Enterprise PL/I for z/OS

# Compiler and Run-Time Migration Guide

*Version 5 Release 1*

GC27-8930-00

Enterprise PL/I for z/OS

# Compiler and Run-Time Migration Guide

*Version 5 Release 1*

GC27-8930-00

**Note**
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 193.

**First Edition (August 2017)**

This edition applies to Version 5 Release 1 of Enterprise PL/I for z/OS, 5655-PL5, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, Department H150/090
555 Bailey Ave
San Jose, CA, 95141-1099
United States of America

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

**© Copyright IBM Corporation 1999, 2017.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents


**Tables . . . . . ix**

**Figures . . . . . xi**

**About this book . . . . . xiii**
Using your documentation . . . . . xiii
How to send your comments . . . . . xiv
Accessibility . . . . . xv

**Part 1. Overview . . . . . 1**

**Chapter 1. Do I need to recompile? . . . 3**
Migration basics . . . . . 3
Runtime migration - Moving to Language Environment . . . . . 4
Compiler migration . . . . . 4
Migration roadmap . . . . . 5
Service support for OS PL/I and PL/I for MVS & VM . . . . . 5

**Chapter 2. Introducing the new compiler and run time . . . . . 7**
Product relationships - compiler, run time, Debug Tool . . . . . 7
General PL/I compiler information . . . . . 7
Language Environment's runtime support for other programs . . . . . 8
Advantages of the new compiler and run time . . . 8
Major changes with the new compiler and run time . 9
General conversion tasks . . . . . 10
Planning your strategy . . . . . 10
Moving to the Language Environment run time 10
Recompiling your source with Enterprise PL/I . 11
Adding Enterprise PL/I programs to existing applications . . . . . 11

**Part 2. Conversion Strategies . . . 13**

**Chapter 3. Planning the move to Language Environment . . . . . 15**
Preparing to move to the Language Environment runtime library . . . . . 15
Installing Language Environment . . . . . 15
Assessing storage requirements . . . . . 15
Educating your programmers about Language Environment . . . . . 16
Taking an inventory of your applications . . . . 16
Vendor tools, packages, and products . . . . . 17
PL/I applications . . . . . 17
Existing PL/I load modules . . . . . 18
Deciding how to phase in Language Environment 18
Handling multilanguage conversion . . . . . 18
Determining how applications will have access to the library . . . . . 19
Setting up a regression testing procedure . . . . 21
Taking performance measurements . . . . . 22
Cutting over to production use . . . . . 22

**Chapter 4. Planning to move to the new compiler . . . . . 23**
Preparing to move your source to the new compiler 23
Installing Enterprise PL/I. . . . . 23
Assessing storage requirements . . . . . 23
Educating your programmers on new compiler features . . . . . 24
Taking an inventory of your applications . . . . 24
Taking an inventory of vendor tools, packages, and products . . . . . 24
Taking an inventory of PL/I applications . . . 24
Prioritizing your applications . . . . . 25
Determining conversion priority . . . . . 25
Setting up move/no move categories . . . . . 25
Making application program updates . . . . . 26

**Part 3. Moving existing applications to Language Environment . . . . . 29**

**Chapter 5. Running existing applications under Language Environment . . . . . 31**
Invoking existing applications . . . . . 31
Considerations for non-CICS applications . . . 31
Considerations for CICS applications . . . . . 32
Link-editing existing applications . . . . . 32

**Chapter 6. Considerations before migrating. . . . . 35**
Differences in runtime options . . . . . 35
Deleted runtime options . . . . . 35
Replaced runtime options . . . . . 35
New runtime options . . . . . 36
Differences in condition handling . . . . . 37
Timing differences . . . . . 37
Differences in performing some fixed decimal divides with large precision . . . . . 38
Unhandled condition differences . . . . . 38
IBMBXITA and IBMBEER differences . . . . . 38
ABEND U4039 differences . . . . . 38
Severity differences . . . . . 39
Differences in PLICALLA and PLICALLB support 39
PLICALLA considerations . . . . . 39
PLICALLB considerations . . . . . 39
Differences in preinitialization support . . . . . 41
Differences in PLISRTx support. . . . . 42
Differences in multitasking support . . . . . 42


© Copyright IBM Corp. 1999, 2017

Differences in OS PL/I shared library support . . . 42
Differences in DATE/TIME built-in functions . . . 43
Differences in User Return Code . . . . . 43
Differences in runtime messages . . . . . 43
Differences in PLIDUMP . . . . . 44
Differences in storage report . . . . . 45
Differences in Interlanguage Communication support . . . . . 45
Differences in assembler support . . . . . 46

**Chapter 7. Object module and load module considerations . . . . . 49**
OS PL/I Version 1 object module and load module compatibility . . . . . 49
OS PL/I Version 1 Release 5.1 . . . . . 49
OS PL/I Version 1 Release 5 . . . . . 50
OS PL/I Version 1 Release 3.0 - Release 4.0 . . . 51
OS PL/I earlier than Version 1 Release 3.0 . . . 51
OS PL/I Version 2 object module and load module compatibility . . . . . 51
Summary of support for OS PL/I object modules and load modules . . . . . 51

**Chapter 8. Link-Edit considerations . . 53**
SCEERUN . . . . . 53
Symbol table considerations . . . . . 53
NCAL linkage editor option . . . . . 53
ENTRY cards . . . . . 54
Using OS PL/I math routines . . . . . 54

**Chapter 9. Subsystem considerations 55**
CICS considerations . . . . . 55
Updating CICS system definition (CSD) file . . 55
Error handling . . . . . 55
Restrictions on user-written condition handlers under CICS . . . . . 56
Macro-level interface . . . . . 56
FETCHing a PL/I MAIN Procedure . . . . . 56
STACK runtime option . . . . . 56
Runtime output . . . . . 56
Abend codes used by PL/I under CICS . . . . 57
IMS considerations . . . . . 57
Interfaces to IMS . . . . . 57
SYSTEM(IMS) compile-time option . . . . . 57
PLICALLA support in IMS . . . . . 57
PSB language options supported . . . . . 58
Storage usage considerations . . . . . 58
Coordinated condition handling under IMS . . 58
Performance enhancement with Library Retention (LRR) . . . . . 59
DB2 considerations . . . . . 59

**Part 4. Moving to the new compiler 61**

**Chapter 10. Understanding the limitations of the new compiler . . . . 63**
Language Environment requirements . . . . . 63
Language not supported . . . . . 63
Language restricted . . . . . 63
RECORD I/O . . . . . 64
STREAM I/O . . . . . 64
Structure expressions . . . . . 65
Array expressions . . . . . 65
DEFAULT statement . . . . . 65
Extents of automatic variables . . . . . 65
Built-in functions . . . . . 66
DEFINED BIT aggregates . . . . . 66
OPTIONS(REENTRANT) . . . . . 66
iSUB defining . . . . . 66
LABEL arrays . . . . . 66
DBCS . . . . . 67
GRAPHIC and POSITION . . . . . 67
Macro preprocessor . . . . . 67
Options restricted . . . . . 67
Options not supported . . . . . 68
Restrictions on other interfaces to the compiler . . 68
Batch compilation . . . . . 68
Invoking the compiler from assembler . . . . 69
Compiling under TSO . . . . . 69
Specifying INCLUDE data set names . . . . . 70
Defining the SYSLIN data set . . . . . 70
Compiler time and space requirements . . . . . 70
AMODE(24) restrictions . . . . . 70
EXTERNAL names restricted . . . . . 71
Listing differences . . . . . 71
Control block differences . . . . . 72
ISAM support differences . . . . . 72

**Chapter 11. Understanding the new compiler's options . . . . . 73**
Understanding the effect of default options on compatibility . . . . . 73
BACKREG(5) . . . . . 73
BIFPREC(15) . . . . . 74
CMPAT(V2) . . . . . 74
EXTRN(FULL) . . . . . 75
LIMITS(EXTNAME(7)) . . . . . 75
NORENT and WRITABLE . . . . . 76
SYSTEM . . . . . 76
Choosing nondefault options for even more compatibility . . . . . 76
COMMON . . . . . 77
DFT(NOBIN1ARG) . . . . . 77
DEFAULT(LINKAGE(SYSTEM)) . . . . . 77
DFT(OVERLAP) . . . . . 77
NOREDUCE . . . . . 77
NORESEXP . . . . . 78
RULES(LAXCTL) . . . . . 78
RULES(NOLAXINOUT NOLAXSEMI) . . . . 78
NOWRITABLE . . . . . 78
Choosing options for improved performance . . . 79
ARCH . . . . . 79
BIFPREC(31) . . . . . 79
DEFAULT(NONASGN) . . . . . 80
DEFAULT(CONNECTED) . . . . . 80
DEFAULT(REORDER) . . . . . 80
DEFAULT(NOOVERLAP) . . . . . 80
OPTIMIZE(2)/OPTIMIZE(3) . . . . . 80
REDUCE . . . . . 81
NORENT . . . . . 82

RULES(NOLAXCTL) . . . . . 82
Choosing options for better quality . . . . . 83
RULES(NOLAXDCL) . . . . . 83
RULES(NOLAXIF) . . . . . 83
RULES(NOLAXLINK) . . . . . 84
RULES(NOLAXMARGINS) . . . . . 85
RULES(LAXSTRZ) . . . . . 85
RULES(NOMULTICLOSE) . . . . . 85
Choosing options for test . . . . . 85
CHECK(CONFORMANCE) . . . . . 86
GONUMBER . . . . . 86
PREFIX . . . . . 86
TEST . . . . . 86

**Chapter 12. Understanding the new compiler's messages . . . . . 87**
IBM1044: one-byte FIXED BIN . . . . . 87
IBM1053: scaled FIXED BIN evaluation . . . . . 87
IBM1065: imprecise float constants. . . . . 87
IBM1091: FIXED BIN precision warning . . . . . 88
IBM1099: mixed FIXED . . . . . 88
IBM1181: miscoded DO loops . . . . . 89
IBM1206: misuse of BIT operators . . . . . 90
IBM1208: incompletely initialized arrays. . . . . 90
IBM1215: incomplete declares . . . . . 91
IBM1216: incorrect structure declares . . . . . 91
IBM1220: pointless comparisons . . . . . 92
IBM1927: SIZE condition . . . . . 92
IBM1948: restricted expression evaluation . . . . 93
IBM2063: invalid ALLOCATE . . . . . 93
IBM2402: storage overlay . . . . . 93
IBM2409: RETURN; in a function . . . . . 94
IBM2410: No RETURN in a function . . . . . 94
IBM2412: missing RETURNS option . . . . . 94
IBM2421: CLOSE in ENDFILE . . . . . 95
IBM2610: precision interpretation . . . . . 95
IBM2611, IBM2612: duplicate whens . . . . . 95
IBM2617: passing labels out of PL/I . . . . . 96
IBM2621: missing ON ERROR SYSTEM . . . . . 96
IBM2622: warning on poorly coded DO loops . . . 96
IBM2626: SUBSTR with a zero length. . . . . 97
IBM2628: large BYVLAUE parameters . . . . . 97
IBM2801: introduction of scaled FIXED BIN . . . 98
IBM2804: suboptimal compares. . . . . 98
IBM2810: conversion of scaled FIXED BIN . . . . 98
IBM2811: use of PICTURE as DO control variables 99
IBM2812: poor TRANSLATE and VERIFY . . . . 99
PLIXOPT messages . . . . . 99
Using the compiler user exit . . . . . 100

**Chapter 13. Understanding when working code must be changed . . . 101**
Incorrect code . . . . . 101
Relying on the order of declarations. . . . . 101
Using invalid FIXED DECIMAL data . . . . 101
Using invalid SUBSTR references. . . . . 102
Using dissimilar EXTERNAL declares . . . . 102
Using an incorrect PLITABS declare . . . . . 103
Initializing variables . . . . . 103
Initializing AUTOMATIC . . . . . 103
Initializing BASED . . . . . 104
Initializing CONTROLLED . . . . . 104
Initializing STATIC . . . . . 104
Retaining unused declarations. . . . . 104
Retaining unused INTERNAL STATIC . . . . 104
Incorrect code that will now raise exceptions . . . 105
FIXEDOVERFLOW when SIZE is disabled . . 105
Invalid allocations. . . . . 106
Incorrect code that will now loop endlessly . . . 107
Even precision PICTURE loop control variables 107
Assignments that will produce different results . . 108
Source-target overlap . . . . . 108
Float-to-float assignments . . . . . 109
Other statements that will produce different results 110
STREAM I/O with unprintable characters . . . 110
Uninitialized EXTERNAL STATIC . . . . . 110
Incompletely declared FILEs . . . . . 111
Dummy arguments and alignment . . . . . 111
Dummy arguments and CONTROLLED . . . 112
Pointer arithmetic . . . . . 112
Code that will not perform as well . . . . . 112
FIXED DEC as a loop control . . . . . 112
FIXED BIN(15) as a loop control . . . . . 113
I/O using TOTAL . . . . . 113

**Chapter 14. Understanding when working code may need to be changed . . . . . 115**
Code that will now raise an exception . . . . . 115
ZERODIVIDE and OVERFLOW promoted to ERROR . . . . . 115
Conditions raised when disabled . . . . . 115
Invalid RETURNs . . . . . 116
GOTO holes . . . . . 116
The scope of NOFOFL . . . . . 117
Code that will now not raise exceptions . . . . 117
FIXEDOVERFLOW for FIXED BIN . . . . . 117
CONVERSION when assigning blanks to numeric variables . . . . . 117
ERROR when mapping excessively large aggregates . . . . . 118
Storage mapped differently. . . . . 118
One-byte FIXED BIN . . . . . 118
Declarations handled differently . . . . . 119
AREA with INITIAL . . . . . 119
Conversions handled differently . . . . . 119
Conversions from float to character . . . . . 119
Conversions from scaled FIXED BINARY . . . 119
Built-in functions handled differently . . . . . 120
Arithmetic built-in functions with scale factors and FIXED BIN. . . . . 120
String-handling built-in function for conversion of DBCS character strings . . . . . 121
MACRO preprocessor differences. . . . . 122
SQL preprocessor differences . . . . . 122

