK,(7,7,1)),UNIT=SYSALLDA
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(32000,(30,30))
//SYSPRINT DD SYSOUT=*
//SYSIN DD DUMMY
//*
//********************************************************************
//* GO STEP:
//********************************************************************
//GO EXEC PGM=*.LKED.SYSLMOD,PARM=' '
//SYSPRINT DD SYSOUT=*
//SYSCPRT DD SYSOUT=*
```

Chapter 9. Cataloged procedures

This information describes the cataloged procedures that the Enterprise Metal C for z/OS compiler provides to call the various Enterprise Metal C for z/OS utilities. You can use the following cataloged procedures for both 31-bit and 64-bit programs.

When you specify a data set name without enclosing it in single quotation marks ('), your user prefix will be added to the beginning of the data set name. If you enclose the data set name in quotation marks, it is treated as a fully qualified name.

Cataloged procedures	Task Description
CDAASMC	Compile Common Debug Architecture assembler code to generate both DWARF and ADATA debug information, by default.
MTCC	Compile a program
MTCCA	Compile and assemble a program
MTCI	Compile a program with IPA link
MTCIA	Compile a program with IPA link and assemble

Tailoring cataloged procedures

A system programmer must modify the cataloged procedures before they are used.

The following data sets contain the cataloged procedures that are to be modified:
- CJT.SCJTPRC
- CEE.SCEEPROC

The IBM-supplied cataloged procedures provide many parameters to allow each site to customize them easily. The table below describes the commonly used parameters. Use only those parameters that apply to the cataloged procedure you are using. For example, if you are only compiling (MTCC), do not specify any binder parameters.

Parameter	Description
INFILE	For compile procedures, the input source file name, PDS name of source files, or directory name of source files. For IPA Link procedures (for example, MTCI), the input IPA object. If you do not specify the input data set name, you must use JCL statements to override the appropriate SYSIN DD statement in the cataloged procedure.
OUTFILE	Output module name and file characteristics. For compile procedures, specify the name of the file where the assembly source is to be stored. For assembly procedures, specify the name of the object module is to be stored. If you do not specify an OUTFILE name, a temporary data set will be generated.

© Copyright IBM Corp. 2018

Parameter	Description
CPARM	Compiler options: If two contradictory options are specified, the last is accepted and the first ignored.
APARM	Assembler options: If two contradictory options are specified, the last is accepted and the first ignored.
IPARM	IPA link step options: If two contradictory options are specified, the last is accepted and the first ignored.
CRUN	Compile step execution runtime parameters for the compiler.
IRUN	IPA link step runtime parameters: for the compiler.

Data sets used

The following table gives a cross-reference of the data sets that each job step requires, and a description of how the data set is used.

Table 32. Cross reference of data set used and job step

DD Statement	COMPILE	IPALINK	ASSEMBLE
STEPLIB[1]	X	X	
SYSCPRT	X	X	
SYSIN	X	X	X
SYSLIB	X	X	X
SYSLIN	X	X	X
SYSOUT	X	X	
SYSPRINT		X	
SYSUTx	X	X	X
IPACNTL		X	

Note: [1] Optional data sets, if the compiler is in DLPA and the runtime library is in LPA, DLPA, or ELPA. To save resources (especially in z/OS UNIX System Services), do not unnecessarily specify data sets on the STEPLIB ddname.

Description of data sets used

The following table lists the data sets that the IBM-supplied cataloged procedures use. It describes the uses of the data set, and the attributes that it supports. You require compiler work data sets only if you specified NOMEM at compile time.

Notes:

1. You should check the defaults at your site for SYSOUT=*
2. The compiler does not directly deal with the SYSOUT DD statement. It uses `stderr`, which in turn is associated with SYSOUT. However, this is just a default ddname, which can be changed by specifying the MSGFILE runtime option. Since the compiler does not directly deal with the DD statement associated with the `stderr`, it cannot provide an alternate DD statement for SYSOUT. Applications that invoke the compiler using one of the documented assembler macros can affect the DD statement that is associated with the `stderr` only by specifying the MSGFILE runtime option in the parameter list, but not via an alternate DD statement.

Table 33. Data set descriptions for cataloged procedures

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
COMPILE	SYSIN	For a C compilation, the output data set containing the assembler source. For an IPA compilation, the output data set containing the IPA object. RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760. It can be a PDS.
COMPILE	SYSLIB	For a C or IPA compilation, the data set for Enterprise Metal C for z/OS system header files for a source program. SYSLIB must be a PDS or PDSE (DSORG=PO) and RECFM=VS, V, VB, VBS, F, FB LRECL≤32760. RECFM cannot be mixed. The LRECLs for F or FB RECFM must match. For more information on searching system header files, see "SEARCH | NOSEARCH" on page 126.
COMPILE	SYSLIN	Data set for object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS. It can be a PDS. LRECL=80
COMPILE	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*).
COMPILE	STEPLIB	Data set for Enterprise Metal C for z/OS compiler and runtime library modules. STEPLIB must be a PDS or PDSE (DSORG=PO) with RECFM=U, BLKSIZE=32760, LRECL=0.
COMPILE	SYSCPRT	Output data set for compiler listing. LRECL>=137, RECFM=VB,VBA, BLKSIZE=882 (default for SYSOUT=*) LRECL=133, RECFM=FB,FBA, BLKSIZE=133*n(where *n* is an integer value)
COMPILE	SYSUT1	Obsolete work data set. LRECL=80 and RECFM=F or FB or FBS.

Table 33. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
COMPILE	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200**n* (where *n* is an integer value).
COMPILE	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137**n* (where *n* is an integer value).
COMPILE	SYSUT10	PPONLY output data set. 72≤LRECL≤32760, RECFM=VS, V, VB, VBS, F, FB, FBS or FS (if not pre-allocated, V is the default). It can be a PDS.
COMPILE	SYSUTIP	Work data set. LRECL=3200, RECFM=FB, BLKSIZE=3200**n* (where *n* is an integer value), DSORG=PO, and DSNTYPE=LIBRARY.
COMPILE	SYSEVENT	Events output file. Must be allocated by the user. For a description of this file, see "EVENTS | NOEVENTS" on page 55.
COMPILE	USERLIB	User header files. Must be a PDS or PDSE. LRECL≤32760, and RECFM=VS, V, VB, VBS, F or FB. For more information on searching user header files, see "SEARCH | NOSEARCH" on page 126.
IPALINK	SYSIN	Data set containing object module for the IPA link step. LRECL=80 and RECFM=F or FB.
IPALINK	IPACNTL	IPA Link control file directives. RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760.

Table 33. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
IPALINK	SYSLIB	IPA link step secondary input. SYSLIB can be a mix of two types of libraries: • Object module libraries. These can be PDSs (DSORG=PO) or PDSEs, with attributes RECFM=F or RECFM=FB, and LRECL=80. • Load module libraries. These must be PDSs (DSORG=PO) with attributes RECFM=U and BLKSIZE≤32760. SYSLIB member libraries must be cataloged.
IPALINK	SYSLIN	The output data set containing the assembler source. One of the following: • RECFM=F or FS • RECFM=FB or FBS
IPALINK	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*).
IPALINK	STEPLIB	Data set for Enterprise Metal C for z/OS compiler/runtime library modules. STEPLIB must be a PDS or PDSE (DSORG=PO) with RECFM=U, BLKSIZE≤32760.
IPALINK	SYSCPRT	Output data set for IPA link step listings. LRECL=137, RECFM=VBA, BLKSIZE=882 (default for SYSOUT=*).
IPALINK	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200*_n_ (where _n_ is an integer value).
IPALINK	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137*_n_ (where _n_ is an integer value).
IPALINK	SYSUTIP	Work data set. LRECL=3200, RECFM=FB, BLKSIZE=3200*_n_ (where _n_ is an integer value), DSORG=PO, and DSNTYPE=LIBRARY.

Table 33. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
ASSEMBLE	SYSLIN	Data set for object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS. It can be a PDS. LRECL=80
ASSEMBLE	SYSLIB	The data set identifies the macro library to be used when assembling the assembler source code.
ASSEMBLE	SYSIN	The data set containing the assembly source. • LREL=80 • RECFM=F or FB

Chapter 10. CDAHLASM — Use the HLASM assembler to create DWARF debug information

Description

The CDAHLASM utility is the MVS batch equivalent of the as utility. This utility is shipped as part of the Run-Time Library Extensions and is installed in CEE.SCEERUN2.

The compiler generates output in the form of assembler source. The compiler cannot generate DWARF information directly because it cannot create symbolic debugging information. The symbolic debugging information can be obtained only during object code generation, in this case, during the assembly stage.

Debuggers can use the DWARF-formatted output from the CDAHLASM utility to debug Metal C applications. To enable the generation of complete DWARF information, the compiler embeds the type information, created during the compilation stage, into the generated assembler source output. The assembly stage takes the embedded information, and combines it with the symbolic debugging information obtained during assembling, and produces the final DWARF information side file.

The CDAHLASM utility also produces debug information in ADATA format, which is required for the generation of DWARF information. The ADATA assembler option will be passed to the assembler unless the NODEBUG option is passed to CDAHLASM.

The compiler might put a debug data block in the generated assembly file. The CDAHLASM utility gets the MD5 signature from the debug data block, if the block exists, and puts the signature in the debug side file. In addition, the compiler generates a placeholder for the debug side file name in the debug data block. If the CDAHLASM utility has the write permission to the assembly file, it will update the assembly file by replacing the debug side file name in the debug data block with the user provided name or a default debug side file name. Otherwise, the CDAHLASM utility will fail to update the debug side file name in the debug data block.

For information on the CDAASMC cataloged procedure, which executes the CDAHLASM utility, see Chapter 9, "Cataloged procedures," on page 199.

Options directed to the CDAHLASM utility can be specified only through the DD:CDAHOPT.

Options

PHASEID
: Displays the version of CDAHLASM as well as the Common Debug Architecture runtime `phaseid` information.

NODEBUG
: Suppresses the generation of DWARF debug information.

© Copyright IBM Corp. 2018

VERBOSE
Specifies verbose mode, which writes additional informational messages to DD:SYSOUT.

Chapter 11. make utility

This information describes the z/OS UNIX System Services **make** utility. For information on the syntax and use, refer to *z/OS UNIX System Services Command Reference*.

The z/OS Shell and Utilities provides the **make** utility that you can use to simplify the task of creating and managing z/OS UNIX System Services Enterprise Metal C for z/OS application programs. You can use the **make** utility with the **metalc** utility to build application programs into easily updated and maintained executable files.

Creating makefiles

The **make** utility maintains all the parts of and dependencies for your application program. It uses a makefile to keep your application parts (listed in it) up to date with one another. If one part changes, the **make** utility updates all the other files that depend on the changed part.

A *makefile* is a z/OS UNIX text file. You can use any text editor to create and edit the file. It describes the application program files, their locations, dependencies on other files, and rules for building the files into an executable file. When creating a makefile, remember that tabbing of information in the file is important and not all editors support tab characters the same way.

The **make** utility uses **make** dependencies to determine which targets require recompile and invokes **metalc** to do the recompile.

See *z/OS UNIX System Services Programming Tools* and *z/OS UNIX System Services Command Reference* for a detailed discussion of the shell **make** utility.

You can use the **-M** flag option instructs the compiler to generate a dependency file or dependency files that can be used by the **make** utility.

The **-qmakedep** compiler option produces the dependency files that are used by the **make** utility for each source file.

Related information

For more information on related compiler options, see

- “-M” on page 227
- “MAKEDEP” on page 97

Chapter 12. BPXBATCH utility

This information provides a quick reference for the IBM-supplied BPXBATCH program. BPXBATCH makes it easy for you to run shell scripts and Enterprise Metal C for z/OS executable files that reside in z/OS UNIX files through the z/OS batch environment. If you do most of your work from TSO/E, use BPXBATCH to avoid going into the shell to run your scripts and applications.

In addition to using BPXBATCH, if you want to perform a local spawn without being concerned about environment set-up (that is, without having to set specific environment variables, which could be overwritten if they are also set in your profile) you can use BPXBATSL. BPXBATSL, which provide you with an alternate entry point into BPXBATCH, and force a program to run using a local spawn instead of fork or exec as BPXBATCH does. This ultimately allows a program to run faster.

BPXBATSL is also useful when you want to perform a local spawn of your program, but also need subsequent child processes to be forked or executed. Formerly, with BPXBATCH, this could not be done since BPXBATCH and the requested program shared the same environment variables. BPXBATSL is provided as an alternative to BPXBATCH. It will force the running of the target program into the same address space as the job itself is initiated in, so that all resources for the job can be used by the target program; for example, DD allocations. In all other respects, it is identical to BPXBATCH.

BPXBATCH usage

The BPXBATCH program allows you to submit z/OS batch jobs that run shell commands, scripts, or Enterprise Metal C for z/OS executable files in z/OS UNIX files from a shell session. You can invoke BPXBATCH from a JCL job, from TSO/E (as a command, through a CALL command, from a REXX EXEC).

JCL: Use one of the following:
- `EXEC PGM=BPXBATCH,PARM='SH program-name'`
- `EXEC PGM=BPXBATCH,PARM='PGM program-name'`

TSO/E: Use one of the following:
- `BPXBATCH SH program-name`
- `BPXBATCH PGM program-name`

BPXBATCH allows you to allocate the z/OS standard files `stdin`, `stdout`, and `stderr` as z/OS UNIX files for passing input, for shell command processing, and writing output and error messages. If you do allocate standard files, they must be z/OS UNIX files. If you do not allocate them, `stdin`, `stdout`, and `stderr` default to `/dev/null`. You allocate the standard files by using the options of the data definition keyword `PATH`.

Note: The BPXBATCH utility also uses the STDENV file to allow you to pass environment variables to the program that is being invoked. This can be useful when not using the shell, such as when using the PGM parameter.

© Copyright IBM Corp. 2018

Example: For JCL jobs, specify PATH keyword options on DD statements; for example:

```
//jobname JOB ...

//stepname EXEC PGM=BPXBATCH,PARM='PGM program-name parm1 parm2'

//STDIN   DD  PATH='/stdin-file-pathname',PATHOPTS=(ORDONLY)
//STDOUT  DD  PATH='/stdout-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//            PATHMODE=SIRWXU
//STDERR  DD  PATH='/stderr-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//            PATHMODE=SIRWXU
⋮
```

You can also allocate the standard files dynamically through use of SVC 99.

For TSO/E, you specify PATH keyword options on the ALLOCATE command. For example:

```
ALLOCATE FILE(STDIN) PATH('/stdin-file-pathname') PATHOPTS(ORDONLY)
ALLOCATE FILE(STDOUT) PATH('/stdout-file-pathname')
         PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)
ALLOCATE FILE(STDERR) PATH('/stderr-file-pathname')
         PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)

BPXBATCH SH program-name
```

You must always allocate `stdin` as read. You must always allocate `stdout` and `stderr` as write.

Parameter

BPXBATCH accepts one parameter string as input. At least one blank character must separate the parts of the parameter string. When BPXBATCH is run from a batch job, the total length of the parameter string must not exceed 100 characters. When BPXBATCH is run from TSO, the parameter string can be up to 500 characters. If neither SH nor PGM is specified as part of the parameter string, BPXBATCH assumes that it must start the shell to run the shell script allocated by `stdin`.

SH | PGM
Specifies whether BPXBATCH is to run a shell script or command or an Enterprise Metal C for z/OS executable file that is located in a z/OS UNIX file.

SH Instructs BPXBATCH to start the shell, and to run shell commands or scripts that are provided from `stdin` or the specified *program-name*.

Note: If you specify SH with no program-name information, BPXBATCH attempts to run anything read in from `stdin`.

PGM Instructs BPXBATCH to run the specified *program-name* as a called program.

If you specify PGM, you must also specify *program-name*. BPXBATCH creates a process for the program to run in and then calls the program. The HOME and LOGNAME environment variables are set automatically when the program is run, only if they do not exist in the file that is referenced by STDENV. You can use STDENV to set these environment variables, and others.

program-name
Specifies the shell command or the z/OS UNIX path name for the shell script or Enterprise Metal C for z/OS executable file to be run. In addition, *program-name* can contain option information.

BPXBATCH interprets the program name as case-sensitive.

Note: When PGM and *program-name* are specified and the specified program name does not begin with a slash character (/), BPXBATCH prefixes your *initial* working directory information to the program path name.

Usage notes

You should be aware of the following:

1. BPXBATCH is an alias for the program BPXMBATC, which resides in the SYS1.LINKLIB data set.
2. BPXBATCH must be invoked from a user address space running with a program status word (PSW) key of 8.
3. BPXBATCH does not perform any character translation on the supplied parameter information. You should supply parameter information, including z/OS UNIX path names, using only the POSIX portable character set.
4. A program that is run by BPXBATCH cannot use allocations for any files other than `stdin`, `stdout`, or `stderr`.
5. BPXBATCH does not close file descriptors except for `0`, `1`, and `2`. Other file descriptors that are open and not defined as "marked to be closed" remain open when you call BPXBATCH. BPXBATCH runs the specified script or executable file.
6. BPXBATCH uses write-to-operator (WTO) routing code 11 to write error messages to either the JCL job log or your TSO/E terminal. Your TSO/E user profile must specify WTPMSG so that BPXBATCH can display messages to the terminal.

Files

The following list describes the files:

- SYS1.LINKLIB(BPXMBATC) is the BPXBATCH program location.
- The `stdin` default is `/dev/null`.
- The `stdout` default is `/dev/null`.
- The `stdenv` default is `/dev/null`.
- The `stderr` default is the value of `stdout`. If all defaults are accepted, `stderr` is `/dev/null`.

Chapter 13. as — Use the HLASM assembler to produce object files

Format

as
[**--***option*[, *option*] ...] ...
[**-a**[**egimrsx**][=*file*]] ...
[**-g**]
[**--[no]gadata**[=*file*]]
[**--[no]gdwarf4**[=*file*]]
[**-m***option*]
[**-I** *name*]
[**-o** *objectfile*]
[**-d** *textfile*]
[**-v**]
[**--[no]help**]
[**--[no]verbose**]
file

Description

The **as** command processes assembler source files and invokes the HLASM assembler to produce object files.

Options

--
Accepts all options that are accepted by HLASM. Multiple options can be specified by separating them with a comma. This style of option specification is designed to provide smooth migration for users accustomed to specifying options in JCL. For example:

```
--"FLAG(ALIGN),RENT"
```

-a[egimrsx][=*file*]
Instructs the assembler to produce a listing.

-ae Instructs the assembler to produce the External Symbol Dictionary section of the assembler listing. This is equivalent to specifying: --ESD.

-ag Instructs the assembler to produce the General Purpose Register Cross Reference section of the assembler listing. This is equivalent to specifying: --RXREF.

-ai Instructs the assembler to copy all product information to the list data set. This is equivalent to specifying: --INFO.

-am Instructs the assembler to produce the Macro and Copy Code Source Summary section of the assembler listing. This is equivalent to specifying: --MXREF.

-ar Instructs the assembler to produce the Relocation Dictionary (RLD) section of the assembler listing. This is equivalent to specifying: --RLD.

-as Instructs the assembler to produce the Ordinary Symbol and

© Copyright IBM Corp. 2018

Literal Cross Reference section of the assembler listing. It also instructs the assembler to produce the un-referenced symbols defined in the CSECTs section of the assembler listing. This is equivalent to specifying: --XREF(SHORT,UNREFS).

-ax Instructs the assembler to produce the DSECT Cross Reference section of the assembler listing. This is equivalent to specifying: --DXREF.

=*file* Specifies the file name of the listing output. If you do not specify a file name, the output goes to `stdout`.

You may combine these options; for example, use **`-ams`** for an assembly listing with expanded macro and symbol output. The **`=file`** option, if used, must be specified last.

-g Instructs the assembler to collect debug information. By default, the debug information is produced in DWARF Version 4 format (or **`--gdwarf4`**).

--[no]gadata[=*file*]
Instructs the assembler to collect associated data and write it to the associated data file. You can optionally specify the name of the output debug file. The specified name cannot be a PDS or z/OS UNIX file system directory name. If you do not specify a file name, the default name is created as follows:

- If you are compiling a data set, the **`as`** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the user ID under which the **`as`** command is running, and .ADATA is appended as the low-level qualifier. For example, if `TS12345` is compiling `TSMYID.MYSOURCE(src)` with this option, the produced debug file name will be `TS12345.MYSOURCE.ADATA(src)`.
- If you are compiling a z/OS UNIX file, the **`as`** command stores the debug information in a file that has the name of the source file with an .ad extension. For example, if you are compiling `src.a` with this option, the compiler will create a debug file named `src.ad`.

--[no]gdwarf4[=*file*]
Instructs the assembler to generate debug information conforming to the DWARF Version 4 format. Debugging tools (for example, dbx) can take advantage of this debug information. You can optionally specify the name of the output debug file. The file name of the output debug file must be a PDS member, a sequential data set or z/OS UNIX file; it cannot be a PDS directory or z/OS UNIX System Services file system directory name. If you do not specify a file name, the default name is created as follows:

- If you are compiling a data set, the **`as`** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the userid under which the **`as`** command is running, and .DBG is appended as the low-level qualifier. For example, if `TS12345` is compiling `TSMYID.MYSOURCE(src)` with the **`-g`** option, the produced debug file name will be `TS12345.MYSOURCE.DBG(src)`. If `TS12345` is compiling `TSMYID.SEQSRC` with the **`-g`** option, the produced debug file name will be `TS12345.SEQSRC.DBG`.
- If you are compiling a z/OS UNIX file, the **`as`** command stores the debug information in a file that has the name of the source file with a .dbg extension. For example, if you are compiling `src.a` with the **`-g`** option, the produced debug file name will be `src.dbg`.

-m*option*

HLASM keyword options are specified using the following syntax:

```
-m<option>[=<parm>[=<value>][:<parm>[=<value>]]...]
```

where <option> is an option name, <parm> is a suboption name, and <value> is the suboption value.

Keyword options with no parameters represent switches that may be either on or off. The keyword by itself turns the switch on, and the keyword preceded by the letters NO turns the switch off. For example, **-mLIST** tells the HLASM assembler to produce a listing and **-mNOLIST** tells the HLASM assembler not to produce a listing. If an option that represents a switch is set more than once, the HLASM assembler uses the last setting.

Keyword option and parameter names may appear in mixed case letters in the invocation command.

-I *name*

Instructs HLASM to look for assembler macro invocation in the specified location. The *name* can be either a PDS name or z/OS UNIX file system directory name. If a PDS data set is specified, it must be fully qualified. The specified locations are then prepended to a default set of macro libraries. The **as** command assumes a default set of macro libraries that is compatible with the defaults for the compiler. The default data sets used are: -I CEE.SCEEMAC, -I SYS1.MACLIB, and -I SYS1.MODGEN. The default data sets can be changed via the environment variable _AS_MACLIB, for example:

```
export _AS_MACLIB="FIRST.PDS:SECOND.PDS"
```

-o *objectfile*

Specifies the name of the object file. If the name specified is a PDS or z/OS UNIX System Services directory name, a default file name is created in the PDS or z/OS UNIX directory specified as follows:

- If the source file is a sequential data set, the second last part of the data set name will be used. If the data set name only contains one part after the high-level qualifier, then the last part will be used.
- If the source file is a PDS member, the member name will be used.
- If the source file is a z/OS UNIX file, the suffix will be removed if applicable.
- If the object file is going into a PDS, the first eight characters of the name will be used. If there is a dot, anything after the first dot will be removed.
- If the object file is going into a z/OS UNIX directory, .o will be appended to the name.

For example:

```
Source file: //'abc.hello.source'
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o

Source file: //'ABC.HELLO'
Ouput file in PDS: HELLO
Output file in UNIX directory: HELLO.o

Source file: //SOURCE(hello)
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o
```

```
Source file: /abc/hello.s
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o