**Chapter 15. Linking your new objects 125**
Prelinker and PDSE considerations . . . . . 125
AMODE(24) considerations. . . . . 125
Using PLICALLA or PLICALLB entry . . . . . 125

CHANGE cards . . . . . 125

**Chapter 16. Using Language Environment with the new compiler. . 127**
Using the right runtime options . . . . . 127
Calling PL/I from assembler main programs . . . 128
Understanding when your results might vary . . 128
Return codes . . . . . 128
When the run time issues messages . . . . . 128
What the runtime messages say . . . . . 129
Where the runtime messages go . . . . . 129
Math built-ins . . . . . 129
Dumps . . . . . 130
Storage reports . . . . . 130
Prerequisite Language Environment PTFs . . . . 130

**Chapter 17. Tuning for better CPU and storage utilization . . . . . 131**
Improving CPU utilization . . . . . 131
Improving storage utilization . . . . . 132
Improving performance under subsystems . . . 132

**Chapter 18. Adding Enterprise PL/I programs to existing PL/I applications 135**
Object and load module considerations. . . . . 135
Sharing SYSPRINT . . . . . 136
Runtime option considerations . . . . . 137
Condition handling considerations . . . . . 137
Partitioning PL/I source programs into units of execution. . . . . 138

**Chapter 19. Migrating from earlier releases of Enterprise PL/I to Enterprise PL/I V5R1 . . . . . 139**
Migrating from Enterprise PL/I V4R5 . . . . . 139
Migrating from Enterprise PL/I V4R4 . . . . . 141
Migrating from Enterprise PL/I V4R3 . . . . . 142
Migrating from Enterprise PL/I V4R2 . . . . . 143
Migrating from Enterprise PL/I V4R1 . . . . . 144
Migrating from Enterprise PL/I Version 3 (all releases) . . . . . 146
Messages that are introduced with V5R1 . . . . 148
Messages that are introduced with V4R5 . . . . 149
Messages that are introduced with V4R4 . . . . 150
Messages that are introduced with V4R3 . . . . 151
Messages that are introduced with V4R2 . . . . 153
Compiler messages that are introduced with V4R1 155
Compiler messages that are introduced with V3R9 156
Compiler messages that are introduced with V3R8 157
Compiler messages that are introduced with V3R7 157
Compiler messages that are introduced with V3R6 158
Compiler messages that are introduced with V3R5 158
Compiler messages that are introduced with V3R4 159
Object compatibility . . . . . 160
Runtime changes . . . . . 162

**Part 5. Subsystem and other language considerations . . . . . 163**

**Chapter 20. Assembler considerations for PL/I applications . . . . . 165**
Considerations for assembler programs mimicking PL/I main procedures . . . . . 165
Calling PL/I from assembler and Language Environment conforming assembler . . . . . 165
Condition handling and assembler programs . . . 166
Considerations for using assembler user exits . . 166

**Chapter 21. CICS considerations for PL/I applications . . . . . 167**
General CICS considerations . . . . . 167
Updating CICS System Definition (CSD) file . . 167
Macro-level interface . . . . . 168
Compiler options for programs that run under CICS . . . . . 168
Linking CICS applications and runtime considerations . . . . . 168
Error-handling . . . . . 168
FETCHing a PL/I MAIN procedure . . . . . 168
Runtime output . . . . . 168
Abend codes used by PL/I under CICS . . . 169
Migrating to the integrated CICS preprocessor . . 169

**Chapter 22. IMS considerations for PL/I applications . . . . . 171**
Interfaces to IMS . . . . . 171
SYSTEM(IMS) compile-time option . . . . . 171
PLICALLA support in IMS . . . . . 171
PSB language options supported . . . . . 171
Storage usage considerations . . . . . 172
Coordinated condition handling under IMS . . . 172
Performance enhancement with Library Retention (LRR) . . . . . 173

**Chapter 23. DB2 considerations for PL/I applications . . . . . 175**
General DB2 considerations . . . . . 175
Migrating to the integrated SQL preprocessor . . 175
Programming and compilation considerations 175
FOR BIT DATA assignment notes. . . . . 176
Prerequisite DB2 APARs. . . . . 176

**Part 6. Appendixes . . . . . 177**

**Appendix A. Conversion and migration aids . . . . . 179**
OS PL/I Routine Replacement Tool . . . . . 179
OS PL/I Version 1 Release 5.1 main load module ZAP . . . . . 180
OS PL/I Shared Library replacement tool . . . . 180
OS PL/I Object Module Relinking Tool - APAR PN69803 . . . . . 181
ILC applications . . . . . 181
PLISRTx applications. . . . . 181
EDGE Portfolio Analyzer . . . . . 182
Vendor products . . . . . 182

**Appendix B. Compiler elements comparison . . . . . 183**

**Appendix C. Compiler limit comparison . . . . . 185**

**Appendix D. Batch processing sample 189**

**Appendix E. Debugging tool comparison . . . . . 191**
Differences between debugging tools . . . . . 191

**Notices . . . . . 193**
Programming interface information . . . . . 194
Trademarks . . . . . 194

**Bibliography . . . . . 195**
PL/I publications . . . . . 195
Related publications . . . . . 195

**Index . . . . . 197**

# Tables


1. How to use Enterprise PL/I publications xiii
2. How to use z/OS Language Environment publications . . . . . xiv
3. PL/I compiler IDR values. . . . . 18
4. Specification of new DDNAMEs . . . . . 31
5. Mapping of SPIE and STAE Options to the TRAP Option . . . . . 35
6. OS PL/I Version 2 Release 3 ERROR ON-Unit and message for an ERROR condition. . . . . 37
7. Language Environment ERROR ON-Unit and message for an ERROR condition . . . . . 38
8. Differences in PLICALLB argument list support . . . . . 40
9. Return Code Behavior under Language Environment . . . . . 43
10. Summary of object module and load module support by Language Environment . . . . . 51
11. PSB LANG options for IMS/ESA Version 4 Release 1, and later . . . . . 58
12. PSB LANG options for IMS/ESA Version 4 Release 1, and later . . . . . 172
13. PL/I element names . . . . . 183
14. Language element limits . . . . . 185
15. PLITEST Commands and Their Debug Tool Equivalents . . . . . 191


© Copyright IBM Corp. 1999, 2017

# Figures

1. CESE Output Data Queue. . . . . 57
2. CESE Output Data Queue . . . . . 169

© Copyright IBM Corp. 1999, 2017

# About this book

This book provides information to help you to move from a pre-Language Environment runtime library to IBM Language Environment for z/OS and to upgrade your source programs to IBM Enterprise PL/I for z/OS Version 5 Release 1. The book suggests solutions to problems that arise because of differences in support between previous releases of PL/I (OS PL/I, PL/I for MVS & VM, and VisualAge PL/I) and Enterprise PL/I.

**Important:** The information in this book discusses migration considerations for using Enterprise PL/I V5R1M0 and z/OS V2R1 Language Environment or later. These two products must be installed in order to take advantage of the migration enhancements discussed in this book. The use of Enterprise PL/I refers to Version 5 Release 1 unless indicated otherwise. The use of Language Environment refers to z/OS V2R1 Language Environment or later unless indicated otherwise.

This book is for system programmers, application programmers, and IBM support personnel who are involved in PL/I product migration. You must have the following prerequisite knowledge to use this book:

- A general understanding of your operating system
- Some knowledge of the PL/I language and options
- Some knowledge of how PL/I uses Language Environment for its runtime environment

## Using your documentation

The publications provided with Enterprise PL/I are designed to help you program with PL/I. The publications provided with Language Environment are designed to help you manage your runtime environment for applications generated with Enterprise PL/I. Each publication helps you perform a different task.

The following tables show you how to use the publications you receive with Enterprise PL/I and Language Environment. You will want to know information about both your compiler and runtime environment. For the complete titles and order numbers of these and other related publications, see "Bibliography" on page 195.

### PL/I information

*Table 1. How to use Enterprise PL/I publications*

| To... | Use... |
|---|---|
| Evaluate Enterprise PL/I | Fact Sheet |
| Understand warranty information | Licensed Programming Specifications |
| Plan for and install Enterprise PL/I | Enterprise PL/I Program Directory |
| Understand compiler and runtime changes and adapt programs to Enterprise PL/I and Language Environment | Compiler and Run-Time Migration Guide |
| Prepare and test your programs and get details on compiler options | Programming Guide |
| Get details on PL/I syntax and specifications of language elements | Language Reference |

© Copyright IBM Corp. 1999, 2017

*Table 1. How to use Enterprise PL/I publications (continued)*

| To... | Use... |
|---|---|
| Diagnose compiler problems and report them to IBM | Diagnosis Guide |
| Get details on compile-time messages | Compile-Time Messages and Codes |

### Language Environment information

*Table 2. How to use z/OS Language Environment publications*

| To... | Use... |
|---|---|
| Evaluate Language Environment | Concepts Guide |
| Plan for Language Environment | Concepts Guide<br>Run-Time Migration Guide |
| Install Language Environment on z/OS | z/OS Program Directory |
| Customize Language Environment on z/OS | Customization |
| Understand Language Environment program models and concepts | Concepts Guide<br>Programming Guide |
| Find syntax for Language Environment runtime options and callable services | Programming Reference |
| Develop applications that run with Language Environment | Programming Guide and your language Programming Guide |
| Debug applications that run with Language Environment, get details on runtime messages, diagnose problems with Language Environment | Debugging Guide and Run-Time Messages |
| Develop interlanguage communication (ILC) applications | Writing Interlanguage Applications |
| Migrate applications to Language Environment | Run-Time Application Migration Guide and the migration guide for each Language Environment-enabled language |

## How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information.

If you have comments about this document or any other PL/I documentation, contact us in one of these ways:

- Use the Online Readers' Comment Form at

  `www.ibm.com/software/awdtools/rcf/`

  or send an e-mail to

  `comments@us.ibm.com`

  Be sure to include the name of the document, the publication number of the document, the version of PL/I, and, if applicable, the specific location (for example, page number) of the text that you are commenting on.

- Fill out the Readers' Comment Form at the back of this document, and return it by mail or give it to an IBM representative. If the form has been removed, address your comments to:

  International Business Machines Corporation
  Reader Comments
  H150/090
  555 Bailey Avenue
  San Jose, CA 95141-1003
  USA
- Fax your comments to this U.S. number: (800)426-7773.

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

# Accessibility

Accessibility features assist users who have a disability, such as restricted mobility or limited vision, to use information technology content successfully. The accessibility features in z/OS provide accessibility for Enterprise PL/I.

## Accessibility features

z/OS includes the following major accessibility features:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- Ability to customize display attributes such as color, contrast, and font size

z/OS uses the latest W3C Standard, WAI-ARIA 1.0 (http://www.w3.org/TR/wai-aria/), to ensure compliance to US Section 508 (http://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards/section-508-standards) and Web Content Accessibility Guidelines (WCAG) 2.0 (http://www.w3.org/TR/WCAG20/). To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Enterprise PL/I online product documentation in IBM® Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at http://www.ibm.com/support/knowledgecenter/en/about/releasenotes.html.

## Keyboard navigation

Users can access z/OS user interfaces by using TSO/E or ISPF.

Users can also access z/OS services by using IBM Rational® Developer for System z®.

For information about accessing these interfaces, see the following publications:

- *z/OS TSO/E Primer* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120)

- *z/OS TSO/E User's Guide* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3)
- *z/OS ISPF User's Guide Volume I* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70)
- IBM Rational Developer for System z Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en)

These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

## Interface information

The Enterprise PL/I online product documentation is available in IBM Knowledge Center, which is viewable from a standard web browser.

PDF files have limited accessibility support. With PDF documentation, you can use optional font enlargement, high-contrast display settings, and can navigate by keyboard alone.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains the period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.

Assistive technology products work with the user interfaces that are found in z/OS. For specific guidance information, see the documentation for the assistive technology product that you use to access z/OS interfaces.

## Related accessibility information

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service
800-IBM-3383 (800-426-3383)
(within North America)

## IBM and accessibility

For more information about the commitment that IBM has to accessibility, see IBM Accessibility (www.ibm.com/able).

# Part 1. Overview

© Copyright IBM Corp. 1999, 2017

# Chapter 1. Do I need to recompile?

Ideally, programs should be compiled with IBM Enterprise PL/I for z/OS and run with the supported runtime library (Language Environment). You can reach this ideal state gradually, by starting with a runtime migration followed by a compiler migration. This chapter explains when and why you might want to migrate your applications (run time or source).

### Terminology clarification

When you use this book, be aware of the following terms:

*Enterprise PL/I*
: Is a general reference to IBM Enterprise PL/I for z/OS Version 5 Release 1.

*PL/I*
: Is a general reference to the following products:
  - OS PL/I
  - PL/I for MVS & VM
  - VisualAge PL/I
  - Enterprise PL/I

*new PL/I compilers*
: Refers to the following compiler products:
  - VisualAge PL/I
  - Enterprise PL/I

*old PL/I compilers*
: Refers to the following compiler products:
  - OS PL/I V3R2 and earlier
  - PL/I for MVS & VM

### Important migration note

It is important to understand, from the very beginning, that the *old* and *new* PL/I compilers are completely different from each other. The *new* PL/I compilers are written in PL/I, and do not make use of certain techniques that the *old* PL/I compilers did. They are so different, in fact, that from the perspective of Language Environment, they are considered different languages, each with its own signature CSECT.

In the past, migrating from an *old* PL/I compiler to another *old* PL/I compiler was not that difficult. With the introduction of the new Enterprise PL/I compiler, the migration process might be much more complicated than before. Migrating to the *new* Enterprise PL/I compiler must be a well researched, planned, and executed project if you want to have a smooth transition.

## Migration basics

The migration process involves runtime migration (moving your applications to a new runtime) and compiler migration (compiling your source programs with the new compiler).

As part of the migration process, you will also need to do inventory assessment and testing. You are not required to migrate your runtime and source concurrently.

© Copyright IBM Corp. 1999, 2017

For more details on the migration process, see “General conversion tasks” on page 10.