Source file: /abc/hellothere.s
Ouput file in PDS: HELLOTHE
Output file in UNIX directory: hellothere.o
```

-d *textfile*
Specifies the name of the object file output in text mode. If the name specified is a PDS or z/OS UNIX System Services directory name, a default file name is created in the PDS or z/OS UNIX directory with the same rule as **-o**.

-v
Writes the version of the **as** command to stderr.

--[no]help
Help menu. Displays the syntax of the **as** command.

--[no]verbose
Specifies verbose mode, which writes additional information messages to stderr.

file may be:
- An MVS data set (for example, //somename)
- An absolute z/OS UNIX file (for example, /somename)
- A relative z/OS UNIX file (for example, ./somename or somename)

The output of the **as** command is an object file. If you do not specify a file name via the **-o** option, the default name is created as follows:
- If you are compiling a data set, the **as** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the user ID under which the **as** command is running, and .OBJ is appended as the low-level qualifier. For example, if TS12345 is compiling TSMYID.MYSOURCE(src), the compiler will create an object file named TS12345.MYSOURCE.OBJ(src).
- If you are compiling a z/OS UNIX file, the **as** command names the object file with the name of the source file with an .o extension. For example, if you are compiling src.a, the object file name will be src.o.

Notes:
- The **as** command does not accept standard input as a file.
- The **as** command invokes the HLASM assembler to produce the object file. The HLASM assembler is invoked with the default options ASA and TERM. The ASA option instructs HLASM to use American National Standard printer control characters in records written to the listing file, thus making the listing file more readable in the z/OS UNIX System Services environment. The TERM option instructs HLASM to write error messages to stderr. These defaults can be changed by using the **-m** option or **--** option.
- HLASM messages and **as** error messages are directed to stderr. Verbose option output is directed to stdout.
- When invoking **as** from the shell, any option arguments or operands specified that contain characters with special meaning to the shell must be escaped. For example, source files specified as PDS member names contain parentheses; if they are specified as fully qualified names, they contain single quotation marks. To escape these special characters, either enclose the option argument or operand in double quotation marks, or precede each character with a backslash.

- The compiler might put a debug data block in the generated assembly file. The **as** utility gets the MD5 signature from the debug data block, if the block exists, and puts the signature in the debug side file. In addition, the compiler generates a placeholder for the debug side file name in the debug data block. If the **as** utility has the write permission to the assembly file, it will update the assembly file by replacing the debug side file name in the debug data block with the user provided name or a default debug side file name. Otherwise, the **as** utility will fail to update the debug side file name in the debug data block.

Chapter 14. metalc — Compiler invocation using a customizable configuration file

Format

metalc

Description

The **metalc** utility uses an external configuration file to control the invocation of the compiler and compiles C source files.

accept the following input files with their default z/OS UNIX file system and host suffixes:

z/OS UNIX files:
- filename with .c suffix (C source file)
- filename with .i suffix (preprocessed C source file)
- filename with .o suffix (object file for binder/IPA Link)
- filename with .s suffix (assembler source file)
- filename with .a suffix (archive library)
- filename with .x suffix (definition side-file or side deck)

Host files:
- filename with .C suffix (C source host file)
- filename with .CEX suffix (preprocessed C source host file)
- filename with .OBJ suffix (object host file for the binder/IPA Link)
- filename with .ASM suffix (assembler source host file)
- filename with .LIB suffix (host archive library)
- filename with .EXP suffix (host definition side-file or side deck)

Note: For host files, the host data set name must by preceded by a double slash (//). The last qualifier of the data set name is .C instead of a file name with a .C suffix.

Setting up the compilation environment

Before you compile your C programs, you must set up the environment variables and the configuration file for your application. For more information on the configuration file, see “Setting up a configuration file” on page 221.

Environment variables

You can use environment variables to specify necessary system information.

Setting environment variables

Different commands are used to set the environment variables depending on whether you are using the z/OS UNIX System Services shell (sh), which is based on the Korn Shell and is upward-compatible with the Bourne shell, or tcsh shell,

© Copyright IBM Corp. 2018

which is upward-compatible with the C shell. To determine the current shell, use the echo command, which is **echo $SHELL**.

The z/OS UNIX System Services shell path is /bin/sh. The tcsh shell path is /bin/tcsh.

For more information about the NLSPATH and LANG environment variables, see *z/OS UNIX System Services Command Reference*.

Setting environment variables in z/OS shell

The following statements show how you can set environment variables in the z/OS shell:

```
LANG=En_US
NLSPATH=/usr/lpp/IBM/cjt/v3r1/metalc/msg/%L/%N:/usr/lpp/IBM/cjt/v3r1/metalc/msg/%L/%N.cat
PATH=/bin:/usr/lpp/IBM/cjt/v3r1/metalc/bin${PATH:+:${PATH}}
export LANG NLSPATH PATH
```

To set the variables so that all users have access to them, add the commands to the file `/etc/profile`. To set them for a specific user only, add the commands to the `.profile` file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables in tcsh shell

The following statements show how you can set environment variables in the tcsh shell:

```
setenv LANG En_US
setenv NLSPATH /usr/lpp/IBM/cjt/v3r1/metalc/msg/%L/%N:/usr/lpp/IBM/cjt/v3r1/metalc/msg/%L/%N.cat
setenv PATH /bin:/usr/lpp/IBM/cjt/v3r1/metalc/bin${PATH:+:${PATH}}
```

To set the variables so that all users have access to them, add the commands to the file `/etc/csh.cshrc`. To set them for a specific user only, add the commands to the `.tcshrc` file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables for the message file

Before using the compiler, you must install the message catalogs and set the environment variables:

LANG
: Specifies the national language for message and help files.

NLSPATH
: Specifies the path name of the message and help files.

XL_CONFIG
: Specifies the name of an alternative configuration file (`.cfg`) for the **metalc** utility. For the syntax of the configuration file, see the description for the -F flag option in "Flag options syntax" on page 226.

The LANG environment variable can be set to any of the locales provided on the system.

The national language code for United States English may be En_US or C.

To determine the current setting of the national language on your system, see the output from both of the following echo commands:

- **echo $LANG**
- **echo $NLSPATH**

The LANG and NLSPATH environment variables are initialized when the operating system is installed, and may differ from the ones you want to use.

Note: You can change the default NLSPATH and PATH when you install the compiler.

Setting up a configuration file

The configuration file specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C programs. You can make entries to this file to support specific compilation requirements or to support C compilation environments.

A configuration file is a UNIX file consisting of named sections called stanzas. Each stanza contains keywords called configuration file attributes, which are assigned values. The attributes are separated from their assigned value by an equal sign. A stanza can point to a default stanza by specifying the "use" keyword. This allows specifying common attributes in a default stanza and only the deltas in a specific stanza, referred to as the local stanza.

For any of the supported attributes not found in the configuration file, the **metalc** utility uses the built-in defaults. It uses the first occurrence in the configuration file of a stanza or attribute it is looking for. Unsupported attributes, and duplicate stanzas and attributes are not diagnosed.

Notes:

1. The difference between specifying values in the stanza and relying on the defaults provided by the **metalc** utility is that the defaults provided by the **metalc** utility will not override pragmas.
2. Any entry in the configuration file must occur on a single line. You cannot continue an entry over multiple lines.

Configuration file attributes

A stanza in the configuration file can contain the following attributes:

acceptable_rc
Enables you to specify a number that represents a return code value for a program invoked by the **metalc** utility. The **metalc** utility does not place any restriction on the value assigned to the `acceptable_rc` attribute. `acceptable_rc` can appear as part of any stanza in the configuration file.

Note: If the `acceptable_rc` attribute is not specified in the configuration file, the **metalc** utility will use the value from the _C89_ACCEPTABLE_RC environment variable if it is exported, or value 4 otherwise.

as
Path name to be used for the assembler. The default is `/bin/c89`.

Note: It is recommend to use the **as** utility instead of the **metalc** utility to compile assembler source code.

asmlib
Specifies assembler macro libraries to be used when assembling the assembler source code.

asopt The list of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the `as` attribute.

asuffix
The suffix for archive files. The default is `a`.

asuffix_host
The suffix for archive data sets. The default is LIB.

ccomp The Enterprise Metal C for z/OS compiler. The default is `/usr/lpp/IBM/cjt/v3r1/metalc/exe/cjtdrvr`.

cinc A comma separated list of directories or data set wild cards used to search for C header files. For further information on the list of search places used by the compiler to search for system header files, see the note at the end of this list of configuration file attributes.

classversion
The USL class library version.

csuffix
The suffix for source programs. The default is `c` (lowercase c).

csuffix_host
The suffix for C source data sets. The default is `C` (uppercase C).

cversion
The compiler version.

exportlist_c
A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of non-XPLINK C applications. The default for this attribute is NONE.

exportlist_c_64
A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of 64-bit C applications.

isuffix The suffix for C preprocessed files. The default is `i`.

isuffix_host
The suffix for C preprocessed data sets. The default is CEX.

ilsuffix
The suffix for IPA output files. The default is `I`.

ilsuffix_host
The suffix for IPA output data sets. The default is IPA.

ld The path name to be used for the binder. The default is `/bin/c89`.

ld_c The path name to be used for the binder when only C sources appear on the command line invoked with a C stanza. The default is `/bin/c89`.

libraries
`libraries` specifies the default libraries that the binder is to use at bind time. The libraries are specified using the **-l***libname* syntax, with multiple library specifications separated by commas. The default is empty.

libraries2
: `libraries2` specifies additional libraries that the binder is to use at bind time. The libraries are specified using the **-l***libname* syntax, with multiple library specifications separated by commas. The default is empty.

options
: A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.

osuffix
: The suffix for object files. The default is `.o`.

osuffix_host
: The suffix for object data sets. The default is OBJ.

pversion
: The runtime library version.

ssuffix
: The suffix for assembler files. The default is `.s`.

ssuffix_host
: The suffix for assembler data sets. The default is ASM.

steplib
: A colon separated list of data sets or keyword NONE used to set the STEPLIB environment variable. The default is NONE, which causes the **metalc** utility to load the Enterprise Metal C for z/OS compiler from LPA or linklist.

syslib
: A colon separated list of data sets used to resolve runtime library references. Data sets from this list are used to construct the SYSLIB DD for the IPA Link and the binder invocation for non-XPLINK applications.

use
: Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the use stanza apply if no value is provided in the local, or default stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.

usuffix
: The suffix for make dependency file names. The default make dependency file name suffix is `.u`, but it is overwritten by the value assigned to this attribute.

  There is no host version of this attribute, because make dependency feature only applies to z/OS UNIX files.

xsuffix
: The suffix for definition side-deck files. The default is `x`.

xsuffix_host
: The suffix for definition side-deck data sets. The default is EXP.

Note: When using the **metalc** utility to invoke the compiler, the compiler uses the following list of search places to search for system header files:

- If the **-qnosearch** option is not specified on the command line or in the configuration file:
  1. Search places defined in the customizable defaults module (CJTEDFLT)
  2. Followed by those specified on the command line using the **-I** flag option
  3. Followed by those specified in the configuration file

- If **-qnosearch** is specified in the configuration file, it turns off all search places specified on the command line or in the default module and the only search places are those specified in the configuration file following the last **-qnosearch** option.
- If the **-qnosearch** option is specified on the command line:
  1. search places specified on the command line following the last specified **-qnosearch** option
  2. followed by those specified in the configuration file

Tailoring a configuration file

The default configuration file is installed in /usr/lpp/IBM/cjt/v3r1/metalc/etc/metalc.cfg.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C compilation environments. The **-F** option is used to specify a configuration file other than the default. For example, to make **-qnoro** the default for the **metalc** compiler invocation command, add **-qnoro** to the **metalc** stanza in your copied version of the configuration file.

You can link the compiler invocation command to several different names. The name you specify when you invoke the compiler determines which stanza of the configuration file the compiler uses. You can add other stanzas to your copy of the configuration file to customize your own compilation environment.

Only one stanza, in addition to the one referenced by the use attribute, is processed for any one invocation of the **metalc** utility. By default, the stanza that matches the command name used to invoke the **metalc** utility is used, but it can be overridden using the **-F** flag option as described in the example below.

Example: You can use the **-F** option with the compiler invocation command to make links to select additional stanzas or to specify a stanza or another configuration file:

```
metalc myfile.c -Fmyconfig:SPECIAL
```

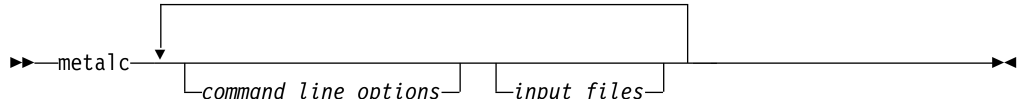
would compile myfile.c using the SPECIAL stanza in a myconfig configuration file that you had created.

Default configuration file

The default configuration file, (/usr/lpp/IBM/cjt/v3r1/metalc/etc/metalc.cfg), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C programs. You can make entries to this file to support specific compilation requirements or to support other C compilation environments. Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line. Options that do not follow this scheme are listed in "Specifying compiler options" on page 229.

Invoking the compiler

The Enterprise Metal C for z/OS compiler is invoked using the following syntax:



The parameters of the compiler invocation command can be names of input files and compiler options. Compiler options perform a wide variety of functions such as setting compiler characteristics, describing object code and compiler output to be produced, and performing some preprocessor functions.

To compile, you can use the **metalc** invocation command to produces as output, a HLASM source file `file_name.s` for each `file_name.c` input source file, unless the **-o** option was used to specify a different output file name.

Note: Any assembler file that is produced from an earlier compilation with the same name as expected output file name in this compilation is deleted as part of the compilation process, even if new output file is not produced.

Supported options

In addition to **-W** syntax for specifying keyword options, the **metalc** utility supports **-q** options syntax and several flag options.

–q options syntax

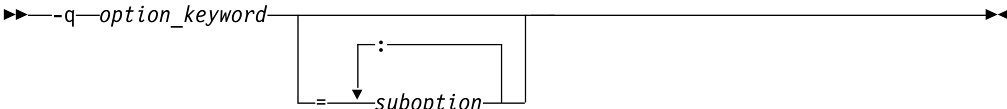
The following principles apply to the use of z/OS option names with **-q** syntax:

- Any valid abbreviation of an option in z/OS native syntax can be specified using the **-q** syntax. For example, LANGLVL can be specified as **-qlang**, **-qlangl**, **-qlanglv**, and **-qlanglvl**.
- Some options that are valid using z/OS native syntax cannot be specified using the **-q** syntax. For example, **ILP32** or **LP64** can only be specified as **-q32** or **-q64** using the **-q** syntax. Similarly, options DEFINE and UNDEFINE can be specified using -D and -U flag options.

Suboptions with negative forms of **-q** options are not supported, unless they cause an active compiler action, as in the case of **-qnokeyword=<keyword>**.

For a brief description of the compiler options that can be specified with **metalc**, type **metalc**.

The following syntax diagram shows how to specify keyword options using **-q** syntax:



In the diagram, *option_keyword* is an option name and the optional *suboption* is a value associated with the option. Keyword options with no suboptions represent switches that may be either on or off. The *option_keyword* by itself turns the switch on, and the *option_keyword* preceded by the letters NO turns the switch off. For example, **-qlist** tells the compiler to produce a listing and **-qnolist** tells the

compiler not to produce a listing. If an option that represents a switch is set more than once, the compiler uses the last setting.

Some keyword options only have values. Keywords which have values are specified as `keyword=value` pairs. In `-qfloat=ieee`, for instance, `ieee` is a value.

Some keyword options have suboptions, which in turn have values. Suboptions which have values are specified as `suboption=value` pairs. In `-qipa=level=2`, for instance, `level` is a suboption and `2` is a value.

Keyword options and suboptions may appear in mixed case letters in the command that invokes the **metalc** utility. Keyword options that have suboptions can also be preceded by the letters NO in which case they are similar to off switches and do not allow suboptions. This is a noticeable departure from the z/OS options, which allow suboptions even if they are preceded by the letters NO. However, the function that the z/OS behavior provides can easily be emulated by specifying all required suboptions with an *option_keyword* followed by the same *option_keyword* that is preceded by the letters NO. The subsequent specification of the same *option_keyword* unlocks all previously specified suboptions.

Example: NODEBUG(FORMAT(DWARF)) is equivalent to **-qdebug=format=dwarf -qnodebug**

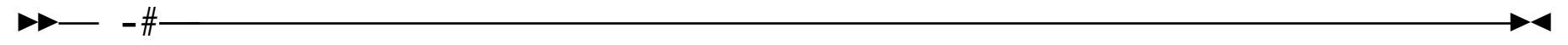
Related information

For detailed descriptions of the compiler options that can be specified with **metalc**, refer to Chapter 2, “Compiler options,” on page 7.

Flag options syntax

The **metalc** utility also supports several additional flag options, which are described below:

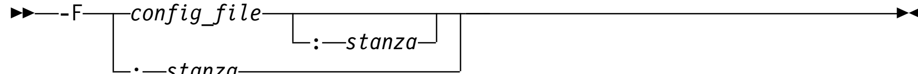
-# Displays processing options but does not invoke the compiler; the output is produced in `stderr`.



-F Names an alternative configuration file (`.cfg`) for the **metalc** utility.

Suboptions are:

- *config_file* (specifies the name of an **metalc** configuration file.)
- *stanza* (directs the compiler to use the entries under the specified *stanza* name in the *config_file* to set up the compiler environment.)



Note: The default configuration file supplied at installation time is called /usr/lpp/IBM/cjt/v3r1/metalc/etc/metalc.cfg. Any file names or stanzas that you specify on the command line override the defaults specified in the /usr/lpp/IBM/cjt/v3r1/metalc/etc/metalc.cfg configuration file.

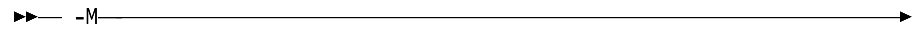
Example: To compile `myprogram.c` using a configuration file called `/usr/tmp/mycbc.cfg`, enter:

```
metalc myprogram.c -F/usr/tmp/mycbc.cfg
```

-M Instructs the compiler to generate a dependency file or dependency files that can be used by the make utility. Dependency file name can be overridden by the **-MF** option.

The compiler will generate as many dependency files as there are source files specified.

-M is the equivalent of specifying **-qmakedep** with no suboption.



Example: To compile myprogram.c and create an output file named myprogram.u, enter:

```
metalc -c -M myprogram.c
```

-MF If **-M** or **-qmakedep** is specified, this option can be used to override the default name of the dependency file.

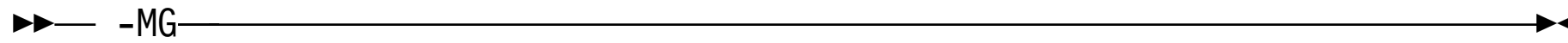
```
►►── -MF──file_name──────────────────────────────────────────►◄
```

In the syntax, *file_name* can be either a file name or a directory. By default, the dependency file name and path is the same as the **-o** compiler option but with .u suffix. The default suffix can be modified through usuffix configuration file attribute. If a directory is specified, the default dependency file name is used and placed in this directory. If a relative file name is specified, it is relative to the current working directory.

Notes:

1. The argument of *file_name* can not be the name of a data set.
2. If the file specified by **-MF** already exists, it will be overwritten. Moreover, if the output path specified does not exist or is write-protected, an error message will be issued.
3. If you specify a single file name for the **-MF** option when compiling multiple source files, each generated dependency file overwrites the previous one. Only a single output file will be generated for the last source file specified on the command line.

-MG If **-M** or **-qmakedep** is specified, this option instructs the compiler to include missing header files into the make dependencies file.



When used with **-qmakedep=pponly**, **-MG** instructs the compiler to include missing header files into the make dependencies file and suppress diagnostic messages about missing header files.

When used with **-M**, **-qmakedep**, or **-qmakedep=gcc**, **-MG** instructs the C compiler to include missing header files into the make dependency output file, but the C compiler emits only warning messages and proceeds to create an object file if the missing headers do not cause subsequent severe compile errors.

-MT If **-M** or **-qmakedep** is specified, this option sets the target to the <*target_name*> instead of the default target name. This is useful in cases

where the target is not in the same directory as the source or when the same dependency rule applies to more than one target.

```
►►— -MT—target_name——————————————————————————————————————►◄
```

When **-MT** is used with **-M** or **-qmakedep** with no suboption, all targets are repeated for each dependency.

When **-MT** is used with **-qmakedep=gcc** or **-qmakedep=pponly**, all targets appear on a single line containing all dependencies.

If the **-MT** option is specified multiple times, the targets from each specification are included in the dependency file.

-MQ **-MQ** is the same as **-MT** except that **-MQ** escapes any characters that have special meaning in make.

```
►►— -MQ—target_name——————————————————————————————————————►◄
```

The **-MQ** option is useful in cases where the target contains characters that have special meaning in make.

If the **-MQ** option is specified multiple times, the targets from each specification are included in the dependency file.

If **-MT** and **-MQ** are mixed on the command line, the targets from all **-MQ** flags will precede the targets from all **-MT** flags when they are emitted in the make dependency file.

-O Optimizes generated code.

```
►►—-O————————————————————————————————————————————————————►◄
```

-O2 Same as -O.

```
►►—-O2———————————————————————————————————————————————————►◄
```

-O3 Performs memory and compile-time intensive optimizations in addition to those executed with -O2. The -O3 specific optimizations have the potential to alter the semantics of a user's program. The compiler guards against these optimizations at -O2 and the option **-qstrict** is provided at -O3 to turn off these aggressive optimizations.

```
►►—-O3———————————————————————————————————————————————————►◄
```

-O4 Equivalent to -O3 **-qipa** and **-qhot**.

```
►►—-O4———————————————————————————————————————————————————►◄
```

-O5 Equivalent to -O3 **-qipa=level=2** and **-qhot**.

```
►►—-O5———————————————————————————————————————————————————►◄
```

-P Produces preprocessed output in a file that has a suffix that is defined by

isuffix, isuffix_host, ixxsuffix, and ixxsuffix_host. The default for host files is .CEX and for z/OS UNIX files is .i.

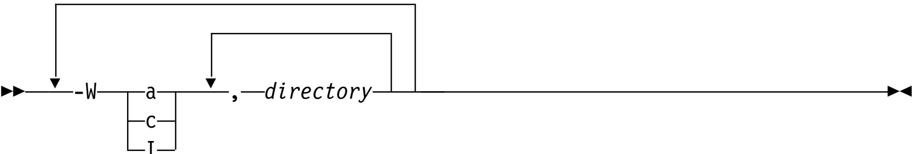
As with the **-E** option, the **-C** option can be combined with the **-P** option to preserve the comments.

-S

Accepted and ignored to allow interoperability with z/OS XL C compiler.

-W Passes the listed options to a designated compiler program where programs are:
- a (assembler)
- c (Enterprise Metal C for z/OS compiler)
- I (Interprocedural Analysis tool - compile phase)

Note: When used in the configuration file, the **-W** option requires the escape sequence back slash comma (\,) to represent a comma in the parameter string.



Specifying compiler options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of the following ways:
- On the command line
- In your source program
- In a configuration file

The compiler uses default settings for the compiler options not explicitly set by you in these listed ways. The defaults can be compiler defaults, installation defaults, or the defaults set by the **metalc** utility. The compiler defaults are overridden by installation defaults, which are overridden by the defaults set by the **metalc** utility.

When specifying compiler options, it is possible for option conflicts and incompatibilities to occur. Enterprise Metal C for z/OS resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:

```
Source  overrides    Command   overrides   Configuration    overrides   Default
 file  ----------->   line    ---------->      file        -----------> settings
```

Options that do not follow this scheme are summarized in the following table:

Table 34. Compiler option conflict resolution

Option	Conflicting Options	Resolution
-E	-o	-E
-#	-v	-#

In general, if more than one variation of the same option is specified, the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

If a command-line flag is valid for more than one compiler program (for example **-W** or **-I** applied to the compiler and assembler program names), you must specify it in `options`, or `asopt` in the configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Three exceptions to the rules of conflicting options are the **-I***directory* or `-I//dataset_name`, **-l***library*, and **-L***directory* options, which have cumulative effects when they are specified more than once.

Specifying compiler options on the command line

There are two kinds of command-line options:

- **-qoption_keyword** (compiler-specific)
- Flag options (available to Enterprise Metal C for z/OS compiler in z/OS UNIX System Service environment)

Command-line options in the **-q option_keyword** format are similar to on and off switches. For most **-q** options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, **-qsource** turns on the source option to produce a compiler listing, and **-qnosource** turns off the source option so that no source listing is produced.

Example: The following example would produce a source listing for both `MyNewProg.c` and `MyFirstProg.c` because the last source option specified (**-qsource**) takes precedence:

```
metalc -qnosource MyFirstProg.c -qsource MyNewProg.c
```

You can have multiple **-q option_keyword** instances in the same command line, but they must be separated by blanks. Option keywords can appear in mixed case, but you must specify the **-q** in lowercase.

Example: You can specify any **-q option_keyword** before or after the file name:

```
metalc -qLIST -qnomaf file.c
metalc file.c -qsource
```

Some options have suboptions. You specify these with an equal sign following the **-qoption**. If the option permits more than one suboption, a colon (:) must separate each suboption from the next.

Example: The following example compiles the C source file `file.c` using the option **-qipa** to specify the inter procedural analysis options. The suboption `level=2` tells the compiler to use the full inter procedural data flow and alias analysis, **map** tells the compiler to produce a report.

```
metalc -qipa=level=2:map file.c
```

Specifying flag options

The Enterprise Metal C for z/OS compiler uses a number of common conventional flag options. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and **-C** are two different compiler options:

- **-c** specifies that the compiler should only preprocess and compile, but not produce assembler source..
- **-C** can be used with **-E** or **-P** to specify that user comments should be preserved

Some flag options have arguments that form part of the flag. Here is an example:

```
metalc stem.c -F/home/tools/test3/new.cfg:myc -qflag=w
```

where `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string; for instance,

```
metalc -Ocv file.c
```

has the same effect as:

```
metalc -O -v -c test.c
```

Specifying compiler options in a configuration file

The default configuration file, (/usr/lpp/IBM/cjt/v3r1/metalc/etc/metalc.cfg), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C programs. You can make entries to this file to support specific compilation requirements or to support other C compilation environments.

Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line.

Specifying compiler options in your program source files

You can specify compiler options within your program source by using `#pragma` directives. Options specified with pragma directives in program source files override all other option settings for most options.

Specifying compiler options for architecture-specific 32-bit or 64-bit compilation

You can use Enterprise Metal C for z/OS compiler options to optimize compiler output for use on specific processor architectures. You can also instruct the compiler to compile in either 32-bit or 64-bit mode.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1. Compiler default (32-bit mode)
2. Configuration file settings
3. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)
4. Source file statements (**#pragma options(ARCH(suboption),TUNE(suboption))**)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options, subject to the following conditions:

- Compiler mode is set according to the last-found instance of the **-q32**, or **-q64** compiler options. If neither of these compiler options is chosen, the compiler mode is set to 32-bit.
- Architecture target is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the compiler mode setting. If the **-qarch** option is not set, the compiler assumes a **-qarch** setting of 5.

- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the architecture target and compiler mode settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use.

Possible option conflicts and compiler resolution of these conflicts are described below:

- **-q32** or **-q64** setting is incompatible with user-selected **-qarch** option.

  Resolution: **-q32** or **-q64** setting overrides **-qarch** option; compiler issues a warning message, sets **-qarch** to 10, and sets the **-qtune** option to the **-qarch** setting's default **-qtune** value.
- **-q32** or **-q64** setting is incompatible with user-selected **-qtune** option.

  Resolution: **-q32** or **-q64** setting overrides **-qtune** option; compiler issues a warning message, and sets **-qtune** to the **-qarch** settings's default **-qtune** value.
- **-qarch** option is incompatible with user-selected **-qtune** option.

  Resolution: Compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.
- Selected **-qarch** and **-qtune** options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-qarch** to 10, and sets **-qtune** to the **-qarch** setting's default **-qtune** setting. The compiler mode (32 or 64-bit) is determined by the **-q32** or **-q64** compiler settings.

Appendix. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact z/OS web page (www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that the screen reader is set to read out

© Copyright IBM Corp. 2018

punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the

default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included
The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Glossary

This glossary defines technical terms and abbreviations that are used in Enterprise Metal C for z/OS documentation. If you do not find the term you are looking for, refer to the index of the appropriate Enterprise Metal C for z/OS manual or view the IBM Glossary of Computing Terms (www.ibm.com/software/globalization/terminology).

The following cross-references are used in this glossary:

- See refers you from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
- See also refers you to a related or contrasting term.

To view glossaries for other IBM products, go to IBM Glossary of Computing Terms (www.ibm.com/software/globalization/terminology).

A

abstract data type
A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

access mode

1. The manner in which files are referred to by a computer. See also dynamic access, sequential access.
2. A form of access permitted for a file.

access specifier
A specifier that defines whether a class member is accessible in an expression or declaration. The three access specifiers are public, private, and protected.

addressing mode (AMODE)
The attribute of a program module that identifies the addressing range in which the program entry point can receive control.

address space
The range of addresses available to a computer program or process. Address space can refer to physical storage, virtual storage, or both.

aggregate

1. A structured collection of data objects that form a data type.

alert

1. A message or other indication that signals an event or an impending event.
2. To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred.

alert character
A character that in the output stream causes a terminal to alert its user by way of a visual or audible notification. The alert character is the character designated by a '\a' in the C language. It is unspecified whether this

character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

alias

1. An alternative name for an integrated catalog facility (ICF) user catalog, a file that is not a Virtual Storage Access Method (VSAM) file, or a member of a partitioned data set (PDS) or a partitioned data set extended (PDSE).
2. An alternative name used instead of a primary name.

aliasing
A compilation process that attempts to determine what aliases exist, so that optimization does not result in incorrect program results.

alignment
The storing of data in relation to certain machine-dependent boundaries.

alternate code point
A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standards Institute (ANSI)
A private, nonprofit organization whose membership includes private companies, U.S. government agencies, and professional, technical, trade, labor, and consumer organizations. ANSI coordinates the development of voluntary consensus standards in the U.S.

American Standard Code for Information Interchange (ASCII)
A standard code used for information exchange among data processing systems, data communication systems, and associated equipment. ASCII uses a coded character set consisting of 7-bit coded characters. See also Extended Binary Coded Decimal Interchange Code.

AMODE
See addressing mode.

angle bracket
Either the left angle bracket (<) or the right angle bracket (>). In the portable character set, these characters are referred to by the names <less-than-sign> and <greater-than-sign>.

anonymous union
An unnamed object whose type is an unnamed union.

ANSI See American National Standards Institute.

AP See application program.

API See application programming interface.

application
One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application generator
An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program (AP)
A complete, self-contained program, such as a text editor or a web

browser, that performs a specific task for the user, in contrast to system software, such as the operating system kernel, server processes, and program libraries.

application programming interface (API)
An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

archive library
A facility for grouping application-program object files. The archive library file, when created for application-program object files, has a special symbol table for members that are object files.

argument
A value passed to or returned from a function or procedure at run time.

argument declaration
See also parameter declaration.

arithmetic object
An integral object or objects having the float, double, or long double type.

array In programming languages, an aggregate that consists of data objects, with identical attributes, each of which can be uniquely referenced by subscripting. See also scalar.

array element
One of the data items in an array.

ASCII See American Standard Code for Information Interchange.

assembler
A computer program that converts assembly language instructions into object code.

Assembler H
An IBM licensed program that translates symbolic assembler language into binary machine language.

assembler user exit
A routine to tailor the characteristics of an enclave prior to its establishment.

assembly language
A symbolic programming language that represents machine instructions of a specific architecture.

assignment expression
An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand.

automatic call library
A group of modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed. The automatic call library can contain: object modules, with or without binder control statements; load modules; and runtime routines.

automatic library call
The process by which the binder resolves external references by including additional members from the automatic call library.

automatic storage
: Storage that is allocated on entry to a routine or block and is freed when control is returned. See also dynamic storage.

auto storage class specifier
: A specifier that enables the programmer to define a variable with automatic storage; its scope is restricted to the current block.

B

background process
: A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. See also foreground process.

background processing
: A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backslash
: The character \. The backslash enables a user to escape the special meaning of a character. That is, typing a backslash before a character tells the system to ignore any special meaning the character might have.

binary expression
: An expression containing two operands and one operator.

binary stream
: A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams.

binder

1. The z/OS program that processes the output of language translators and compilers into an executable program (a load module or program object). The binder replaces the linkage editor and batch loader. See also prelinker.
2. See linkage editor.

bit field
: A member of a structure or union that contains 1 or more named bits.

bitwise operator
: An operator that manipulates the value of an object at the bit level.

blank character

1. One of the characters that belong to the blank character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character.
2. A graphic representation of the space character.

block

1. A string of data elements recorded, processed, or transmitted as a unit. The elements can be characters, words, or physical records.
2. In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes.

block statement
: In the C language, a group of data definitions, declarations, and statements that are located between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single, C-language statement.

boundary alignment
: The position in main storage of a fixed-length field, such as halfword or doubleword, which is aligned on an integral boundary for that unit of information. For example, a word boundary alignment stores the object in a storage address evenly divisible by four.

brace
: Either of the characters left brace ({) and right brace (}). When an object is enclosed in braces, the left brace immediately precedes the object and the right brace immediately follows it.

bracket
: Either of the characters left bracket ([) and right bracket (]).

break statement
: A C control statement that contains the keyword break and a semicolon (;). It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in
: In programming languages, pertaining to a language object that is defined in the programming language specification.

built-in function
: A function that is predefined by the compiler and whose code is incorporated directly into the compiled object rather than called at run time. See also function.

byte-oriented stream
: A byte-oriented stream refers to a stream which only single byte input/output is allowed.

C

callable service
: A program service provided through a programming interface.

call chain
: A trace of all active routines and subroutines, such as the names of routines and the locations of save areas, that can be constructed from information included in a system dump.

caller
: A function that calls another function.

cancelability point
: A specific point within the current thread that is enabled to solicit cancel requests.

carriage return character
: A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred.

case clause
: In a C switch statement, a CASE label followed by any number of statements.

case label
: The word case followed by a constant expression and a colon. When the selector is evaluated to the value of the constant expression, the statements following the case label are processed.

cast expression
: An expression that converts or reinterprets its operand.

cast operator
: An operator that is used for explicit type conversions.

cataloged procedure
: A set of job control language (JCL) statements that has been placed in a library and that is retrievable by name.

catch block
: A block associated with a try block that receives control when an exception matching its argument is thrown. See also try block.

CCS See coded character set.

character

1. A sequence of one or more bytes representing a single graphic symbol or control code.
2. In a computer system, a member of a set of elements that is used for the representation, organization, or control of data.

character class
: A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale.

character constant
: The actual character value (a symbol, quantity, or constant) in a source program that is itself data, instead of a reference to a field that contains the data.

character set
: A defined set of characters with no coded representation assumed that can be recognized by a configured hardware or software system. A character set can be defined by alphabet, language, script, or any combination of these items.

character special file
: An interface file that provides access to an input or output device, which uses character I/0 instead of block I/0.

character string
: A contiguous sequence of characters terminated by and including the first null byte.

child A node that is subordinate to another node in a tree structure. Only the root node is not a child.

child enclave
: The nested enclave created as a result of certain commands being issued from a parent enclave. See also nested enclave, parent enclave.

child process
: A process that is created by a parent process and that shares the resources of the parent process to carry out a request.

C language
: A language used to develop application programs in compact, efficient code that can be run on different types of computers with minimal change.

C library
: A system library that contains common C language subroutines for file access, string operations, character operations, memory allocation, and other functions.

CLIST See command list.

COBOL
: See Common Business Oriented Language.

coded character set (CCS)
: A set of unambiguous rules that establishes a character set and the one-to-one relationships between the characters of the set and their coded representations.

code element set
: The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. X/Open.

code generator
: The part of the compiler that physically generates the object code.

code page
: A particular assignment of code points to graphic characters. Within a given code page, a code point can have only one specific meaning. A code page also identifies how undefined code points are handled. See also code point.

code point

1. An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.
2. A unique bit pattern that represents a character in a code page. See also code page.

collating element
: The smallest entity used to determine the logical ordering of strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. See also collating sequence.

collating sequence
: The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order.

collation
: The logical ordering of characters and strings according to defined rules.

collection
: An abstract class without any ordering, element properties, or key properties.

Collection Class Library
: A complete set of abstract data structure such as trees, stacks, queues, and linked lists.

column position
: A unit of horizontal measure related to characters in a line. It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes). The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. X/Open.

comma expression
: An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. If the left operand produces a value, the compiler discards this value.

command
: A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command list (CLIST)
: A language for performing TSO tasks.

COMMAREA
: See communication area.

Common Business Oriented Language (COBOL)
: A high-level programming language that is used primarily for commercial data processing.

compilation unit
: A portion of a computer program sufficiently complete to be compiled correctly.

compiler option
: A keyword that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, and the destination of error messages.

condition

1. An expression that can be evaluated as true, false, or unknown. It can be expressed in natural language text, in mathematically formal notation, or in a machine-readable language.
2. An exception that has been enabled, or recognized, by the Language Environment and thus is eligible to activate user and language condition handlers. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

conditional expression
A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

condition handler
A user-written routine or language-specific routine (such as a PL/ION-unit or C signal() function call) invoked by the Language Environment condition manager to respond to conditions.

condition manager
The condition manager is the part of the common execution environment that manages conditions by invoking various user-written and language-specific condition handlers.

condition token
In Language Environment, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

constant
A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants.

constant expression
An expression that has a value that can be determined during compilation and that does not change during the running of the program.

constant propagation
An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy
The attribute of applications that contain external data and require additional processing to make them reentrant. See also natural reentrancy.

control character
A character whose occurrence in a particular context initiates, modifies, or stops a control function.

controlling process
A session leader that has control of a terminal.

controlling terminal
The active workstation from which the process group for that process was started. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session.

control section (CSECT)
The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

control statement
In programming languages, a statement that is used to interrupt the continuous sequential processing of programming statements. Conditional statements such as IF, PAUSE, and STOP are examples of control statements.