For information about performing an inventory assessment and test plan, see “Taking an inventory of your applications” on page 16.

## Runtime migration - Moving to Language Environment

Every PL/I program requires runtime library routines to execute.

Do not make more than one PL/I runtime library available to your applications at execution time. For example, there should be one and only one PL/I runtime library, such as SCEERUN for Language Environment, in LNKLST. If you have more than one, you will either get hard-to-find errors or have an unused load library in your concatenation. In addition, if you have more than one runtime library in your concatenation, you have an invalid configuration that is not supported by IBM.

If you have not already moved to Language Environment and are using a pre-Language Environment PL/I compiler, such as OS PL/I V2R3, you will need to read Chapter 3, “Planning the move to Language Environment,” on page 15.

If you have already moved to Language Environment and are migrating to the new IBM Enterprise PL/I for z/OS compiler, you can begin reading about compiler migration in Chapter 4, “Planning to move to the new compiler,” on page 23.

## Compiler migration

It is strongly recommended that you recompile all your source with the new Enterprise PL/I compiler (unless you have already recompiled all your source with VisualAge PL/I). Because the Enterprise PL/I compiler is a completely different compiler from the *old* PL/I compilers, recompiling your source might be the best way to avoid the limitations imposed by mixing *new* PL/I objects and load modules with *old* PL/I objects and load modules.

Compiler migration can be done all at once or by separate execution units. For information about how to divide your PL/I source into separate execution units, see “Partitioning PL/I source programs into units of execution” on page 138.

If you decide to mix *old* PL/I modules with Enterprise PL/I modules, there are limited circumstances in which this mix will work. For details, see “Object and load module considerations” on page 135.

In a few cases, some changes to your code will be necessary when you move from OS PL/I to Enterprise PL/I. For details, see Chapter 13, “Understanding when working code must be changed,” on page 101 and Chapter 14, “Understanding when working code may need to be changed,” on page 115.

If you have already moved to Language Environment and are migrating to the new IBM Enterprise PL/I for z/OS compiler, you can begin reading about migrating to the new compiler in Chapter 4, “Planning to move to the new compiler,” on page 23.

If you are migrating to the PL/I for MVS & VM compiler, you must use the *IBM PL/I for MVS & VM Compiler and Run-Time Migration Guide*.

## Migration roadmap

This topic provides a short summary of migration possibilities.

### Migrating from OS PL/I or from PL/I for MVS & VM

- If you are NOT currently migrated to Language Environment, you can migrate to Language Environment first, and then migrate to Enterprise PL/I for z/OS. In this case, you can begin with Chapter 3, "Planning the move to Language Environment," on page 15 and then continue with Part 3, "Moving existing applications to Language Environment," on page 29.
- If you are migrating from OS PL/I, it is recommended that you migrate to PL/I for MVS & VM first. In this case, you must follow the *IBM PL/I for MVS & VM Compiler and Run-Time Migration Guide*.
- If you have already migrated to Language Environment, you can migrate to Enterprise PL/I for z/OS. In this case, you can begin with Chapter 2, "Introducing the new compiler and run time," on page 7, and then continue with Chapter 4, "Planning to move to the new compiler," on page 23 and Part 4, "Moving to the new compiler," on page 61.

### Migrating from VisualAge PL/I or an earlier release of Enterprise PL/I

Review Part 4, "Moving to the new compiler," on page 61, paying special attention to Chapter 19, "Migrating from earlier releases of Enterprise PL/I to Enterprise PL/I V5R1," on page 139.

For additional information concerning subsystems, see Part 5, "Subsystem and other language considerations," on page 163.

## Service support for OS PL/I and PL/I for MVS & VM

IBM will continue to provide service support for the execution of programs compiled with the OS PL/I compiler when these programs use the Language Environment runtime library versions of the PL/I library routines.

**Note:** The CICS TS (Transaction Server) release that follows CICS TS Version 2 Release 2 will not support OS PL/I modules. You must move from OS PL/I to a Language Environment-enabled PL/I compiler to use CICS after CICS TS V2R2.

For more information about this support and its restrictions, see Chapter 7, "Object module and load module considerations," on page 49.

# Chapter 2. Introducing the new compiler and run time

This chapter provides an overview of the Enterprise PL/I compiler (IBM Enterprise PL/I for z/OS) and the common run time (Language Environment) and introduces you to the terminology used throughout this book.

## Product relationships - compiler, run time, Debug Tool

IBM Enterprise PL/I for z/OS is IBM's strategic PL/I compiler for the z Systems platform. Language Environment provides a language runtime environment. Debug Tool provides significantly improved debugging functions.

**IBM Enterprise PL/I for z/OS**
Enterprise PL/I is comprised of features from OS PL/I, PL/I for MVS & VM, and VisualAge PL/I with additional features such as Unicode support, XML parsing capabilities, improved C and Java interoperability, integrated CICS preprocessor, and integrated SQL preprocessor.

**Language Environment**
Language Environment provides a single language runtime environment for COBOL, PL/I, C, and FORTRAN. In addition to support for existing applications, Language Environment also provides common condition handling, improved interlanguage communication (ILC), reusable libraries, and more efficient application development of interlanguage applications. Application development is simplified by the use of common conventions, common runtime facilities, and a set of shared callable services. Language Environment is required to run Enterprise PL/I programs.

**Debug Tool**
You can use Debug Tool to debug Enterprise PL/I programs and other Language Environment-conforming language programs including COBOL and C/C++. Debug Tool provides significantly improved debugging functions over previous PL/I debugging tools.

## General PL/I compiler information

You must have access to Language Environment when you compile your Enterprise PL/I application. When you compile your application and you use existing JCL, ensure that your STEPLIB or JOBLIB statement includes SCEERUN (the Language Environment runtime library), or that SCEERUN is in LNKLST.

You can use the IBMZC cataloged procedure to compile PL/I applications.

Your compile step should include the following:

```
//PLI      EXEC PGM=IBMZPLI,REGION=4000K
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
```

Reading about the cataloged procedures provided with Enterprise PL/I can help you understand the use of SCEERUN during compilation. See “Using PL/I cataloged procedures” in the *Enterprise PL/I for z/OS Programming Guide* for more details.

© Copyright IBM Corp. 1999, 2017

When you link-edit your Enterprise PL/I application with Language Environment and you use existing JCL, ensure that your SYSLIB statement includes SCEELKED (the Language Environment link-time library).

## Language Environment's runtime support for other programs

Enterprise PL/I uses Language Environment as its runtime environment. Language Environment provides a single language runtime environment for COBOL, PL/I, C, and FORTRAN.

Language Environment is the common runtime environment for the following language compilers:

C/370
C/C++
COBOL for MVS & VM
COBOL for OS/390 & VM
Fortran
PL/I for MVS & VM
Enterprise PL/I

Language Environment provides a common set of runtime options and callable services. It also improves interlanguage communication (ILC) between high-level languages (HLL) and the assembler by eliminating language-specific initialization and termination on each ILC invocation. Language Environment provides compatibility support for existing applications with a few restrictions.

## Advantages of the new compiler and run time

The new IBM Enterprise PL/I for z/OS compiler has many new features and advantages. This topic gives you an overview of the enhancements that you can enjoy.

- FETCH improvements:
  - FETCHed routines can FETCH other routines.
  - FETCHed routines can perform same I/O as MAIN.
  - FETCHed routines can have their own CONTROLLED.
- 31 digit DECIMAL and PICTURE precision
- Increased limits:
  - Internal and external names can have up to 100 characters.
  - There is no compiler limit on the number of FILEs and CONTROLLED variables.
  - Up to 4095 parameters are allowed per PROCEDURE.
- Support for many new 390 instructions (such as AHI and ALCR)
- Support for writeable reentrant static and DLLs
- Easier compatibility and interoperability with C/C++
- Better integer support:
  - Maximum precision of 63 for signed FIXED BIN
  - UNSIGNED attribute supported (with a maximum precision of 64)
  - Signed FIXED BIN(7) mapped to one byte (as is UNSIGNED FIXED BIN(8))
- Many powerful new language features, including these:
  - PACKAGEs (the ANSI alternative to secondary ENTRYs)
  - DO FOREVER (as a good alternate to `DO WHILE( 1 = 1 ); )`
  - Delimiting strings with " (double quotation marks)

  - Using underscores to make constants more readable (e.g. "0011_0101"b )
  - Compound assignments (for example, *x += 1;*)
  - RESIGNAL (for more powerful exception handling)
- Many powerful new attributes, including these:
  - ABNORMAL (like volatile in C)
  - NONASSIGNABLE (like const in C)
  - BYVALUE
  - LIMITED ENTRY (for C function pointers)
  - ORDINAL (for strongly-typed enums)
  - RESERVED (for C-like static)
  - UNION
  - UNSIGNED
  - VALUE (for named constants)
  - VARYINGZ (for C-style null-terminated strings)
- Over 100 new built-in functions, including these:
  - HEX and HEXIMAGE (for debugging)
  - PROCNAME and SOURCELINE (for tracing)
  - PLIMOVE, PLIFILL and COMPARE (like memcpy, memset, and memcmp)
  - IAND, IOR, IEOR, and NOT (for bitwise integer operations)
  - COPY (the "nice" REPEAT as defined by ANSI)
- Full z/OS UNIX System Services support, including these
  - Source, object and listing files in HFS
  - I/O to HFS files
- Improved macro facility:
  - Deck file preserves case of source.
  - Macro variables can now be arrays.
  - Many more built-in functions are supported.
  - These statements are supported: ANSWER, ITERATE, LEAVE, SELECT (in open code and in macros), and REPLACE.
  - WHILE, UNTIL, and LOOP keywords are supported in %DO statements.
- Support for multithreading
- Support for UTF-16 Unicode
- Support for IEEE floating-point
- SAX-style XML parsing
- XML generation
- Integrated CICS preprocessor
- Integrated SQL preprocessor

For more information, see the *PL/I Language Reference* and the *Enterprise PL/I for z/OS Programming Guide*.

## Major changes with the new compiler and run time

With Enterprise PL/I, existing PL/I applications are affected by several areas such as removed or changed compiler options, different default compiler options, and restrictions in combining old and new load modules.

The following list of concerns is merely a representative list that reflects what has been important to some customers. It does not indicate what is important to any one individual customer. More details are provided in the rest of this book.

- Enterprise PL/I supports only Language Environment releases currently in support.
- Enterprise PL/I has no support for VM.

- Enterprise PL/I has no support for multitasking (but it does support multithreading).
- Code that is incorrect or invalid (for instance, code that uses uninitialized variables) might not run the same. This might not seem like an important problem, but it has been a significant issue for most of the customers that have migrated.
- You might need to specify some nondefault options to get the most compatible behavior from the compiler and to get the best performance from the compiler.
- Programs might need to be tuned for optimal performance. In particular, the use of the runtime option RPTSTG(ON), while useful when you are tuning performance, is much more costly now to leave on in a production program.
- Recompiling all your PL/I source is recommended. If this is not done, you need to carefully select the compiler options for compiling Enterprise PL/I code that will be mixed with older PL/I objects. You will also need to divide your source into partitions according to how they use FILEs, CONTROLLED variables and conditions. For more information, see "Object and load module considerations" on page 135.

## General conversion tasks

Depending on your needs, you will most likely need to complete one or more of the general conversion tasks.
- Plan your strategy.
- Move to the Language Environment runtime library.
- Recompile your source with Enterprise PL/I.
- Add Enterprise PL/I programs to existing applications.

### Planning your strategy

Before moving to the Language Environment runtime library or recompiling your source programs with Enterprise PL/I, develop a conversion strategy. A thorough strategy will help ensure a smooth transition to the new compiler and run time.

Your conversion strategy might be to move to Language Environment, and then gradually recompile your existing applications with Enterprise PL/I as needed. This book provides separate strategies for moving to the new run time and for recompiling your PL/I source.

If you are not currently on Language Environment and want information about how to plan your move, see Chapter 3, "Planning the move to Language Environment," on page 15.

If you have already moved to Language Environment and want information about moving to the new compiler, see Chapter 4, "Planning to move to the new compiler," on page 23.

### Moving to the Language Environment run time

You can run existing load modules under Language Environment and receive the same results as with pre-Language Environment libraries.

For important compatibility information, see Chapter 5, "Running existing applications under Language Environment," on page 31.

For information about moving applications that are running under older PL/I run time, see Chapter 6, "Considerations before migrating," on page 35.

In almost all cases, you will need to link-edit existing applications with Language Environment or recompile programs with Enterprise PL/I. To determine which programs require link-editing with Language Environment, see Chapter 8, “Link-Edit considerations,” on page 53.

## Recompiling your source with Enterprise PL/I

The new Enterprise PL/I compiler has many powerful, new features of which you might want to take advantage. There are also some differences between this new compiler and the previous PL/I compilers.

To read about the differences between the Enterprise PL/I compiler and the previous PL/I compilers, see Chapter 10, “Understanding the limitations of the new compiler,” on page 63.

To find out more about the new compiler options, see Chapter 11, “Understanding the new compiler's options,” on page 73.

To determine which programs must be changed and then recompiled with Enterprise PL/I, see Chapter 13, “Understanding when working code must be changed,” on page 101 and Chapter 14, “Understanding when working code may need to be changed,” on page 115.

## Adding Enterprise PL/I programs to existing applications

You can create new Enterprise PL/I programs (or recompile existing programs with Enterprise PL/I) and run them with existing applications under Language Environment.

When adding Enterprise PL/I programs to existing applications, you need to be aware of the limitations of mixing old and new PL/I modules. For details, see Chapter 18, “Adding Enterprise PL/I programs to existing PL/I applications,” on page 135.

# Part 2. Conversion Strategies

© Copyright IBM Corp. 1999, 2017

# Chapter 3. Planning the move to Language Environment

This chapter describes a general strategy for moving your runtime environment to Language Environment.

The following tasks are necessary, and are performed roughly in the following order:

- Prepare to move to the Language Environment runtime library.
- Take an inventory of your applications.
- Decide how to phase in Language Environment.
- Set up a regression testing procedure.
- Cut over to production use.

If you have already moved to Language Environment, you do not need to read this chapter and can proceed to reading Chapter 4, "Planning to move to the new compiler," on page 23.

**Important:** Enterprise PL/I programs can only run with the Language Environment element of z/OS Version 1 Release 7, or later.

## Preparing to move to the Language Environment runtime library

In preparing to move to Language Environment, you need to install Language Environment, assess storage requirements, and educate your programmers about Language Environment.

You can do these tasks concurrently.

### Installing Language Environment

This topic lists the documents that you can follow to install Language Environment.

#### Installing Language Environment on z/OS

To install z/OS, including the Language Environment element, see the *z/OS Program Directory*, or consult your *ServerPac: Installing Your Order*.

#### Installing Language Environment on OS/390

To install OS/390, including the Language Environment element, see the *OS/390 Program Directory*, or consult your *ServerPac: Installing Your Order*.

**Important:** To ensure that the Language Environment runtime results are compatible with pre-Language Environment results, you might need to change the default runtime options. See "Differences in runtime options" on page 35 for more details.

### Assessing storage requirements

Storage requirements for Language Environment are larger than for pre-Language Environment PL/I libraries.

© Copyright IBM Corp. 1999, 2017

### DASD storage requirements

During conversion, you will need DASD storage for the Language Environment runtime libraries as well as any pre-Language Environment runtime libraries. When you have finished moving to Language Environment, you will be able to free the storage reserved for the previous PL/I runtime libraries.

To determine the amount of DASD storage required by Language Environment, see the following document:
- On z/OS: *z/OS Program Directory*
- On OS/390: *OS/390 Program Directory*

### Virtual storage requirements

Virtual storage requirements for running PL/I programs with Language Environment will increase over the OS PL/I run time. For both CICS and non-CICS applications, the amount of increase depends on many factors, such as these:

- The values used for the Language Environment runtime storage options: STACK, LIBSTACK, HEAP, ANYHEAP, and BELOWHEAP
- The value used for the Language Environment runtime option ALL31
- Which runtime routines are in the LPA (link pack area) or the ELPA (extended link pack area)

**Note:** You can use the information generated by the Language Environment RPTSTG(ON) runtime option to help tune your storage options during the tuning phase. For details, see the *z/OS Language Environment Programming Reference*. Be sure to reset this option to RPTSTG(OFF) before putting the PL/I application into production, because it will greatly worsen performance.

## Educating your programmers about Language Environment

Before moving to Language Environment, ensure that your application programmers are familiar with the features of Language Environment and the differences between the pre-Language Environment run time and the Language Environment run time.

When your programmers are familiar with Language Environment, they can better prepare for the move to Language Environment. For example, they can assist in taking an inventory of applications.

For information about Enterprise PL/I and Language Environment education available through IBM, you can call 1-800-IBM-TEACH. You can also get information directly from Language Environment publications, from user groups (such as SHARE), and from the Language Environment home page (http://www.ibm.com/s390/le).

# Taking an inventory of your applications

When planning your move to the Language Environment run time, you need to take a comprehensive inventory of the applications that you intend to run on Language Environment.

You must include these in this inventory:
- Vendor tools, packages, and products
- PL/I applications

The Edge Portfolio Analyzer can help you take an inventory of your existing load modules.

**Related information**:

"EDGE Portfolio Analyzer" on page 182

## Vendor tools, packages, and products

Before you can begin moving your run time to Language Environment, you need to know if your vendor tools, packages, and products are designed to run under Language Environment.

Verify these:

- All packages will run under Language Environment, especially if you do not have the source code for them.
- Source code for packages, if you do have the source, is able to be compiled with the Enterprise PL/I compiler.
- Code generators generate source code that is able to be compiled with the Enterprise PL/I compiler.
- Development tools and debuggers that issue their own ESPIE or ESTAE coordinate with Language Environment.

For information about how to obtain a list of vendor products that are enabled for Language Environment, see "Vendor products" on page 182.

## PL/I applications

When taking an inventory of your PL/I applications, you need to gather information about the program attributes that affect moving to Language Environment. This information includes how and what to test and what will affect performance under Language Environment.

For your inventory, determine these:

**For moving your applications to Language Environment:**

- Which programs have been compiled with OS PL/I and which programs have been compiled with PL/I for MVS & VM
- Which programs have been linked with PL/I shared libraries
- Runtime options used (and how specified)
- Which PL/I programs call or are called by assembler programs
- Which PL/I programs are multitasking
- Which PL/I programs use interlanguage communication (COBOL, C, or FORTRAN)
- Which PL/I programs are used under CICS, IMS, DB2, or other subsystems
- Frequency and types of abends expected

**For regression testing:**

- Test cases required and available

**For performance measurements:**

- Amount of storage used
- Frequency of execution of reusable/common modules
- Program execution time (both CPU and elapsed)

## Existing PL/I load modules

Knowing what versions of PL/I load modules you have in your libraries is important in planning your migration to Language Environment. The Edge Portfolio Analyzer can help you take an inventory of your existing load modules.

Another tool that will give you some information about your load modules is the **AMBLIST** utility. AMBLIST is provided by IBM and is usually found in *SYS1.LINKLIST*. Using the LISTIDR control statement, you can obtain listings of selected CSECT identification records (IDR). One of the fields in the IDR contains the name of the translator, or compiler in the case of PL/I, that was used to compile the CSECT. The following example is a sample output from AMBLIST:

```
-------------------------------------------------------------------------
CSECT       TRANSLATOR       VR.MD                YR/DY
MYPLI       5655-H31         32.00                2003/171
MYPLI2      5655-B22         22.01                2001/073
D1          566896201        02.01                1972/271
UNRES       566896201        02.01                1992/034
-------------------------------------------------------------------------
```

Using the text in the *TRANSLATOR* column, you can determine which PL/I compiler created the module. See Table 3 for the Translator field values for the various PL/I compilers.

*Table 3. PL/I compiler IDR values*

| PL/I compiler version | Translator identification record |
|---|---|
| OS PL/I V1 Release 5.1 | 5734-PL1 |
| OS PL/I V2.3 | 5668-910 |
| PL/I for MVS & VM | 5688-235 |
| VisualAge PL/I for OS/390 V2R2 | 5655-B22 |
| Enterprise PL/I for z/OS Version 3 | 5655-H31 |
| Enterprise PL/I for z/OS Version 4 | 5655-W67 |

# Deciding how to phase in Language Environment

When you are ready to use Language Environment in production mode, you need to determine these:

- How to handle multilanguage conversion
- How applications will have access to the library

## Handling multilanguage conversion

If you have PL/I applications with ILC, move them to the Language Environment run time after you have converted each of the languages involved.

For example, move a PL/I-COBOL application to Language Environment after you have moved your PL/I-only and COBOL-only applications to Language Environment.

**Note:** Do not install two different libraries for a given language in LNKLST/LPALST. For example, if you install Language Environment with the PL/I component in LNKLST/LPALST, do not have the OS PL/I library or the PL/I for MVS & VM library installed in LNKLST/LPALST.

After Language Environment has been installed in LNKLST, all of your PL/I applications will run under Language Environment by default.

## Determining how applications will have access to the library

Two general methods are available for moving Language Environment into production: adding Language Environment to the LNKLST/LPALST, or using a STEPLIB approach.

### LNKLST/LPALST

After you add Language Environment to the LNKLST/LPALST, Language Environment is available to all of your applications.

To ensure that all applications are functioning correctly under Language Environment before adding Language Environment to your LNKLST/LPALST, you can temporarily install Language Environment in LNKLST/LPALST or use STEPLIB.

Do not make more than one PL/I runtime library available to your applications at execution time. For example, there should be one and only one PL/I runtime library, such as SCEERUN for Language Environment, in LNKLST. If you have more than one, you will either get hard-to-find errors, or you will have an unused load library in your concatenation. When you add Language Environment to LNKLST/LPALST, remove any other PL/I runtime libraries.

#### Suggestions for temporarily installing in LNKLST/LPALST or using STEPLIB

- Install Language Environment in LNKLST/LPALST on a test or development machine first.
- Use the SETPROG MVS system command to temporarily modify the LNKLST or LPA, without having to perform an IPL of the system. For information about using the SETPROG command, see *z/OS MVS System Commands*, SA22-7627 or *OS/390 MVS System Commands*, GC28-1781.
- Perform an IPL over a weekend and install Language Environment in LNKLST/LPALST. Verify over the weekend that your applications run under Language Environment.

**Note:** Although many elements of z/OS and OS/390 depend on the Language Environment runtime library, neither z/OS nor OS/390 requires Language Environment to be installed in LNKLST. (However, Language Environment must be installed in the same zone as z/OS and OS/390.) If you choose not to place Language Environment in LNKLST, you must use STEPLIB to access Language Environment in the individual z/OS or OS/390 PROCs that require Language Environment. For information about which elements require Language Environment, see *z/OS Program Directory for z/OS Version 1 Release 1* or *OS/390 Program Directory for OS/390 Version 2 Release 10*.

### STEPLIB

You can choose to phase in Language Environment gradually by using the STEPLIB approach. In this way, you phase in one region (CICS or IMS), batch (group of applications), or user (TSO) at a time.

Although using STEPLIB means changing your JCL, a gradual conversion can be easier than moving all of your applications at one time. Also note that when you use STEPLIB, programs will run slower than when they access the runtime library through LNKLST/LPALST, and more virtual storage will be used.

**Note:** If you have multiple processors linked together with channel-to-channel connections, you must treat the entire system as one processor and should convert subsystem by subsystem. In addition to revising your JCL to STEPLIB to the Language Environment run time during initial setup, you might also need to specify CEEDUMP DD if the default allocation for CEEDUMP does not meet your shop's needs. (CEEDUMP is the ddname where Language Environment writes its dump output.)

**Problems with STEPLIB and IMS programs:**

When you use STEPLIB on IMS/DC online to access the Language Environment run time, any Language Environment library routines that you have preloaded will not be loaded into read-only storage.

If your application has an error and overwrites non-application storage, preloaded runtime routines can become corrupted and eventually cause abends when used. At refresh time, these preloaded routines marked reentrant are not refreshed unless loaded from the LPA or the LNKLST/LPALST. Thus, the abends will recur.

**Note:** This is a 20-year-old problem with MVS (OS/390), IMS, and STEPLIB, and is mentioned here because of the proposed STEPLIB approach for gradually moving to Language Environment.

You can use either of the following methods to prevent this problem:
- Install Language Environment into the LNKLST/LPALST.
- Do not preload any runtime routines. (This will slow performance.)

**How to minimize the impact**
- Keep your certification of Language Environment as short as possible. (The sooner it is certified, the sooner you can install in LNKLST/LPALST.)
- Watch for different applications abending in the same region, which would indicate that you need to follow the recovery procedure.

**How to recover**

If you do notice several different applications abending in the same region, stop the region and restart with these IMS commands:
1. Determine the region number by issuing the following command:
   ```
   '/DISPLAY ACTIVE'
   ```
2. Stop the region by issuing the following command:
   ```
   '/STOP REGION region#'
   ```
3. Restart the region by issuing the following command:
   ```
   '/START REGION region-name'
   ```

**STEPLIB example:**

This example shows how to phase in Language Environment by using the STEPLIB method, for an organization that has a central development center (all compiling and linking is done in one location) and separate production sites.

This is a very conservative approach, but it has been used by many customers who require absolutely no disruption in production applications.

1. Certify Language Environment and Enterprise PL/I at the central development center.
   a. Run tests with captured data on your current run time, and save all results.

b. Install Language Environment in a STEPLIB environment. This means that unchanged jobs will run with your current run time, and that some users can use the Language Environment run time by using STEPLIB JCL to access the Language Environment runtime library.
c. Run tests with captured data on the Language Environment run time, using the STEPLIB environment, and compare the results to your current run time. Run parallel tests throughout the certification cycle to ensure that your applications produce the same results when run with Language Environment as they did with your current run time.
d. Compile your test applications with Enterprise PL/I. Use STEPLIB to access the Language Environment runtime library, and rerun the certification tests.

2. Install Language Environment on the central development center's system and test.
   a. Run parallel tests of the nonconverted versions of your existing applications by using STEPLIB to access your current run time.
   b. Run all new applications in the Language Environment runtime environment before releasing to production runs.
3. Prepare a backout strategy by saving the procedures for installing your current run time in case you need to back out the Language Environment run time.
4. Install the Language Environment run time at one production site.
   a. Continue to run parallel tests of the nonconverted versions of your existing applications with your current run time in the STEPLIB environment.
   b. Run the Language Environment run time for one month at this production site.
5. Install the Language Environment run time at all production sites.
   a. Continue to run parallel tests of the nonconverted versions of your existing applications with your current run time in the STEPLIB environment.
   b. Run the Language Environment run time for one month at all production sites.
   c. After one month, delete the entire contents of your current run time library.

Try to move the largest units of work that you can move. Moving entire online regions, applications, or run units at once ensures that interactions between programs within an application or run unit can be tested.

## Setting up a regression testing procedure

Because there are so many possible combinations of coding techniques, the only way to determine whether your applications will run under Language Environment and receive the expected results is to set up a procedure for regression testing.

Move your applications to a test environment, and ensure that you receive the expected results when running under Language Environment.

Regression testing will help you identify these issues:

- Source code changes required as indicated in Chapter 14, “Understanding when working code may need to be changed,” on page 115.
- Storage usage differences between your current run time and the Language Environment run time.
- CPU time differences between your current run time and the Language Environment run time.

During testing, run your existing applications in parallel on both your current run time and under the Language Environment run time to verify that the results are the same. Take performance measurements of your existing applications to compare with Language Environment.

After the program runs correctly, test it separately and also test it with other programs in a run unit. By testing it against a variety of data, you can exercise all the program processing features to help ensure that there are no unexpected execution differences.

Analyze program output and, if the results are not correct, use Debug Tool or the Language Environment dump output to uncover any errors and correct those errors. Make any further changes that you need and then rerun, and, if necessary, continue to debug.