conversion

1. In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types.
2. The process of changing from one form of representation to another. Changing a code point that is assigned to a character in one code page to its corresponding code point in another code page is an example of conversion.

Coordinated Universal Time (UTC)
The international standard of time that is kept by atomic clocks around the world.

cross-compiler
A compiler that produces executable files that run on a platform other than the one on which the compiler is installed.

CSECT
See control section.

current working directory
See working directory.

cursor A reference to an element at a specific position in a data structure.

D

data abstraction
A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations.

data definition (DD)
A program statement that describes the features of, specifies relationships of, or establishes the context of data. A data definition reserves storage and can provide an initial value.

data definition name (ddname)
The name of a data definition (DD) statement that corresponds to a data control block that contains the same name.

data definition statement (DD statement)
A job control statement that is used to define a data set for use by a batch job step, started task or job, or an online user.

data member
The smallest possible piece of complete data. Elements are composed of data members.

data object
An element of data structure such as a file, an array, or an operand that is needed for the execution of an application.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data stream
The commands, control codes, data, or structured fields that are transmitted between an application program and a device such as printer or nonprogrammable display station.

data structure
In Open Source Initiative (OSI), the syntactic structure of symbolic expressions and their storage allocation characteristics.

data type
A category that identifies the mathematical qualities and internal representation of data and functions.

Data Window Services (DWS)
Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as TEMPSPACE.

DBCS See double-byte character set.

DCT See destination control table.

DD See data definition.

ddname
See data definition name.

DD statement
See data definition statement.

dead code
Code that is never referenced, or that is always branched over.

dead store
A store into a memory location that will later be overwritten by another store without any intervening loads. In this case, the earlier store can be deleted.

decimal constant
A numerical data type used in standard arithmetic operations. Decimal constants can contain any digits 0 through 9. See also integer constant.

decimal overflow
A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration
1. In the C language, a description that makes an external object or function available to a function or a block statement.
2. A statement that establishes the names and characteristics of data objects and functions used in a program.

default clause
In the C languages, within a switch statement, the keyword default followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen.

default initialization
The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. In C language, external and static

variables receive a default initialization of zero, while the default initialization for auto and register variables is undefined.

definition
A declaration that reserves storage and can provide an initial value for a data object or define a function.

degree
The number of children of a node.

dereference
In the C language, to apply the unary operator * to a pointer to access the object the pointer points to. See also indirection.

descriptor
A PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

device
A piece of equipment such as a workstation, printer, disk drive, tape unit, or remote system.

difference
Given two sets A and B, the set of all elements contained in A but not in B (A-B).

digraph
A combination of two keystrokes used to represent unavailable characters in a C source program. Digraphs are read as tokens during the preprocessor phase.

directive
A control statement that directs the operation of a feature and is recognized by a preprocessor or other tool. See also pragma.

directory

1. The part of a partitioned data set that describes the members in the data set.
2. In a hierarchical file system, a grouping of related files.

display
To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined.

do statement
A looping statement that contains the keyword do, followed by a statement (the action), the keyword while, and an expression in parentheses (the condition).

dot
A symbol (.) that indicates the current directory in a relative path name. See also period.

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. These character sets are commonly used by national languages, such as Japanese and Chinese, that have more symbols than can be represented by a single byte. See also single-byte character set.

double-precision
Pertaining to the use of two computer words to represent a number in accordance with the required precision.

doubleword
A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. See also halfword, word.

DSA See dynamic storage area.

DWS See Data Window Services.

dynamic
Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

dynamic access
A process where records can be accessed sequentially or randomly, depending on the form of the input/output request. See also access mode.

dynamic allocation
Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage.

dynamic binding
The act of resolving references to external variables and functions at run time.

dynamic storage
An area of storage that is explicitly allocated by a program or procedure while it is running. See also automatic storage.

dynamic storage area (DSA)
A type of storage allocation in which storage is assigned to a program or application at run time.

E

EBCDIC
See Extended Binary Coded Decimal Interchange Code.

effective group ID
An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime.

element
The smallest unit in a table, array, list, set, or other structure. Examples of an element are a value in a list of values and a data field in an array.

element equality
A relation that determines if two elements are equal.

element occurrence
A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value
All the instances of an element with a particular value in a collection. In a non-unique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause
The part of an if statement that contains the keyword 'else' followed by a

statement. The else clause provides an action that is started when the if condition evaluates to a value of 0 (false).

empty line
A line consisting of only a newline character. X/Open.

empty string
A character array whose first element is a null character.

encapsulation
In object-oriented programming, the technique that is used to hide the inherent details of an object, function, or class from client programs.

entry point
The address or label of the first instruction processed or entered in a program, routine, or subroutine.There might be a number of different entry points, each corresponding to a different function or purpose.

enum constant
See enumeration constant.

enumeration constant (enum constant)
In the C language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed.

enumeration data type
A data type that represents a set of values that a user defines.

enumeration tag
The identifier that names an enumeration data type.

enumeration type
A data type that defines a set of enumeration constants.

enumerator
An enumeration constant and its associated value.

equivalence class
A grouping of characters or character strings that are considered equal for purposes of collation. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation.

escape sequence
A string of bit combinations that is used to escape from normal data, such as text code points, into control information.

exception
A condition or event that cannot be handled by a normal process.

executable file
A file that contains programs or commands that perform operations on actions to be taken.

executable program
A program in a form suitable for execution by a computer. The program can be an application or a shell script.

Extended Binary Coded Decimal Interchange Code (EBCDIC)
A coded character set of 256 8-bit characters developed for the representation of textual data. See also American Standard Code for Information Interchange.

extended-precision
: Pertains to the use of more than two computer words to represent a floating point number in accordance with the required precision. For example, in z/OS, four computer words are used for an extended-precision number.

extension
: An element or function not included in the standard language.

F

FIFO special file
: A type of file with the property that data written to such a file is read on a first-in-first-out (FIFO) basis.

file descriptor
: A positive integer or a data structure that uniquely identifies an open file for the purpose of file access.

file mode
: An object containing the file permission bits and other characteristics of a file.

file permission bit
: Information about a file that is used, along with other information, to determine whether a process has read, write, or execute permission to a file. The use of file permission bits is described in file access permissions.

file scope
: A property of a file name that is declared outside all blocks, classes, and function declarations and that can be used after the point of declaration in a source file.

filter
: A command that reads standard input data, modifies the data, and sends it to standard output. A pipeline usually has several filters.

flat collection
: A collection that has no hierarchical structure.

float constant

1. A constant representing a non-integral number.
2. A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an "e" or "E," an optional sign (+ or -), and one or more digits (0 through 9).

footprint
: The amount of computer storage that is occupied by a computer program. For example, if a program occupies a large amount of storage, it has a large footprint.

foreground process
: A process that must be completed before another command is issued. See also background process.

foreground process group
: A group whose member processes have privileges that are denied to background processes when the controlling terminal is being accessed. Each controlling terminal can have only one foreground process group.

form-feed character
: A character in the output stream that indicates that printing should start

on the next page of an output device. The form-feed character is designated by '\f' in the C language. If the form-feed character is not the first character of an output line, the result is unspecified. X/Open.

for statement
A looping statement that contains the word for followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon, which cannot be omitted.

forward declaration
A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

freestanding application

1. An application that is created to run without the run-time environment or library with which it was developed.
2. An application that does not use the services of the dynamic run-time library or of the Language Environment. Under z/OS C support, this ability is a feature of the System Programming C support.

free store
Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

function
A named group of statements that can be called and evaluated and can return a value to the calling statement. See also built-in function.

function call
An expression that transfers the path of execution from the current function to a specified function (the called function). A function call contains the name of the function to which control is transferred and a parenthesized list of values.

function declarator
The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition
The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype
A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a semicolon (;). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function scope
Labels that are declared in a function have function scope and can be used anywhere in that function after their declaration.

G

GCC See GNU Compiler Collection.

GDDM
: See Graphical Data Display Manager.

Generalized Object File Format (GOFF)
: This object module format extends the capabilities of object modules so that they can contain more information. It is required for XPLINK.

global
: Pertaining to information available to more than one program or subroutine. See also local.

global variable
: A symbol defined in one program module that is used in other program modules that are independently compiled.

GMT
: See Greenwich mean time.

GNU Compiler Collection (GCC)
: An open source collection of compilers supporting C, C++, Objective-C, Ada, Java, and Fortran.

GOFF
: See Generalized Object File Format.

Graphical Data Display Manager (GDDM)
: An IBM computer-graphics system that defines and displays text and graphics for output on a display or printer.

graphic character
: A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying.

Greenwich mean time (GMT)
: The mean solar time at the meridian of Greenwich, England.

H

halfword
: A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit. See also doubleword, word.

hash function
: A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table

1. A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in.
2. A table of information that is accessed by way of a shortened search key (the hash value). The use of a hash table minimizes average search time.

header file
: See include file.

heap storage
: An area of storage used for allocation of storage that has a lifetime that is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant
A constant, usually starting with special characters, that contains only hexadecimal digits.

High Level Assembler
An IBM licensed program that translates symbolic assembler language into binary machine language.

hiperspace memory file
A type of file that is stored in a single buffer in an address space, with the rest of the data being kept in a hiperspace. In contrast, for regular files, all the file data is stored in a single address space.

hook A location in a compiled program where the compiler has inserted an instruction that allows programmers to interrupt the program (by setting breakpoints) for debugging purposes.

hybrid code
Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using iconv().

I

ID See identifier.

identifier (ID)
One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element.

if statement
A conditional statement that specifies a condition to be tested and the action to be taken if the condition is satisfied.

ILC

1. See interlanguage communication.
2. See interlanguage call.

implementation-defined
Pertaining to behavior that is defined by the compiler rather than by a language standard. Programs that rely on implementation-defined behavior may behave differently when compiled with different compilers. See also undefined behavior.

IMS™ See Information Management System.

include directive
A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file
A text file that contains declarations that are used by a group of functions, programs, or users.

incomplete type
A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures, and unions of unspecified content.

indirection

1. A mechanism for connecting objects by storing, in one object, a reference to another object. See also dereference.
2. In the C language, the application of the unary operator * to a pointer to access the object to which the pointer points.

induction variable
A controlling variable of a loop.

Information Management System (IMS)
Any of several system environments that have a database manager and transaction processing that can manage complex databases and terminal networks.

initial heap
A heap that is controlled by the HEAP run-time option and designated by a heap_id of 0.

initializer
An expression used to initialize data objects.

inline To replace a function call with a copy of the function's code during compilation.

inline function
A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword inline. Both member and non-member functions can be inlined.

input stream
A sequence of control statements and data submitted to an operating system by an input device.

instruction
A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

instruction scheduling
An optimization technique that reorders instructions in code to minimize execution time.

integer constant
A decimal, octal, or hexadecimal constant. See also decimal constant.

integral object
A character object, an object having an enumeration type, an object having variations of the type int, or an object that is a bit field.

Interactive System Productivity Facility (ISPF)
An IBM licensed program that serves as a full-screen editor and dialog manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogs between the application programmer and the terminal user.

interlanguage call (ILC)
A call to a procedure or function made by a program written in one language to a procedure or function coded in a different language.

interlanguage communication (ILC)
: The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

interoperability
: The ability of a computer or program to work with other computers or programs.

interprocess communication (IPC)
: The process by which programs send messages to each other. Sockets, semaphores, signals, and internal message queues are common methods of interprocess communication.

IPC See interprocess communication.

ISPF See Interactive System Productivity Facility.

iteration
: The repetition of a set of computer instructions until a condition is satisfied.

J

JCL See job control language.

job control language (JCL)
: A command language that identifies a job to an operating system and describes the job requirements.

K

kernel The part of an operating system that contains programs for such tasks as input/output, management and control of hardware, and the scheduling of user tasks.

keyword

1. One of the predefined words of a programming language, artificial language, application, or command. See also operand, parameter.
2. A symbol that identifies a parameter in job control language (JCL).

L

label An identifier within or attached to a set of data elements.

Language Environment
: An element of z/OS that provides a common runtime environment and common runtime services for C/C++, COBOL, PL/I, and Fortran applications.

last element
: The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

leaf In a tree, an entry or node that has no children.

library

1. A collection of model elements, including business items, processes, tasks, resources, and organizations.

2. A set of object modules that can be specified in a link command.

linkage
Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a non-member function declared with the static keyword. All other functions have external linkage.

linkage editor