**Note:** Although most applications will run under Language Environment with the same results as on their existing run time, results could differ depending on coding styles, resource utilization, performance, abend behavior, or more strict adherence to IBM conventions in Language Environment. For information about situations where existing code might behave differently, see Chapter 14, "Understanding when working code may need to be changed," on page 115.

## Taking performance measurements

After your applications are running under Language Environment in a test environment, take performance measurements—especially on any time-critical or response-critical applications.

After you have compared runtime performance between Language Environment and your current runtime environment and identified which applications, if any, need performance improvements, you can investigate the methods available to tune your programs and improve performance. For example, you can modify storage values by using the Language Environment runtime options.

## Cutting over to production use

When your testing shows the entire application (or group of applications, if running more than one application in an IMS region, or on TSO) receives the expected results, you can move the entire unit over to production use. However, you must be prepared for unexpected errors.

You can prepare for recovery as follows:

- Under z/OS and OS/390, run the old version as a substitute from the latest productivity checkpoint.
- Under DB2, CICS, and IMS, return to the last commit point and then continue processing from that point by using the unmigrated PL/I program. (For DB2, use an SQL ROLLBACK WORK statement.)
- For batch applications, use your shop's backup and restore facilities to recover.

After you move your existing applications to production use under the Language Environment run time, monitor your applications for a short time to ensure that they continue to work properly. Then, you can run with the confidence that you had in your previous run time.

# Chapter 4. Planning to move to the new compiler

This chapter describes a general strategy for moving your source programs to Enterprise PL/I.

The following tasks are necessary, and are performed roughly in the following order:

1. Prepare to move your source to the new compiler.
2. Take an inventory of your applications.
3. Make application program updates.

Because of the loss of service support for older PL/I compilers, you must eventually move all of your PL/I source programs to the new compiler. Although this is not an immediate requirement, at some future date the older compilers and any supported fixes will not be available. At that point, you will be forced to do a 'quick' migration, and this might be a very inconvenient time.

Before you can move your source programs to the new compiler, you must move your applications to Language Environment.

## Preparing to move your source to the new compiler

In preparing to move your source to the new Enterprise PL/I compiler, you need to install Enterprise PL/I, assess storage requirements, and educate your programmers on new compiler features.

You can do these tasks concurrently.

### Installing Enterprise PL/I

If you have not installed Enterprise PL/I, install the compiler.

For z/OS or OS/390, see the *Program Directory* for your product.

### Assessing storage requirements

Enterprise PL/I object programs can execute in 31-bit or 64-bit addressing mode and can reside above the 16-MB line, which frees storage below the 16-MB line. You can use the freed storage for programs or data that must reside below the 16-MB line.

During the compiler migration, you will need DASD storage for your current PL/I compilers as well as for the Enterprise PL/I compiler. When you have completed the compiler migration, and if you have moved all of your OS PL/I, PL/I for MVS & VM, or VisualAge PL/I programs to Enterprise PL/I, you will be able to free the storage reserved for your current PL/I compiler.

The load module produced from the same source code when compiled with Enterprise PL/I might be larger than when compiled with OS PL/I or PL/I for MVS & VM.

© Copyright IBM Corp. 1999, 2017

## Educating your programmers on new compiler features

Early in the conversion effort, ensure that your application programmers are familiar with the features of Enterprise PL/I and the relationship and interdependencies between Enterprise PL/I, Language Environment, Debug Tool, and any other application productivity tools your shop uses.

In addition, your programmers will need to be familiar with Language Environment runtime options, condition handling, and callable services.

Choosing the right compiler options for your environment is a critical task. The options you choose can vary widely depending on whether you are looking for optimum performance or maximum compatibility with previous versions of PL/I. For more information about choosing compiler options, see Chapter 11, "Understanding the new compiler's options," on page 73.

For information about Enterprise PL/I and Language Environment education available through IBM, you can call 1-800-IBM-TEACH. You can also get information directly from Language Environment publications or technical conferences such as SHARE, or the IBM Technical Interchange.

After your programmers are familiar with Enterprise PL/I features, they can assist you in taking the inventory of programs.

**Related information**:

"Taking an inventory of your applications"
In planning to move your PL/I source programs to Enterprise PL/I, you need to take a comprehensive inventory of applications in which you have programs that you intend to compile with Enterprise PL/I.

# Taking an inventory of your applications

In planning to move your PL/I source programs to Enterprise PL/I, you need to take a comprehensive inventory of applications in which you have programs that you intend to compile with Enterprise PL/I.

By taking an inventory of your applications, you get a detailed picture of the work that is required. You need to take an inventory of the following applications:

- Vendor tools, packages, and products
- PL/I applications

The Edge Portfolio Analyzer can help you take an inventory of your existing load modules.

**Related information**:

"EDGE Portfolio Analyzer" on page 182

## Taking an inventory of vendor tools, packages, and products

Before you can begin moving your source, you need to know if your vendor tools, packages, and products are designed to work with Enterprise PL/I.

- Verify whether PL/I code generators generate PL/I programs that can be compiled with Enterprise PL/I.
- Verify whether PL/I packages can be compiled with Enterprise PL/I.

## Taking an inventory of PL/I applications

When you assess your PL/I applications, you must consider various factors.

For each program in your PL/I applications, include at least the following information in your inventory:

**OS PL/I, PL/I for MVS & VM, and VisualAge PL/I:**

- Programmer responsible
- Compiler used
- Compiler options used, especially CMPAT
- Precompiler options used
- PL/I modules
- INCLUDE library members used in PL/I programs
- Called or FETCHed subprograms
- Calling or FETCHing programs
- Frequency of execution
- Test cases required and available

If you are planning to mix *old* PL/I modules with Enterprise PL/I modules, you will also want the following information in your inventory:

- Use of CONTROLLED variables
- Use of FILE variables and constants

For information about the partitioning of PL/I source programs into units of execution based on the above information, see "Partitioning PL/I source programs into units of execution" on page 138.

## Prioritizing your applications

Using the complete inventory, you can now prioritize the conversion effort.

1. Assign complexity ratings to each item in your completed inventory and determine each program or application's resulting overall complexity rating.
2. Determine the conversion priority of each program or application.

### Determining conversion priority

After you have determined the complexity rating for each program in your inventory, you can make informed decisions about the programs that you want to move to the new Enterprise PL/I compiler, and the order in which you want to move them.

Consider the following when deciding on conversion priorities:

- If your application is at the limits of the storage available below the 16-MB line, it is a prime candidate for conversion to Enterprise PL/I. With z/OS or OS/390 architecture, you can obtain virtual storage constraint relief.
- If the program cannot run under Language Environment, you must convert it.

After you determine the priority of each program that you need to move and the effort required to move those programs, you can decide the order in which you want to convert your applications and programs.

### Setting up move/no move categories

By using the conversion priorities that you have established and taking into account program importance and frequency of execution, you can list most of your programs in the order that you want to convert them to Enterprise PL/I.

There might be some programs that you do not want to convert at all, such as these programs:

- Programs for which you have no source code, that will never need recompilation, and that run correctly under Language Environment.
- Programs of low importance to your organization that run correctly under Language Environment and that would take a very high conversion effort.
- Programs that are being phased out of production.

Note, however, that there might be restrictions on running existing modules mixed with programs that have been moved to the new Enterprise PL/I compiler. See Chapter 18, "Adding Enterprise PL/I programs to existing PL/I applications," on page 135.

## Making application program updates

The following application programming tasks are necessary when you convert your source. You must decide what size your program updates will be.

For example, you can choose to update programs along with your regular maintenance, or you can divide your programs into functional groups and update the source group by group. Some customers have followed the "big bang" process and have made all their program updates at once. However you decide to proceed, these tasks should be performed in roughly the following order:

Save the existing source as a back-up—a benchmark to compare to and a version to recover to—if the converted modules have problems.

1. Update the job and module documentation.

   It is extremely important that all updates be properly documented. PL/I itself is reasonably self-documenting. However, keep a log of the compiler options you specify and the reasons for specifying them. Also document any special system considerations. This is an iterative process and should be performed throughout the conversion programming task.
2. Update the available source code.

   Update the source code manually or with tools that you have developed. For information about when source code must or might need to be changed, see Chapter 13, "Understanding when working code must be changed," on page 101 and Chapter 14, "Understanding when working code may need to be changed," on page 115.
3. Compile, link-edit, and run.

   After the source has been updated, you can process the program as you would for a newly written Enterprise PL/I program. (You need the Language Environment run time installed.) If, during the compile process, you see new messages and wish to understand them better, see Chapter 12, "Understanding the new compiler's messages," on page 87.
4. Debug.

   Analyze program output and, if the results are not correct, use Debug Tool or the Language Environment dump output to uncover any errors.
5. Test the converted programs.

   After moving your source to Enterprise PL/I, set up a procedure for regression testing. Regression testing will help to identify these issues:
   - Code that must be changed
   - File attribute mismatches
   - Storage initialization issues
   - Performance differences

- AMODE issues

After you have established a regression testing procedure, and after your programs run correctly, test them against a variety of data:

- Locally—each program separately
- Globally—programs in a run unit in interaction with each other

In this way, you can exercise all the program processing features to help ensure that there are no unexpected execution differences.

The importance of regression testing cannot be stressed enough. You should consider the move from an *old* PL/I compiler to Enterprise PL/I as a move to a different, though similar, language, and plan your testing accordingly.

6. Repeat when necessary.

   Make any further corrections that you need, and then recompile, relink, rerun, and, if necessary, continue to debug.

7. Cut over to production mode.

   When your testing shows that the entire application receives the expected results, you can move the entire unit over to production mode. (This assumes that your production system is already using the Language Environment run time. If not, use "STEPLIB" on page 19 to access the Language Environment run time.)

   In case of unexpected errors, be prepared for recovery:

   - Under z/OS or OS/390, run the old version as a substitute from the latest productivity checkpoint.
   - Under DB2 and IMS, return to the last commit point, and then continue processing from that point using the unmigrated PL/I program. (For DB2, use an SQL ROLLBACK WORK statement.)
   - For non-CICS applications, use your shop's backup and restore facilities to recover.

8. Run in production mode.

   After cut over, monitor the application for a short time to ensure that you are getting the results expected. After that, your source conversion task is completed.

# Part 3. Moving existing applications to Language Environment

**Important:** This part is intended for users who are migrating from OS PL/I and are not currently on Language Environment. If you are currently using PL/I for MVS & VM, VisualAge PL/I, or Enterprise PL/I, you can go directly to Part 4, "Moving to the new compiler," on page 61.

© Copyright IBM Corp. 1999, 2017

# Chapter 5. Running existing applications under Language Environment

Depending on the characteristics of your applications, you might need to make application modifications and perform some Language Environment customization tasks to ensure that your current applications run under Language Environment.

- Invoke existing applications.
- Link-edit existing applications.

Other factors also apply to ensure compatibility, depending on if you are moving your run time from OS PL/I or PL/I for MVS & VM. For details, see Chapter 6, "Considerations before migrating," on page 35.

## Invoking existing applications

To access Language Environment, you will need to change the procedures you use for invoking applications. The procedures required for non-CICS applications are different than the procedures for CICS applications.

**Note:** Make sure your program names do not begin with `AFH`, `CEE`, `EDC`, `IBM`, `IGZ`, `ILB`, or `FOR`. These prefixes are reserved for Language Environment library routine module names.

### Considerations for non-CICS applications

This section provides details about the changes in invoke procedures required for non-CICS applications.

For more information about how to prepare and run your programs with Language Environment, see the *z/OS Language Environment Programming Guide*.

#### Specifying the correct library

To invoke existing applications when running under Language Environment, you need to replace your current library with the Language Environment SCEERUN library.

#### Specifying alternate DDNAMES (optional)

With Language Environment, you can indicate the destination for Language Environment output by changing the ddname in the MSGFILE runtime option to the ddname you want.

Table 4 lists the default ddnames for Language Environment output.

*Table 4. Specification of new DDNAMEs*

| Output | Default ddname |
|---|---|
| Messages | SYSOUT |
| Runtime options report (RPTOPTS) | SYSOUT |
| Storage reports (RPTSTG) | SYSOUT |
| Dumps | CEEDUMP |

© Copyright IBM Corp. 1999, 2017

All of the ddnames in the above table are dynamically allocated.

You do not need to alter your JCL, CLISTs, or Rexx EXECs to define the ddnames for Language Environment messages, reports, or dumps *unless* the defaults used by Language Environment do not meet the needs of your shop. The Language Environment default destinations are as follows:
- On z/OS and OS/390: SYSOUT=*

## Considerations for CICS applications

To run Language Environment on CICS, you need to perform several required steps.

For details on how to invoke PL/I applications running on CICS under Language Environment, including how to specify the Language Environment runtime library SCEERUN, see the following documents:
- For z/OS, *z/OS Language Environment Customization*
- For OS/390, *OS/390 Language Environment Customization*

### Output differences when using Language Environment on CICS

Under CICS, Language Environment output goes to a transient data queue named CESE. Each record written to the file has a header that includes the terminal ID, the transaction ID, date, and time. The transient data queue (CESE) receives the following types of Language Environment output:
- Messages
- Runtime options report (RPTOPTS)
- Storage reports (RPTSTG)
- Dumps
- PL/I Stream output

# Link-editing existing applications

After determining which of your existing applications either require or will benefit from link-editing with Language Environment, you need to specify the correct library name.

The Language Environment link-edit library is the same for non-CICS applications as for CICS applications.

**Under z/OS and OS/390**
: Include the Language Environment SCEELKED in the SYSLIB concatenation.

**Note:** If you link-edit with the NCAL linkage editor option, ensure that all of the required runtime routines from SCEELKED are included in the load module. Otherwise, unpredictable errors will occur (typically a program check).

There are some names in the SCEELKED library that do not follow IBM naming conventions, and that can conflict with your subprogram names. For example, if you have a statically called subroutine named DUMP and if SCEELKED is ahead of your private subroutine library in the concatenation at link-edit time, your references to DUMP will be resolved in SCEELKED. In this example, the FORTRAN routine AFHUDUMS will be link-edited in, and you could get incorrect results, loss of function, or slower performance as a result. (Another common name is ABORT, which is an entry point in EDC4$05C, a C runtime library routine.)