A computer program for creating load modules from one or more object modules or load modules by resolving cross-references among the modules and, if necessary, adjusting addresses.

linker A program that resolves cross-references among separately compiled object modules and then assigns final addresses to create a single executable program.

link pack area (LPA)
The portion of virtual storage below 16 MB that contains frequently used modules.

literal A symbol or a quantity in a source program that is itself data, rather than a reference to data.

loader A program that copies an executable file into main storage so that the file can be run.

load module
A program in a form suitable for loading into main storage for execution.

local

1. Pertaining to information that is defined and used only in one subdivision of a computer program. See also global.
2. In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block.

local custom
A convention of a geographical area or territory for such things as date, time, and currency formats. X/Open.

locale A setting that identifies language or geography and determines formatting conventions such as collation, case conversion, character classification, the language of messages, date and time representation, and numeric representation.

local scope
A name declared in a block that has local scope and can only be used in that block.

loop unrolling
An optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

LPA See link pack area.

lvalue An expression that represents a data object that can be viewed, tested, and changed. An lvalue is usually the left operand in an assignment expression.

M

macro
An instruction that causes the execution of a predefined sequence of instructions.

macro call
See macro.

main function
A function that has the identifier main. Each program must have exactly one function named main. The main function is the first user function that receives control when a program starts to run.

makefile
In UNIX, a text file containing a list of an application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility
A utility that maintains all of the parts and dependencies for an application. The make utility uses a makefile to keep the parts of a program synchronized. If one part of an application changes, the make utility updates all other files that depend on the changed part.

manipulator
A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

method
See member function.

method file

1. For ASCII locales, a file that defines the method functions to be used by C runtime locale-sensitive interfaces. A method file also identifies where the method functions can be found. IBM supplies several method files used to create its standard set of ASCII locales. Other method files can be created to support customized or user-created code pages. Such customized method files replace IBM-supplied charmap method functions with user-written functions.
2. A file that allows users to indicate to the localedef utility where to look for user-provided methods for processing user-designed code pages.

migrate
To install a new version or release of a program to replace an earlier version or release.

module
A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.

multibyte character
A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multibyte control
See escape sequence.

multicharacter collating element
A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements ch and ll. X/Open.

multiprocessor
: A processor complex that has more than one central processor.

multitasking
: A mode of operation in which two or more tasks can be performed at the same time.

mutex
: See mutual exclusion.

mutex attribute object
: A type of attribute object with which a user can manage mutual exclusion (mutex) characteristics by defining a set of variables to be used during its creation. A mutex attribute object eliminates the need to redefine the same set of characteristics for each mutex object created. See also mutual exclusion.

mutex object
: An identifier for a mutual exclusion (mutex).

mutual exclusion (mutex)
: A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. See also mutex attribute object.

N

namespace
: A category used to group similar types of identifiers.

natural reentrancy
: The attribute of applications that contain no static external data and do not require additional processing to make them reentrant. See also constructed reentrancy.

nested enclave
: A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also child enclave, parent enclave.

newline character (NL)
: A control character that causes the print or display position to move down one line.

nickname
: See alias.

NL
: See newline character.

nonprinting character
: See control character.

NUL
: See null character.

null character (NUL)
: A control character with the value of X'00' that represents the absence of a displayed or printed character.

null pointer
: The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0. The C language guarantees that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

null statement
: A statement that consists of a semicolon.

null string
: A character or bit string with a length of zero.

null value
: A parameter position for which no value is specified.

null wide-character code
: A wide-character code with all bits set to zero.

number sign
: The character #, which is also referred to as the hash sign.

O

object

1. A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. See also instance.
2. In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data.

object module
: A set of instructions in machine language that is produced by a compiler or assembler from a subroutine or source module and can be input to the linking program. The object module consists of object code.

octal constant
: The digit 0 (zero) followed by any digits 0 through 7.

open file
: A file that is currently associated with a file descriptor.

operand
: An entity on which an operation is performed.

operating system (OS)
: A collection of system programs that control the overall operation of a computer system.

operator precedence
: In programming languages, an order relationship that defines the sequence of the application of operators with an expression.

orientation
: The orientation of a stream refers to the type of data which may pass through the stream. A stream without orientation is one on which no stream I/O has been performed.

OS See operating system.

overflow
: The condition that occurs when data cannot fit in the designated field.

overlay
: The technique of repeatedly using the same areas of internal storage during different stages of a program. Unions are used to accomplish this in C.

P

parameter (parm)
A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process. See also keyword, operand.

parameter declaration
The description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value. See also argument declaration.

parent enclave
The enclave that issues a call to system services or language constructs to create a nested (or child) enclave. See also child enclave, nested enclave.

parent process
A process that is created to carry out a request or set of requests. The parent process, in turn, can create child processes to process requests for the parent.

parent process ID (PPID)
An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process.

parm See parameter.

partitioned concatenation
The allocation of partitioned data sets (PDSs), partitioned data sets extended (PDSEs), UNIX file directories, or any combination of these such that the basic partitioned access method (BPAM) retrieves them as a single data set.

partitioned data set (PDS)
A data set on direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. See also sequential data set.

partitioned data set extended (PDSE)
A data set that contains an indexed directory and members that are similar to the directory and members of partitioned data sets (PDSs). See also library.

path name
A name that specifies all directories leading to a file plus the file name itself.

path name resolution
The process of resolving a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. X/Open.

pattern
A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical.

PDS See partitioned data set.

PDSE See partitioned data set extended.

period The symbol ".". The term dot is used for the same symbol when referring to a web address or file extension. This character is named <period> in the portable character set. See also dot.

permission
The ability to access a protected object, such as a file or directory. The number and meaning of permissions for an object are defined by the access control list.

persistent environment
An environment that once created by the user may be used repeatedly without incurring the overhead of initialization and termination for each call. The environment remains available until explicitly terminated by the user.

PGID See process group ID.

PID See process ID.

platform
The combination of an operating system and hardware that makes up the operating environment in which a program runs.

pointer
A data element or variable that holds the address of a data object or a function. See also scalar.

portability
1. The ability of a program to run on more than one type of computer system without modification.
2. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

portable character set
A set of characters, specified in POSIX 1003.2, section 4, that must be supported by conforming implementations.

portable file name character set
The set of characters from which portable file names must be constructed to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945.

positional parameter
A parameter that must appear in a specified location, relative to other parameters.

PPID See parent process ID.

pragma
A standardized form of comment which has meaning to a compiler. A pragma usually conveys non-essential information, often intended to help the compiler to optimize the program. See also directive.

precedence
The priority system for grouping different types of operators with their operands.

predefined macro
An identifier predefined by the compiler, which will be expanded by the preprocessor during compilation.

preinitialization
A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker
A utility that preprocesses an object for certain programs. See also binder.

preprocessor
A routine that performs initial processing and translation of source code or data prior to compiling the source code or processing the data in another program such as an emulator.

preprocessor directive
In the C language, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation.

preprocessor statement
In the C language, a statement that begins with the symbol # and contains instructions that the preprocessor can interpret.

primary expression

1. Literals, names, and names qualified by the :: (scope resolution) operator.
2. Any of the following types of expressions: a) identifiers, b) parenthesized expressions, c) function calls, d) array element specifications, e) structure member specifications, or f) union member specifications.

process

1. An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is known as the parent process, and the new process created by the fork() function is known as the child process.
2. An instance of a program running on a system and the resources that it uses.

process group
A collection of processes in a system that is identified by a process group ID.

process group ID (PGID)
The unique identifier representing a process group during its lifetime. A process group ID is a positive integer that is not reused by the system until the process group lifetime ends.

process group lifetime
A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group because either it is the end of the last process' lifetime or the last remaining process is calling the setsid() or setpgid() functions. X/Open. ISO.1.

process ID (PID)
The unique identifier that represents a process. A process ID is a positive integer and is not reused until the process lifetime ends.

process lifetime
The period of time that begins when a process is created and ends when the process ID is returned to the system. X/Open. ISO.1. After a process is

created with a fork() function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a wait() or waitpid() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

profiling
A performance analysis process that is based on statistics for the resources that are used by a program or application.

program object
All or part of a computer program in a form suitable for loading into virtual storage for execution. Program objects are stored in partitioned data set extended (PDSE) program libraries and have fewer restrictions than load modules. Program objects are produced by the binder.

program unit
See compilation unit.

prototype
A function declaration or definition that includes both the return type of the function and the types of its parameters.

Q

QMF™ See Query Management Facility™.

qualified name

1. A data set name consisting of a string of names separated by periods; for example, TREE.FRUIT.APPLE is a qualified name.

qualified type name
A name used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF)
An IBM query and report writing facility that supports a variety of tasks such as data entry, query building, administration, and report analysis.

queue A data structure for processing work in which the first element added to the queue is the first element processed. This order is referred to as first-in first-out (FIFO).

quotation mark
The characters " and '.

R

radix character
The character that separates the integer part of a number from the fractional part. X/Open .

random access
An access mode in which records can be referred to, read from, written to, or removed from a file in any order.

real group ID
: The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. This value is subject to change during the process lifetime.

real user ID
: The attribute of a process that, at the time a process is created, identifies the user who created the process.

reason code
: A value used to indicate the specific reason for an event or condition.

reassociation
: An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection
: In a shell, a method of associating files with the input or output of commands.

reentrant
: The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

refresh
: To ensure that the information on the user's terminal screen is up-to-date.

register variable
: A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression

1. A set of characters, meta characters, and operators that define a string or group of strings in a search pattern.
2. A string containing wildcard characters and operations that define a set of one or more possible strings.
3. A mechanism for selecting specific strings from a set of character strings.

regular file
: A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1]

relation
: An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name
: A string of characters that is used to refer to an object and that starts at some point in the directory hierarchy other than the root. The starting point is frequently a user's current directory.

reserved word
: A word that is defined by a programming language and that cannot be used as an identifier or changed by the user.

residency mode (RMODE)
: In z/OS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

reverse solidus

RMODE
See residency mode.

runtime environment
A set of resources that are used to run a program or process.

runtime library
A compiled collection of functions whose members can be referred to by an application program at run time.

S

SBCS See single-byte character set.

scalar An arithmetic object, an enumerated object, or a pointer to an object.

scope A part of a source program in which an object is defined and recognized.

SDK See software development kit.

semaphore
An object used by multi-threaded applications for signaling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence
A sequentially ordered flat collection.

sequential access
The process of referring to records one after another in the order in which they appear on the file. See also access mode.

sequential concatenation
The allocation of sequential data sets, partitioned data set (PDS) members, partitioned data set extended (PDSE) members, UNIX files, or any combination of these such that the system retrieves them as a single, sequential, data set.

sequential data set
A data set whose records are organized based on their successive physical positions, such as on magnetic tape. See also partitioned data set.

session
A collection of process groups established for job control purposes.

shell A software interface between users and an operating system. Shells generally fall into one of two categories: a command line shell, which provides a command line interface to the operating system; and a graphical shell, which provides a graphical user interface (GUI).

signal

1. A mechanism by which a process can be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes.
2. In operating system operations, a method of inter-process communication that simulates software interrupts.
3. A condition that might or might not be reported during program execution. For example, a signal can represent erroneous arithmetic operations, such as division by zero.

signal handler
A subroutine or function that is called when a signal occurs.

single-byte character set (SBCS)
A coded character set in which each character is represented by a 1-byte code. A 1-byte code point allows representation of up to 256 characters. See also double-byte character set.

single precision
The use of one computer word to represent a number, in accordance with the required precision.

slash The character /, also known as forward slash. This character is named <slash> in the portable character set.

socket In the Network Computing System (NCS), a port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address.

software development kit (SDK)
A set of tools, APIs, and documentation to assist with the development of software in a specific computer language or for a particular operating environment.

sorted map
A sorted flat collection with key and element equality.

sorted relation
A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set
A sorted flat collection with element equality.

source module
See source program.

source program
A set of instructions that are written in a programming language and must be translated into machine language before the program can be run.

space character
In the portable character set, the <space> character.

spanned record
A logical record stored in more than one block on a storage medium.

spill area
A storage area that is used to save the contents of registers.

square bracket
See bracket.

stack frame
See dynamic storage area.

standard error (STDERR)
The output stream to which error messages or diagnostic messages are sent. See also standard input, standard output.

standard input (STDIN)
An input stream from which data is retrieved. Standard input is normally