There are a couple of ways to avoid these problems:

- You can check the names in the SCEELKED data set against the names of your private subroutines. If there are any duplicates, you can rename your private subroutines so that they do not have the same names as the names in the SCEELKED data set.
- Another way is to place your private subroutine libraries before SCEELKED in the SYSLIB concatenation. However, doing this could result in losing function that is available under Language Environment, if your application contains Fortran or C/C++ programs. Changing the name of your subroutine to avoid the conflict with the Language Environment subroutine is preferable to placing your private subroutine libraries ahead of SCEELKED.

To determine which applications require link-editing with Language Environment, see Chapter 8, “Link-Edit considerations,” on page 53.

# Chapter 6. Considerations before migrating

This chapter discusses the functional differences between OS PL/I run time and Language Environment. Consider these differences before migrating your applications to Language Environment.

Language Environment is now part of z/OS and OS/390, and you can start migrating your applications to Language Environment before installing a Language Environment-enabled PL/I compiler such as PL/I for MVS & VM, VisualAge PL/I, or Enterprise PL/I.

## Differences in runtime options

Language Environment runtime options replace PL/I runtime options. Most PL/I runtime options have an equivalent Language Environment runtime option that provides the same function. This section describes differences in the use of runtime options.

You should adapt your applications to allow for the following differences:

### Deleted runtime options

- The OS PL/I COUNT option is ignored.
- The OS PL/I FLOW option is ignored.
- The OS PL/I HEAP option is always in effect. This means that when you allocate storage for BASED and CONTROLLED variables, the storage always comes from HEAP storage. The storage does not come from a PL/I Initial Storage Area (ISA). HEAP(0) is not supported and, if used, is ignored.

### Replaced runtime options

- The Language Environment NATLANG option replaces the OS PL/I LANGUAGE option.
- The Language Environment RPTSTG option replaces the OS PL/I REPORT option.
- The Language Environment TRAP option replaces both OS PL/I SPIE and STAE options. The following table shows how the OS PL/I SPIE and STAE options map to Language Environment's TRAP option:

*Table 5. Mapping of SPIE and STAE Options to the TRAP Option*

| OS PL/I | Language Environment | Action |
|---|---|---|
| SPIE \| NOSPIE | TRAP(ON \| OFF) | If SPIE \| NOSPIE is specified in input, TRAP is set according to the option TRAP(ON) for SPIE, and TRAP(OFF) for NOSPIE. |
| STAE \| NOSTAE | TRAP(ON \| OFF) | If STAE \| NOSTAE is specified in input, TRAP is set according to the option TRAP(ON) for STAE, and TRAP(OFF) for NOSTAE. |

© Copyright IBM Corp. 1999, 2017

*Table 5. Mapping of SPIE and STAE Options to the TRAP Option (continued)*

| OS PL/I | Language Environment | Action |
|---|---|---|
| SPIE STAE or<br><br>SPIE NOSTAE or<br>STAE NOSPIE | TRAP(ON) | If both SPIE\|NOSPIE and STAE\|NOSTAE are specified together in input, TRAP is set according to both options: TRAP(OFF) when both options are negative, and TRAP(ON) otherwise. TRAP(ON) must be in effect for applications to run successfully. |
| NOSPIE NOSTAE | TRAP(OFF) | |

**Note:** Applications performing their own condition management often conflict with Language Environment condition management. See your *z/OS Language Environment Programming Guide* for more information about Language Environment condition handling.

- The Language Environment STACK option replaces both OS PL/I ISASIZE and ISAINC options. You do not need to change and recompile source code that contains ISASIZE and ISAINC. In addition, object modules and/or load modules containing the PLIXOPT string will run under Language Environment with the ISASIZE and ISAINC honored as before.

  STACK(,,ANY) can be used for an OS PL/I application relinked with Language Environment that does not contain any edited stream I/O.

  Your application must run in AMODE(31) to use STACK(,,ANY).

  Under CICS, ALL31(ON) and STACK(,,ANY) are the defaults; however, because STACK(,,BELOW) is required for OS PL/I applications that have not been relinked with Language Environment, you must change the default to STACK(,,BELOW) during installation, or explicitly specify STACK(,,BELOW) for any OS PL/I applications that have not been relinked.

## New runtime options

- The Language Environment ABTERMENC option controls which type of return/abend code your application receives at abnormal termination. ABTERMENC(RETCODE) allows your application to receive a runtime return code, which is equivalent to the way OS PL/I worked.
- The Language Environment ERRCOUNT option limits the number of conditions that are handled at run time. ERRCOUNT(0) specifies that there is no limit, which is equivalent to the way the OS PL/I worked.
- The Language Environment DEPTHCONDLMT option limits the extent to which conditions can be nested. To maintain compatibility, specify DEPTHCONDLMT(0), which means there is an unlimited depth.
- The Language Environment XUFLOW option determines if the UNDERFLOW condition is raised when underflow occurs. XUFLOW(AUTO) preserves PL/I semantics with regard to raising the UNDERFLOW condition.
- The Language Environment ALL31 option controls AMODE switching among library routines. You should set ALL31(0N) if all of your applications are AMODE(31).

When you pass runtime options in the MVS GO step, your runtime options string must end with a slash (/) to distinguish it from a main procedure parameter string. If you omit the slash, the string is passed as the main procedure parameter.

The following runtime options are needed to provide compatibility with OS PL/I:
- ABTERMENC(RETCODE)

- ERRCOUNT(0)
- DEPTHCONDLMT(0)
- STORAGE(,,CLEAR)
- TRAP(ON)
- XUFLOW(AUTO | ON)

Note that before you use the CLEAR suboption of the STORAGE option, you must have the appropriate PTFs for APAR PK02614 installed. Also, the use of the option might not be entirely effective for Enterprise PL/I code: see the discussion later in this book on initializing variables.

For more information about runtime options, see the *z/OS Language Environment Programming Reference*.

For OS PL/I applications, the options specified in the PLIXOPT string is processed as the application-specific options. If you provide the Language Environment CEEUOPT, CEEUOPT is ignored.

If the main load module contains ILC, the PLIXOPT string is ignored. In this case, you must provide CEEUOPT for the application-specific options.

# Differences in condition handling

## Timing differences

PL/I condition handling semantics remain supported under Language Environment; however, the timing of issuing the runtime message for an ERROR condition with respect to the ERROR ON-Unit is different.

It is different in the following way:

- The runtime message for an ERROR condition is issued only if there is no ERROR ON-Unit established, or if the ERROR ON-Unit does not recover from the condition by using a GOTO out of block. You can use a GOTO out of the ERROR ON-Unit to avoid a message for a PL/I ERROR condition.

For PL/I conditions whose implicit action includes issuing a message and raising the ERROR condition, the timing of issuing the message is unchanged.

Table 6 shows when the runtime message for an ERROR condition is issued under OS PL/I with respect to the ERROR On-Unit.

*Table 6. OS PL/I Version 2 Release 3 ERROR ON-Unit and message for an ERROR condition*

| Condition | No ON-Units | ERROR ON-Unit No GOTO | ERROR ON-Unit GOTO |
|---|---|---|---|
| ERROR condition raised[1] | Message | Message prior to ON-unit | Message prior to ON-unit |
| ZERODIVIDE condition raised[2] | Message | Message prior to ON-unit | Message prior to ON-unit |

**Notes:**

[1] Taking the square root of a negative number, data exception, and so on.

[2] With no ZERODIVIDE ON-unit; thus, an implicit action is taken. Message is printed; ERROR condition is raised.

Table 7 shows when the runtime message for an ERROR condition is issued under Language Environment with respect to the ERROR On-Unit.

*Table 7. Language Environment ERROR ON-Unit and message for an ERROR condition*

| Condition | No ON-units | ERROR ON-unit No GOTO | ERROR ON-unit GOTO |
|---|---|---|---|
| ERROR condition raised[1] | Message | Message after ON-unit | No message |
| ZERODIVIDE condition raised[2] | Message | Message prior to ON-unit | Message prior to ON-unit |

**Notes:**

1 Taking the square root of a negative number, data exception, and so on.

2 With no ZERODIVIDE ON-unit; thus, an implicit action is taken. Message is printed; ERROR condition is raised.

The SNAP traceback message produced by ON ERROR SNAP continues to be issued before the ERROR ON-unit receives control. The SNAP traceback message is not identical to the regular ERROR message.

## Differences in performing some fixed decimal divides with large precision

As of Enterprise PL/I V5R1, some fixed decimal divides with large precision are done using the Decimal Floating-Point (DFP) facility. This might cause some ZERODIVIDE exceptions to be reported as INVALIDOP.

## Unhandled condition differences

If your OS PL/I application used to force an abend for an unhandled condition under OS PL/I run time by using OS PL/I assembler user exit IBMBXITA or abend exit IBMBEER, you can use the ABTERMENC(ABEND) option or the CEEBXITA user exit to force an abend under Language Environment.

To force an abend under Language Environment, use one of the following methods:

- Run your application with Language Environment ABTERMENC(ABEND) option. You cannot specify your own abend code with the runtime option.
- Use Language Environment assembler user exit CEEBXITA to force an abend with your own abend code.

## IBMBXITA and IBMBEER differences

Language Environment provides limited support for OS PL/I IBMBXITA and IBMBEER.

See “Considerations for using assembler user exits” on page 166 for details.

## ABEND U4039 differences

An UNHANDLED condition of severity 2 or higher now produces an abend U4039 and optionally a system dump if SYSUDUMP or SYSABEND ddname is present.

If ABTERMENC(RETCODE) is in effect, your application continues the termination with an abend code. If you do not want to see the U4039 abend, Language Environment provides you with the facilities to suppress it.

See "Abnormal Termination Exit" in *z/OS Language Environment Installation and Customization under OS/390* or *z/OS Language Environment Customization* for ways to suppress or change the U4039 abend.

### Severity differences

Severities of some PL/I conditions are different under Language Environment.

See *PL/I Language Reference* for the severities.

## Differences in PLICALLA and PLICALLB support

The interfaces described in this section are not recommended for use under Language Environment.

### PLICALLA considerations

Language Environment provides support for OS PL/I applications that use the PLICALLA entry point. You can also relink your program under Language Environment.

See "OS PL/I Routine Replacement Tool" on page 179 about details on how to relink an application under Language Environment.

You can use PLICALLA as the primary entry point of a FETCHed/CALLed PL/I main load module; however, the calling routine must pass only user arguments which are passed to a subroutine. If runtime options are passed, they are treated as user arguments.

If you develop a new PL/I application and you want the main procedure to receive user arguments like a subroutine, do one of the following:

- Receive control directly from IMS as follows:
  - Using CEESTART or PLISTART as the primary entry point of the load module
  - Specifying the SYSTEM(IMS) compile-time option
- Receive control from an assembler program or a procedure by using a FETCH or CALL statement as follows:
  - Using CEESTART or PLISTART as the primary entry point of the load module
  - Specifying the NOEXECOPS option and the SYSTEM(MVS) compile-time option
  - Specifying either the BYADDR option or the BYVALUE option as necessary and appropriate for the mechanism that the assembler code used to pass the parameters.

Language Environment support of PLICALLA is not available in the following environments:

CICS environment
Preinitialized environment
Nested enclave environment except the PL/I FETCHable main

### PLICALLB considerations

Language Environment provides support for PL/I applications that use the PLICALLB entry point.

The following table shows the PLICALLB parameter mapping between OS PL/I and Language Environment.

*Table 8. Differences in PLICALLB argument list support*

| OS PL/I | Language Environment |
|---|---|
| Address of the length of ISA storage for a nonmultitasking program or the major task in a multitasking program | Mapped to STACK(*init_size*) |
| Address of ISA storage | Used as the initial STACK segment |
| Address of the length of ISA storage for each subtask | Mapped to NONIPTSTACK(*init_size*) |
| Address of the maximum number of concurrent subtasks | Mapped to PLITASKCOUNT(*max_thread*) |
| Address of the options word, in which the following runtime options can be specified: | Supported as follows: |
| REPORT | REPORT mapped to RPTSTG |
| SPIE\|STAE | SPIE\|STAE mapped to TRAP |
| COUNT | COUNT ignored |
| FLOW | FLOW ignored |
| HEAP suboptions | HEAP(,,KEEP\|FREE)\|(,,ANY\|BELOW) |
| TASKHEAP suboptions | THREADHEAP(,,KEEP\|FREE)\|(,,ANY\|BELOW) |
| Address of HEAP storage length for a nonmultitasking program or the major task in a multitasking program | Mapped to HEAP(*init_size*) |
| Address of HEAP storage | Used as the initial HEAP segment |
| Address of HEAP increment for a nonmultitasking program or the major task in a multitasking program | Mapped to HEAP(,*incr_size*) |
| Address of HEAP for subtasks | Mapped to THREADHEAP(,*increment*) |
| Address of ISA increment for a nonmultitasking program or the major task in a multitasking program | Mapped to STACK(,*incr_size*) |
| Address of ISA increment for each subtask (optional for a nontasking application) | Mapped to NONIPTSTACK(,*incr_size*) |

When the above argument list is passed in through the PLICALLB entry point, the argument in the list must either point to an address or be zero. The high-order bit ON in an argument indicates the end of the argument list. R1 must contain the address of the argument list.

With Language Environment, the runtime options passed through the PLICALLB entry point are processed as options specified on invocation of the application and have a higher precedence than CEEUOPT or PLIXOPT options. The assembler user exit cannot be used to alter the runtime options passed through the PLICALLB invocation.

To summarize, the runtime options passed in have the following precedence (from highest to lowest) among Language Environment option specification methods:
1. Options defined at installation time that have the non-overrideable attribute
2. Options specified via the PLICALLB entry point
3. Options specified in the PLIXOPT string or in CEEUOPT
4. Option defaults defined at installation time

The user arguments passed to the PL/I main routine have the following precedence (from highest to lowest):
1. Output from CXIT_PARM or AUE_PARM of the assembler user exit
2. User arguments passed in via the PLICALLB entry

**Note:** The input to CXIT_PARM or AUE_PARM of the assembler user exit is the first argument in the PLICALLB parameter list, that is, the address of a vector of user argument addresses.

Language Environment encourages the use of above-16M-line storage. For compatibility with OS PL/I, Language Environment maps the user-supplied ISA and HEAP storage to STACK and HEAP. With this mapping, however, Language Environment still needs to issue some GETMAINs. Because user-supplied ISA/HEAP storage is usually below the 16M line, below-16M-line storage can be quickly consumed under Language Environment. How Language Environment manages storage is described in the *z/OS Language Environment Programming Guide*.

Language Environment manages storage differently from OS PL/I. It divides storage into more categories than the OS PL/I-supported ISA and HEAP. As a result, mapping the user-supplied OS PL/I ISA or HEAP storage to Language Environment STACK or HEAP storage still requires GETMAINs during run time. Further, Language Environment provides diagnostics to ensure the user-supplied length of ISA or HEAP storage is a multiple of 8 bytes and the address is on a double-word boundary.

Language Environment also ensures that the location of the user-supplied ISA or HEAP storage matches the location specification in the STACK or HEAP runtime option. The user-supplied HEAP storage is ignored when all of the following conditions are true:
1. User-supplied heap storage is above the 16M line.
2. The ANYWHERE suboption of the HEAP option is in effect.
3. The main program is in AMODE(24).

Language Environment allocates below-16M-line storage using the init_sz24 and incr_sz24 suboptions specified in the HEAP option.

Language Environment support of PLICALLB is not available in the following environments:
- CICS
- IMS
- Preinitialized environment
- Nested enclave environment

Only one main routine is supported by the PLICALLB entry point.

## Differences in preinitialization support

Enterprise PL/I does not support the old preinitialization scheme and you might want to consider redesigning your applications.

Language Environment preinitialization services should be used with Enterprise PL/I in the redesigned applications. However, if you want to run your preinitialized programs under Language Environment in the interim while waiting for them to be redesigned, this section describes the differences that you might want to consider before the migration.

The PL/I preinitialized program interface is supported with the following changes:

- The PL/I preinitialized program interface no longer supports the REINITIALIZE request modifier code. If you attempt to use this function, it is diagnosed with the 4093-136 abend code.
- If the routine specified in the CALL request is not statically linked with the assembler driver and it contains ILC, you must ensure the ILC environment is initialized by including the same ILC in the routine specified in the INIT request.
- The TERM request no longer returns 1000 return code as OS PL/I run time did.
- Some of the return and reason codes for the service vector defined by OS PL/I have changed. You must use the return and reason codes for the service vector defined by Language Environment preinitialization services as described in *z/OS Language Environment Programming Reference*.

Language Environment preinitialization services support multiple preinitialization environments under the same TCB. Multiple preinitialization environments under the same TCB is not supported by OS PL/I. To understand how the service works, see "Using Preinitialization Services" in the *z/OS Language Environment Programming Guide*.

## Differences in PLISRTx support

OS PL/I applications containing PLISRTx invocations are supported by Language Environment for OS/390 & VM Release 1.4 or later; however, you must relink your applications if you are using Language Environment Release 1.3 as your run time.

It is a good idea to relink your load module with Language Environment, regardless of the release you are using, for the following reasons:

- Relinking allows the library routine to access the Language Environment-provided DFSORT interface for a more integrated language and sort environment.
- Relinking allows the library routine to replace the 24-bit DFSORT parameter list with the extended 31-bit DFSORT parameter list.

You can relink your OS PL/I PLISRTx applications by using one of the following methods:

- Relinking the object module by using "OS PL/I Object Module Relinking Tool - APAR PN69803" on page 181
- Replacing the library routine by using "OS PL/I Routine Replacement Tool" on page 179
- Relinking the object module directly with Language Environment

## Differences in multitasking support

Enterprise PL/I does not support multitasking.

You must change your applications to use multithreading, or use the PL/I for MVS & VM compiler. Note also that you must use the POSIX(ON) runtime option for Enterprise PL/I multithreading code.

## Differences in OS PL/I shared library support

Enterprise PL/I does not support the old OS PL/I shared library.

## Differences in DATE/TIME built-in functions

The DATETIME and TIME built-in functions now return the number of milliseconds in all environments.

The syntax and description of these built-in functions are in the *PL/I Language Reference*.

## Differences in User Return Code

Language Environment supports a FIXED BIN(31) four-byte user return code value for PLIRETC, PLIRETV, and OPTIONS(RETCODE). This support removes the restriction of maximum value 999. OS PL/I applications must be relinked with Language Environment in order to take advantage of the four-byte user return-code value.

The following table shows how PL/I user return code is supported:

*Table 9. Return Code Behavior under Language Environment*

| Function | OS PL/I load module | OS PL/I object module linked with Language Environment | Enterprise PL/I load module |
|---|---|---|---|
| PLIRETC built-in function | 2-byte value with restriction of 999 | 4-byte value without restriction of 999 | 4-byte value without restriction of 999 |
| PLIRETV built-in function | 2-byte value | Lower 2 bytes of a 4-byte value | 4-byte value |
| RETCODE option | Lower 2 bytes of R15 | Lower 2 bytes of R15 | 2-byte value |

For PLIRETC, relinked OS PL/I load modules can set a 4-byte user return code value.

Under Language Environment, the PL/I user return code is always reset to zero upon return from the PLISRTx invocation. This is not the case with OS PL/I run time.

## Differences in runtime messages

The format and content of runtime messages are different. If you have applications that analyze runtime messages, you must change the applications to allow for the differences.

The following list describes some of the differences:

- The message number in the message prefix is four digits instead of three digits in the form `IBM`*`nnnnx`*, where *nnnn* represents the message number and *x* represents the severity of the message.
- The message severity in the message prefix can be I, W, E, S, or C.
- The message text of some mixed-case English and Japanese messages has been enhanced. The message text of uppercase English messages remains unchanged.

For details, see *Language Environment Debugging Guide and Run-Time Messages*.

Under Language Environment, runtime messages go to the MSGFILE destination specified in the runtime option MSGFILE. The default for MSGFILE destination is

SYSOUT. The user output still goes to SYSPRINT. MSGFILE(SYSPRINT) is supported under Enterprise PL/I only after you have applied the PTFs for the runtime APAR PQ78307. For more information about the MSGFILE option, see the *z/OS Language Environment Programming Guide*.

## Differences in PLIDUMP

PLIDUMP now produces a Language Environment-style dump. The way you use PLIDUMP and the dump output is different.

The following list the differences in the way you use PLIDUMP and the output produced. *Compile unit* refers to the primary entry point of the external procedure, and *Compile unit name* refers to the name of the external procedure.

- The ddname of the dump output file can be CEEDUMP, PLIDUMP, or PL1DUMP. If you do not define one of these files, Language Environment creates a default CEEDUMP file to contain the dump output. The LRECL of the dump output file must be at least 133 bytes to prevent dump records from wrapping, not the 121 bytes required by OS PL/I.
- When you use the hexadecimal (H) option of PLIDUMP, you must specify the ddname CEESNAP for MVS, or the file name CEESNAP for VM; otherwise, the H option is ignored. This data set contains the SNAP dump output.

  When you specify the hexadecimal (H) option under MVS, the output from SNAP includes all system control program information (SDATA=ALL). OS PL/I provides only partial information (SDATA=CB, Q, and TRT).
- When you use ILC, the dump output contains information related to other languages (for example, C/C++ or COBOL).
- The identifier character string is limited to 60 bytes rather than the 90 bytes OS PL/I supported.
- The traceback section lists the compile-unit name associated with each entry point name. When the entry point is a secondary entry point, the primary entry point name associated with the actual entry point is not listed.

  The traceback section also contains offsets relative to the address of the *compile unit*, as well as offsets relative to the address of the real entry point.
- Runtime messages are in a separate section; they are no longer part of the traceback section.
- When you specify the BLOCK (B) option of PLIDUMP, the condition handler save areas appear in the block section of the dump. If you do not specify the BLOCK option of PLIDUMP, the condition handler save areas do not appear in the dump.
- If the program was compiled with the TEST compile-time option, and a begin-block has a label, the begin-block is identified as `Label:BEGIN block.`. Otherwise, the begin-block is identified as `%BLOCK`*nn*, where *nn* is the block count for the begin-block.
- Compiler-generated ILC subroutines now appear in the traceback section. They are identified as the *compile unit name* concatenated with the suffix ILC.
- PL/I library routines that have Language Environment-defined Program Prologue Areas (PPAs) are identified by name in the dump. If the library routines do not have Language Environment PPAs, they are identified as `Library(PL/I)`.
- A HEX dump of STATIC storage is included in the Language Environment-formatted dump. If more than one routine from a compilation unit is on the stack when a dump is produced, static will be dumped only once for that compilation unit.

- Assembler routines that conform to the rules for mimicking PL/I routines are identified by their CSECT names in the dump output.
- PLIDUMP now conforms to National Language Support standards.
- PLIDUMP can supply information across multiple Language Environment enclaves. For example, if an application running in one enclave FETCHes a main procedure (an action that creates another enclave), PLIDUMP contains information about both procedures.

## Differences in storage report

The format, contents, and destination of the runtime storage report have changed.

Language Environment provides storage information equivalent to OS PL/I. The details of the storage report is described in the *z/OS Language Environment Programming Reference*.

The PLIXHD declaration is no longer used to provide the heading for the runtime storage report. Instead, use Language Environment's Callable Service, CEE3RPH, to specify the heading. If you do not use CEE3RPH, the heading includes the main procedure name, date, and time of execution.

## Differences in Interlanguage Communication support

There are some restrictions on support for ILC applications containing OS PL/I and other pre-Language Environment language programs.

The restrictions fall into three groups:

- Fully supported load modules

  Load modules containing OS PL/I and pre-Language Environment C/370 programs are supported under Language Environment.
- Load modules that you must relink

  Load modules containing OS PL/I and VS COBOL II Release 3 (or later) programs must be relinked with Language Environment.

  OS PL/I Version 2 Release 3 provides a migration aid, APARs PN69803 and PN69804, to allow you to do relinking while you are under the OS PL/I Version 2 Release 3 environment. As long as the application is relinked with PN69803 and PN69804 under OS PL/I Version 2 Release 3, the application is supported under Language Environment. See "OS PL/I Object Module Relinking Tool - APAR PN69803" on page 181 for details of the migration aid.
- Unsupported ILC

  ILC between OS PL/I and the following languages is not supported:
  - Fortran (earlier than Language Environment Release 5)
  - OS/VS COBOL
  - VS COBOL II Version 1 Release 2 or earlier releases

For more information, see *Language Environment for OS/390 & VM Writing Interlanguage Communication Applications* or *z/OS Language Environment Writing Interlanguage Communication Applications*.

The behavior of certain applications that use ILC might be different:

- Condition handling might behave differently. The major causes of differences in condition handling are that the INTER option is now ignored, and that PL/I

condition handling facilities can deal with conditions occurring in non-PL/I routines whether or not you specify INTER.

- Under OS PL/I, in applications that used ILC, the environment initialization and termination of the involved languages, including PL/I, could occur multiple times. With Language Environment, there is only one runtime environment, and language-specific initialization and termination occurs only once. Changes in behavior that you might see include opening and closing of files, releasing of allocated storage, and invocation of establish ON-units.

  **Note:** If you have designed your own code to manage your runtime environments, you should remove it as part of your migration efforts. This *private* code is incompatible with Language Environment and will conflict with the runtime environment.

For a complete description of how ILC works in the Language Environment runtime environment, see either *Language Environment for OS/390 & VM Writing Interlanguage Communication Applications* or *z/OS Language Environment Writing Interlanguage Communication Applications*.

## Differences in assembler support

With Language Environment, assembler programs that call a PL/I routine must follow the calling conventions defined by Language Environment (for example, Register 13 pointing to a save area, save areas properly back-chained, and the first word of the save area being zero).

For detailed information, see the *z/OS Language Environment Programming Guide*.

If your OS PL/I main program is called by an assembler program and you want to convert your assembler program to use Language Environment-conforming assembler, you must either recompile your OS PL/I program without OPTIONS(MAIN), or ensure the entry point receiving control is the real entry point of the PL/I program. In either case, the called PL/I program is treated as a subroutine. Either of these programs run under the same Language Environment enclave, where the assembler program is the main program and the called PL/I program is a subroutine.

Your Language Environment-conforming assembler main program must explicitly include the Language Environment-PL/I for MVS & VM signature CSECT, CEESG010, when calling an OS PL/I subroutine to ensure the Language Environment-PL/I-specific runtime environment is initialized. There are three ways Language Environment-conforming assembler can pass control to an OS PL/I subroutine:

1. Branch to a statically linked PL/I subroutine.
2. Use the Language Environment macro CEELOAD and branch to a separately linked PL/I subroutine.
3. Use assembler instructions such as LOAD and BALR to a separately linked PL/I subroutine.

The condition-handling behavior of the LINK from assembler is now clearly defined. For detailed information, see the *z/OS Language Environment Programming Guide*.

### Assembler programs that find the main parameter list

Assembler programs called from PL/I that use the save area back chain to find the parameter list passed to the PL/I main program will no longer work when running on Language Environment. This is because the number of save areas between the assembler program and the save area of the program that invoked the PL/I main program has changed.

Assembler programs that need to find the parameter list passed to the PL/I main program can use the CEEEDB_R13_PARENT field in the Language Environment EDB to obtain the save area address of the program that invoked the PL/I main program.

# Chapter 7. Object module and load module considerations

This chapter describes factors that affect the compatibility of OS PL/I object modules and load modules in Language Environment.

The discussions include both OS PL/I Version 1 and OS PL/I Version 2 object modules and load modules.

All of the library routines in a load module must be from the same release of the runtime library. For example, Language Environment stubs, OS PL/I shared library stubs, and OS PL/I resident library routines cannot exist in the same load module.

To find out what tools are available to help you migrate your libraries to the Language Environment runtime environment, see Appendix A, "Conversion and migration aids," on page 179.