associated with the keyboard, but if redirection or piping is used, the standard input can be a file or the output from a command. See also standard error.

standard output (STDOUT)
The output stream to which data is directed. Standard output is normally associated with the console, but if redirection or piping is used, the standard output can be a file or the input to a command. See also standard error.

stanza A grouping of options in a configuration file to control various aspects of compilation by default.

statement
In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static binding
The act of resolving references to external variables and functions before run time.

STDERR
See standard error.

STDIN
See standard input.

STDOUT
See standard output.

storage class specifier
A storage class keyword that determines storage duration, scope, and linkage.

stream
A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the fdopen() or fopen() functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output.

string A contiguous sequence of bytes terminated by and including the first null byte.

string constant
Zero or more characters enclosed in double quotation marks. See also string literal.

string literal
Zero or more characters enclosed in double quotation marks. See also string constant.

striped data set
An extended-format data set that occupies multiple volumes. A striped data set is a software implementation of sequential data striping.

struct See structure.

struct tag
See structure tag.

structure
A class data type that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

structure tag
The identifier that names a structure data type.

stub routine
Within a runtime library, a routine that contains the minimum lines of code needed to locate a given routine.

subprogram
In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

subscript
One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subtree
A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset
Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support
In system development, to provide the necessary resources for the correct operation of a functional unit.

switch expression
The controlling expression of a switch statement.

switch statement
A C language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default
A default value defined in the system profile.

system process
An implementation-dependent object, other than a process executing an application, that has a process ID. X/Open.

T

tab character
A character that indicates that printing or displaying should start at the next horizontal position on the current line. The tab is designated by '\t' in the C language and is named in the portable character set.

text file
A file that contains only printable characters.

thread A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

tilde One of the accent marks in Latin script (~).

token The basic syntactic unit of a computing language. A token consists of one or more characters, excluding the blank character and excluding characters within a string constant or delimited identifier.

toolchain
A collection of programs or tools used to develop a product.

traceback
A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and status of the routines on the call-chain at the time the traceback was produced.

trigraph
A sequence of three graphic characters that represent another graphic character. For example, in the C programming language, the trigraph ??= is used to denote the # character.

truncate
To shorten a field, value, statement, or string.

type definition
A definition of a name for a data type.

type specifier
In programming languages, a keyword used to indicate the data type of an object or function being declared.

U

ultimate consumer
The target for data in an input and output operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer
The source for data in an input and output operation. An ultimate producer can be a file, a device, or an array of bytes in memory.

unary expression
An expression that contains one operand.

undefined behavior
Referring to a program or function that might produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data. See also implementation-defined.

union tag
An identifier that names a union data type.

UNIX System Services
An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specifications and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

UTC See Coordinated Universal Time.

V

volatile attribute
An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement
A looping statement that executes one or more instructions repeatedly during the time that a condition is true.

white space
A sequence of one or more characters, such as the blank character, the newline character, or the tab character, that belong to the space character class.

wide character
A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character code
An integral value that corresponds to a single graphic symbol or control code.

wide-character string
A contiguous sequence of wide characters terminated by and including the first instance of a null wide character.

wide-oriented stream
A wide-oriented stream refers to a stream which only wide character input/output is allowed.

word
A fundamental unit of storage that refers to the amount of data that can be processed at a time. Word size is a characteristic of the computer architecture. See also doubleword, halfword.

working directory
The active directory. When a file name is specified without a directory, the current directory is searched.

writable static area (WSA)
An area of memory in a program that is modifiable during the running of a program. Typically, this area contains global variables and function and variable descriptors for dynamic link libraries (DLLs).

WSA
See writable static area.

Notices

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

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

IBM Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
U.S.A.

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

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

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 publication 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.

© Copyright IBM Corp. 2018

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write Enterprise Metal C for z/OS programs.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

Standards

The following standard is supported:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information, see International Organization for Standardization (ISO) (www.iso.org).

The following standards are supported in combination with the z/OS UNIX System Services element:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3).* For more information, see IEEE (www.ieee.org).
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.

- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Index

Special characters

–q options syntax 225

A

abbreviated compiler options 11
accessibility 233
  contact IBM 233
  features 233
AGGRCOPY compiler option 22
AGGREGATE compiler option 23
allocation, standard files with BPXBATCH 209
ANSIALIAS compiler option 24
ARCHITECTURE compiler option 27
ARGPARSE compiler option 31
ARMODE compiler option 32
as shell command
  options 213
ASM compiler option 33
ASMDATASIZE compiler option 34
assembling
  HLASM source files 191
ASSERT(NORESRICT) compiler option 35
ASSERT(RESTRICT) compiler option 35
assistive technologies 233
attributes, for DD statements 201

B

binding 4
BITFIELD compiler option 36
BPARM JCL parameter 199
BPXBATCH program
  syntax 209
building
  programs 197
    building 197

C

C language 2
cataloged procedures 199
  data sets used 201
CDAHLASM command
  options 205
CHARS compiler option 37
command
  syntax diagrams vii
COMPACT compiler option 38
compiler
  error messages 57
  input 161, 166
    valid input/output file types 164
  listing 83, 155, 156
    include file option (SHOWINC) 131
compiler *(continued)*
  listing *(continued)*
    source program option (SOURCE) 134
  object module optimization 108
  output
    create listing file 163
    create preprocessor output 163
    using compiler options to specify 162
    using DD statements to specify 167
    valid input/output file types 164
  using cataloged procedures supplied by IBM 165, 186
compiler options
  #pragma options 9
  abbreviations 11
  defaults 11
  IPA considerations 7
  overriding defaults 7
  pragma options 9
COMPRESS compiler option 40
concatenation
  multiple libraries 167
concatenation, multiple libraries 167
configuration file for metalc 221
  default name 224
contact
  z/OS 233
control section (CSECT)
  compiler option 43
CONVLIT compiler option 41
CPARM JCL parameter 199
cross-reference table 156
CSECT compiler option 43

D

data sets
  concatenating 167
  supported attributes 201
  usage 200
data types, preserving unsignedness 150
ddname
  defaults 200
DEBUG compiler option 46
debugging
  errors 55
  SERVICE compiler option 128
  SEVERITY compiler option 130
default
  compiler options 11
  output file names 84
  overriding compiler option 7
DEFINE compiler option 48
digraphs, DIGRAPH compiler option 49
disk search sequence
  LSEARCH compiler option 91
  SEARCH compiler option 126
DSAUSER compiler option 50

E

efficiency, object module optimization 108
ENUMSIZE compiler option 51
environment variable
  used to specify system and operational information to metalc 219
EPILOG compiler option 53
error
  messages
    directing to your terminal 144
escaping special characters 8, 166
EVENTS compiler option 55
exception handling
  compiler error message severity levels 57
EXPMAC compiler option 56
external
  names
    long name support 88

F

feature test macro 173
features 3
feedback xi
files
  names
    generated default 84
    include files 173
  searching paths 91, 126
FLAG compiler option 57
flag options syntax 226
FLOAT compiler option 58

G

GOFF compiler option 63

H

HALT compiler option 64
HALTONMSG compiler option 65
header files
  system 167
heading information 155
  for IPA Link listings 157
HGPR compiler option 66
HLASM
  as utility 191
HOT compiler option 67

I

IAP
  IPA link step
    listing heading information 157
ILP32 compiler option 89

© Copyright IBM Corp. 2018

INCLUDE compiler option 68
include files
  naming 173
  nested 104
  preprocessor directive
    syntax 173
  record format 173
  SHOWINC compiler option 131
INFILE parameter 199
INFO compiler option 69
INITAUTO compiler option 71
INLINE compiler option
  description 73
input
  compiler 161, 166
  record sequence numbers 127
IPA
  IAP link step
    compiler options map listing section 158
    global symbols map listing section 158
    listing message summary 159
    listing messages section 159
    listing prolog 157
    object file map listing section 157
    partition map listing section 158
    source file map listing section 157
  invoking from metalc utility 169
  IPA compile step
    flow of processing 170
  IPA compiler option 75
  IPA link step
    flow of processing 171
    invoking IPA from metalc utility 185
    IPA link step control file 186
    listing overview 155, 156
    object file directives 190
    overview 185
    troubleshooting 190
  overview 170
  using cataloged procedures 165
IPACNTL data set 200, 202
IPARM JCL parameter 199
IRUN JCL parameter 199

J

JCL (Job Control Language)
  C comments 138

K

keyboard
  navigation 233
  PF keys 233
  shortcut keys 233
KEYWORD compiler option 78

L

LANGLVL compiler option 79
LIBANSI compiler option 82
library 161
LIBRARY JCL parameter 199
linking 4
LIST compiler option 83
listings 155, 156
  include file option (SHOWINC) 131
  IPA compile step, using 155
  IPA link step compiler options map 158
  IPA link step global symbols map 158
  IPA link step heading information 157
  IPA link step message summary 159
  IPA link step messages 159
  IPA link step object file map 157
  IPA link step partition map 158
  IPA link step prolog 157
  IPA link step source file map 157
  IPA link step, using 156
  message summary 156
LOCALE compiler option 85
long names
  support 88
LONGLONG compiler option 87
LONGNAME compiler option 88
LP64 compiler option 89
LPARM parameter 199
LSEARCH compiler option 91

M

macor
  feature test 173
macro
  expanded in source listing 56
mainframe
  education x
maintaining
  programs through makefiles 207
make utility
  compiling source and object files 169
  creating makefiles 207
  maintaining application programs 207
MAKEDEP compiler option 97
makefiles
  creating 207
  maintaining application programs 207
MARGINS compiler option 99
MAXMEM compiler option 100
MEMBER JCL parameter 199
memory
  files, compiler work-files 102
  MAXMEM compiler option 100
  MEMORY compiler option 102
messages 156
  directing to your terminal 144
  on IPA link step listings 159
  specifying severity of 57
METAL compiler option 103
metalc compiler utility 4
metalc shell command
  environment variables 219
  specifying
    system and operational information to metalc 219
metalc utility 219
metalc utility *(continued)*
  compiling source and object files 169
  run by the make utility 169
MVS (Multiple Virtual System)
  z/OS batch
    running shell scripts and applications 209

N

natural reentrancy
  generating 117
navigation
  keyboard 233
NESTINC compiler option 104
NOAGGREGATE compiler option 23
NOANSIALIAS compiler option 24
NOARGPARSE compiler option 31
NOARMODE compiler option 32
NOASM compiler option 33
NOCOMPACT compiler option 38
NOCOMPRESS compiler option 40
NOCONVLIT compiler option 41
NOCSECT compiler option 43
NODEBUG compiler option 46
NODIGRAPH compiler option 49
NODSAUSER compiler option 50
NOEVENTS compiler option 55
NOEXPMAC compiler option 56
NOFLAG compiler option 57
NOGOFF compiler option 63
NOHALTONMSG compiler option 65
NOHGPR compiler option 66
NOHOT compiler option 67
NOINCLUDE compiler option 68
NOINFO compiler option 69
NOINITAUTO compiler option 71
NOINLINE compiler option 73
NOIPA compiler option 75
NOKEYWORD compiler option 78
NOLIBANSI compiler option 82
NOLIST compiler option 83
NOLOCALE compiler option 85
NOLONGLONG compiler option 87
NOLONGNAME compiler option 88
NOLSEARCH compiler option 91
NOMARGINS compiler option 99
NOMAXMEM compiler option 100
NOMEMORY compiler option 102
NOMETAL compiler option 103
NONESTINC compiler option 104
NOOE compiler option 104
NOOPTFILE compiler option 106
NOOPTIMIZE compiler option 108
NOPHASEID compiler option 111
NOPPONLY compiler option 112
NOPREFETCH compiler option 115
NORENT compiler option 117
NORESRICT compiler option 120
NOROCONST compiler option 121
NOROSTRING compiler option 123
NOSEARCH compiler option 126
NOSEQUENCE compiler option 127
NOSERVICE compiler option 128
NOSEVERITY compiler option 130
NOSHOWINC compiler option 131

NOSHOWMACROS compiler option 132
NOSOURCE compiler option 134
NOSPLITLIST compiler option 135
NOSSCOMM compiler option 138
NOSTRICT compiler option 139
NOSTRICT_INDUCTION compiler option 141
NOSUPPRESS compiler option 142
NOTERMINAL compiler option 144
NOUNROLL compiler option 149
NOUPCONV compiler option 150
NOVECTOR compiler option 151
NOWARN64 compiler option 153
NOWSIZEOF compiler option 154

O

object
  code 161
  module
    optimization 108
OBJECT
  JCL parameter 199
object files
  object file browse 172
  working with 172
object files variations
  object file variation identification 173
OE compiler option 104
OMVS
  OE compiler option 104
OPARM JCL parameter 199
OPTFILE compiler option 106
optimization
  object module 108
  OPTIMIZE compiler option 108
  storage requirements 108
  TUNE compiler option 145
OPTIMIZE compiler option 108
options
  compiler
    compiler options 10
OUTFILE parameter 199

P

PDF documents ix
PHASEID compiler option 111
PPARM
  JCL parameter 199
PPONLY compiler option 112
pragmas
  options 9
PREFETCH compiler option 115
preprocessor directives
  effects of PPONLY compiler option 112
  include 173
primary data set
  specifying input to the compiler 161
primary input
  compiler 161
PROLOG compiler option 116

R

record format
  system files and libraries
    OPTFILE compiler option 106
    SEARCH compiler option 126
    using 173
  user files and libraries
    using 173
reentrancy
  RENT compiler option 117
reentrant code
  RENT compiler option 117
RENT compiler option syntax 117
RESERVED_REG compiler option 119
RESTRICT compiler option 120
ROCONST compiler option 121
ROSTRING compiler option 123
ROUND compiler option 124

S

SEARCH compiler option 126
search sequence
  system include files 126
  user include files 91
secondary data set
  libraries 167
secondary input
  compiler 162, 167
sending to IBM
  reader comments xi
SEQUENCE compiler option 127
sequence numbers on input records 127
SERVICE compiler option 128
SEVERITY compiler option 130
shell
  compiling using metalc 169
  using BPXBATCH to run commands or scripts 209
shortcut keys 233
SHOWINC compiler option 131
SHOWMACROS compiler option 132
SKIPSRC compiler option 133
source
  program
    comment (SSCOMM compiler option) 138
    compiler listing options 131, 134
    file names in include files 173
    generating reentrant code 117
    input data set 161
    margins 99
    SEQUENCE compiler option 127
source code
  compiling using metalc 168
SOURCE compiler option 134
special characters, escaping 8, 166
SPLITLIST compiler option 135
SSCOMM compiler option 138
standard files, allocation for BPXBATCH 209
standards
  ANSI compiler option 79
  LIBANSI compiler option 82
STEPLIB
  data set 200, 201, 203
storage optimization 108
STRICT compiler option 139
STRICT_INDUCTION compiler option 141
structure and union maps, listing 156
SUPPRESS compiler option 142
syntax diagrams
  how to read vii
SYSCPRT data set 163, 200, 201, 203
SYSEVENT data set
  description of 202
SYSIN data set
  description of 202, 204
SYSIN data set for stdin
  description of 201
  primary input to the compiler 166
  usage 200
SYSLIB data set
  description of 201, 203, 204
  specifying 167
  usage 200
SYSLIN data set
  description of 201, 203
  usage 200
SYSOUT data set
  description of 201, 203, 204
  usage 200
SYSPRINT data set
  usage 200
SYSSTATE compiler option 143
system
  files and libraries 106, 126
system header files 167
SYSUT1 data set 200, 201
SYSUT5-10 data sets 201
SYSUTIP 202, 203

T

technical support x
TERMINAL compiler option 144
TUNE compiler option 145
type conversion, preserving unsignedness 150
type conversions 150
typographical conventions vii

U

UNDEFINE compiler option 148
UNIX System Services 4
UNROLL compiler option 149
unsignedness preservation, type conversion 150
UPCONV compiler option 150
user
  include files
    LSEARCH compiler option 91
    SEARCH compiler option 126
    specifying with #include directive 173
user interface
  ISPF 233
  TSO/E 233
USERLIB 92, 167, 202

V

VECTOR compiler option 151

W

WARN64 compiler option 153
work data sets 200
WSIZEOF compiler option 154

Z

z/OS Basic Skills Knowledge Center x
z/OS batch
  compiling under 165, 186
  running shell scripts and applications 209
z/OS Language Environment
  search sequence
    with LSEARCH compiler option 91
z/OS UNIX System Services 4
  as utility 191
  compiling using metalc 169
  maintaining through makefiles 207
  OE compiler option 104

IBM®

Product Number: 5655-MCE

Printed in USA

SC27-9051-00