## OS PL/I Version 1 object module and load module compatibility

Language Environment supports object modules and load modules for OS PL/I Version 1 with some restrictions. You can continue to use most of your Version 1 object modules and load modules if you observe the rules described in this section.

If a load module contains an OS PL/I Version 1 object module but is linked with OS PL/I Version 2 resident library, the load module is considered an OS PL/I Version 2 load module and the rules for OS PL/I Version 2 apply. If the load module contains OS PL/I Version 1 Release 1.0 - 2.3 object modules, however, the object module must be recompiled.

If a load module contains the OS PL/I abend exit, IBMBEER, the abend exit is ignored by Language Environment. See "Considerations for using assembler user exits" on page 166 for more information on this topic.

### OS PL/I Version 1 Release 5.1

#### Object module

The object module is supported.

#### Load module not using shared library

- Main load module for MVS non-CICS nonmultitasking

  The OS PL/I bootstrap routine, IBMBPIRA, always linked with a user load module, contains features such as the fast initialization and termination that are not compatible with Language Environment. A sample ZAP, IBMRZAPM, is provided in Language Environment SCEESAMP to help you deactivate those incompatible features. The sample ZAP is described in "OS PL/I Version 1 Release 5.1 main load module ZAP" on page 180.

  ZAPped load modules continue to work under OS PL/I V1.5.1 and V2, as well as Language Environment; however, performance degradation might occur if the original load module contains the fast initialization and termination feature.

  If you do not ZAP your load module, you must do one of the following steps:

  – Relink your object module with Language Environment or OS PL/I Version 2.

© Copyright IBM Corp. 1999, 2017

  - Use the "OS PL/I Routine Replacement Tool" on page 179 to replace the library routines in the load module with Language Environment stubs.
- Main load module for MVS non-CICS multitasking

  The load module is supported.
- Main load module under CICS

  The load module is supported.
- Main load module under VM

  The OS PL/I VM-specific bootstrap routine, DMSIBM, contains features that are not compatible with Language Environment. A sample ZAP, IBMRZAPV, is provided in Language Environment SCEESAMP to help you deactivate the incompatible features. The sample ZAP is described in "OS PL/I Version 1 Release 5.1 main load module ZAP" on page 180.

  The ZAPped load module is supported under Language Environment only. It **no longer** works under OS PL/I Version 1 or Version 2. If you do not ZAP your load module, you must do one of the following steps:
  - Relink your object module with Language Environment or OS PL/I Version 2.
  - Use the "OS PL/I Routine Replacement Tool" on page 179 to replace the library routines in the load module with Language Environment stubs.
- FETCHed subroutine load module

  The load module is supported.

### Load module using the shared library

The load module is supported as long as the OS PL/I V1R5.1 Shared Library was created with all PLRSHR options and the Shared Library, including the multitasking Shared Library, is replaced with Language Environment stubs. The Shared Library needs to be replaced only once during Language Environment installation.

If the Shared Library was not created with all PLRSHR options or the Shared Library is not replaced with Language Environment stubs, the object module must be relinked with Language Environment or OS PL/I Version 2, or you can replace the Shared Library stubs in the load module with Language Environment stubs. After the object module is relinked or the load module is replaced, the OS PL/I Shared Library feature is no longer used.

Note that Enterprise PL/I doesn't support the shared library. If you intend to migrate to Enterprise PL/I, you should stop using the shared library. Under Language Environment, PL/I uses stubs instead of full size resident modules and there is no need to use the shared library.

## OS PL/I Version 1 Release 5

OS PL/I Version 1 Release 5 provides support only for MVS applications. VM and CICS are not supported in Release 5.0.

### Object module

The object module is supported.

### Load module

The load module is not supported, whether or not you use the shared library. You must relink your object module with Language Environment or OS PL/I Version 2, or you can use the "OS PL/I Routine Replacement Tool" on page 179 to replace the library routines in the load module with Language Environment stubs.

## OS PL/I Version 1 Release 3.0 - Release 4.0

### Object module

- Under MVS

  The object module is supported except for the CICS macro language.
- Under VM

  The object module is supported.

### Load module

The load module is not supported, whether or not you use the shared library. You must relink your object module with Language Environment or OS PL/I Version 2, or you can use the "OS PL/I Routine Replacement Tool" on page 179 to replace the library routines in the load module with Language Environment stubs.

## OS PL/I earlier than Version 1 Release 3.0

Object modules or load modules created earlier than Release 3.0 are not supported, and you must recompile your application with a Language Environment-supported PL/I compiler or OS PL/I Version 2.

# OS PL/I Version 2 object module and load module compatibility

In most cases, object modules and load modules created with OS PL/I Version 2 do not require relinking.

Earlier sections of this migration guide discuss OS PL/I features in more detail. Some of these features do require relinking, however, and a few are no longer supported.

Language Environment supports OS PL/I applications that contain the PL/I assembler user exit, IBMxXITA. See "Considerations for using assembler user exits" on page 166 for more information on this topic.

# Summary of support for OS PL/I object modules and load modules

This topic summarizes the PL/I object module and load module support described in this section.

Exceptions to support are shown in the footnotes and are described elsewhere in a related section.

*Table 10. Summary of object module and load module support by Language Environment*

| Support description | V2 | V1R5.1 | V1R5.0 | V1R3.0-V1R4.0 | Earlier than V1R3.0 |
|---|---|---|---|---|---|
| Main load module | Yes[3] | Yes[1,3] | No | No | No |

*Table 10. Summary of object module and load module support by Language Environment (continued)*

| Support description | V2 | V1R5.1 | V1R5.0 | V1R3.0-V1R4.0 | Earlier than V1R3.0 |
|---|---|---|---|---|---|
| Fetched subroutine load module | Yes[3] | Yes[3] | No | No | No |
| Object module | Yes | Yes | Yes | Yes[2] | No |

**Exceptions:**

1. MVS non-CICS nonmultitasking load modules and VM load modules are not supported unless specific action is taken. Review "Load module not using shared library" on page 49 for what action you need to take to enable support for these modules.
2. CICS macro language is not supported as described in "Object module" on page 51 under "OS PL/I Version 1 Release 3.0 - Release 4.0" on page 51.
3. Shared library must be created with all PLRSHR options and must be replaced with Language Environment stubs. Review "Differences in OS PL/I shared library support" on page 42 for actions you need to take to make this happen.

# Chapter 8. Link-Edit considerations

This chapter describes factors you must consider when you link-edit an object module produced by OS PL/I.

Topics discussed include symbol tables and math routines.

## SCEERUN

When you run your OS PL/I application under Language Environment and you use existing JCL, ensure that your STEPLIB or JOBLIB statement includes SCEERUN, unless you TASKLIB or LINKLIB which already includes SCEERUN.

## Symbol table considerations

If you link-edit an object module produced by different releases of PL/I, and if the object module contains symbol tables for external variables, the symbol table that appears in the resultant load module must be the one produced by the most recent release of PL/I.

The compiler produces an object module that contains external symbol table control sections (CSECTs) if your program includes one or more of the following PL/I features for external variables:

- GET DATA statements
- PUT DATA statements
- The TEST(SYM) compile-time option

If your program uses one or more of these features with external variables, you must ensure that the correct symbol table appears in your load module. Place the object module produced by the most recent release of PL/I ahead of all other object modules in the link-edit job stream. If more than one object module produces a symbol table CSECT with the same name, the linkage editor keeps the symbol table CSECT that it encounters first and discards the other symbol tables.

For example, suppose that you link-edit an object module produced by OS PL/I Version 1 Release 5.1 with an object module produced by OS PL/I Version 2 Release 3. Put the object module produced by OS PL/I Version 2 Release 3 ahead of the object module produced by OS PL/I Version 1 Release 5.1 in the link-edit job stream. By doing this, the linkage editor keeps the symbol table produced by OS PL/I Version 2 Release 3 if both object modules produce symbol tables.

## NCAL linkage editor option

Under Language Environment, the NCAL linkage editor option continues to be required when you link-edit your subroutine object modules for future use.

Load modules must not contain Language Environment stubs nor OS PL/I resident library routines.

© Copyright IBM Corp. 1999, 2017

## ENTRY cards

The entry point for a MAIN program compiled with the OS PL/I compiler is PLISTART, but the entry point is CEESTART if the MAIN is compiled with the PL/I for MVS & VM compiler or any later compiler.

If an ENTRY card must be used during the building of a batch application, you should not use CEESTART for a program compiled with the OS PL/I compiler.

## Using OS PL/I math routines

Language Environment provides a set of math routines, including routines for exponentiation. For most commonly used routines, Language Environment produces more accurate results than OS PL/I.

Some of the Language Environment routines also have better performance than OS PL/I. You should use the Language Environment-provided math routines.

Language Environment also provides the OS PL/I math routines to help you to migrate to Language Environment; however, the OS PL/I math routines are provided for compatibility only and will be withdrawn in the future.

If your application must use the OS PL/I math routines under Language Environment, place SIBMMATH in front of SCEELKED when you link-edit your object module.

Enterprise PL/I for z/OS does not provide the math routines that were in OS PL/I.

# Chapter 9. Subsystem considerations

This chapter discusses subsystem-specific considerations that you need to know when you migrate your applications running under CICS, IMS, and DB2.

## CICS considerations

Language Environment provides the same level of OS PL/I object module and load module support as for non-CICS.

See Chapter 7, "Object module and load module considerations," on page 49 for details.

If you are running under CICS Version 3 Release 3, you must ensure that the CICS APAR PN38032 is installed. Without PN38032, your application trying to use Language Environment will receive the APLE abend.

The CICS Storage Protect facility was introduced under CICS 3.3. This provides more data integrity and security for the application program and especially for the entire CICS region. Because of the new feature, you might discover that some of the successfully running OS PL/I applications start to fail with ASRA(0C4) abend and the CICS message DFHSR0622.

If the above problem is happening in your OS PL/I application program, you can try either of the following two methods to fix your problem:

1. Set the CICS system initialization parameter RENTPGM=NOPROTECT. This sets the protection of the user program in user key. The default for RENTPGM is PROTECT.
2. Relink your OS PL/I application program under Language Environment with APAR PN38032 installed.

If the stream output function is used in your OS PL/I CICS application, especially the `PUT DATA;` statement, it might trigger the above error. PL/I stream output function is intended for debugging purposes only. For performance reasons, it is recommended that you do not use it in production programs.

### Updating CICS system definition (CSD) file

When you bring up a CICS region with Language Environment, you must ensure the module names listed in Language Environment CEECCSD are defined in the CSD.

You can locate CEECCSD in SCEESAMP. If you use the CICS Version 4 autoinstall facility, you do not need to define Language Environment modules manually in the CSD.

### Error handling

A diagnostic message is issued only if there is no ERROR ON-unit established in the program, or the ERROR ON-unit does not recover from the condition by using a GOTO out of block.

© Copyright IBM Corp. 1999, 2017

## Restrictions on user-written condition handlers under CICS

Some EXEC CICS commands cannot be used within a user-written condition handler established by CEEHDLR, or within any routine called by the user-written condition handler.

The following EXEC CICS commands are not allowed:

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

All other EXEC CICS commands are allowed within a user-written condition handler. However, they must be coded by using the NOHANDLE option, the RESP option, or the RESP2 option. This prevents additional conditions being raised due to a CICS service failure.

## Macro-level interface

The CICS macro-level interface is not supported.

## FETCHing a PL/I MAIN Procedure

CICS does not support PL/I FETCHing a PL/I MAIN procedure.

## STACK runtime option

This topic gives you an overview of the Language Environment support for the STACK runtime option.

Language Environment supports PL/I for MVS & VM applications that use the runtime option STACK(,,ANY).

Language Environment also supports STACK(,,ANY) for OS PL/I applications that have been relinked with Language Environment as long as the applications meet the following conditions:

- The application contains no edited stream I/O (for example, EDIT was not used in a PUT statement).
- The application specifies AMODE(31).

## Runtime output

When a program is compiled with DISPLAY(STD), all runtime output is transmitted to a CICS transient data queue CESE.

When a program is compiled with DISPLAY(WTO), the DISPLAY output is routed to the CICS JESLOG. All other runtime output is transmitted to a CICS transient data queue CESE.

Language Environment ignores the MSGFILE option under CICS. Figure 1 on page 57 shows format of the output data queue.

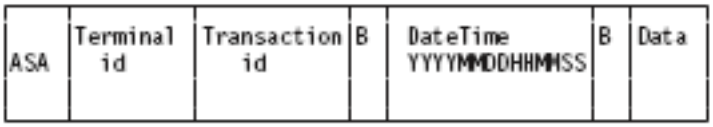



*Figure 1. CESE Output Data Queue*

In addition, PL/I transient queues CPLI and CPLD are no longer used. As a result, you do not need to specify entries for the CPLI and CPLD in the CICS Destination Control Table (DCT).

## Abend codes used by PL/I under CICS

The APL*x* abend codes that were issued under OS PL/I Version 2 are no longer issued. Instead, Language Environment-defined abend codes are issued.

For more information about Language Environment abend codes, see *z/OS Language Environment Run-Time Messages*.

# IMS considerations

Language Environment provides the same level of OS PL/I object and load module support for IMS as for non-IMS.

See Chapter 7, “Object module and load module considerations,” on page 49 for details.

## Interfaces to IMS

Language Environment supports the PLITDLI, ASMTDLI, and EXEC DLI interfaces from a PL/I routine.

## SYSTEM(IMS) compile-time option

The SYSTEM(IMS) option, available in OS PL/I Version 2, was supported for PL/I IMS applications only.

The main procedure of an IMS application must use the POINTER data type for its parameters.

## PLICALLA support in IMS

The OS PL/I PLICALLA entry point is supported under Language Environment; however, it is **not** a recommended interface for IMS and support can be withdrawn at any time.

Instead, use the SYSTEM(IMS) compile-time option and the PLISTART or CEESTART entry point.

Language Environment provides the same support for OS PL/I PLICALLA applications; however, if you recompile your main load module with PL/I for MVS & VM and want to continue to use PLICALLA, you must follow additional rules. See “PLICALLA considerations” on page 39 for details.

## PSB language options supported

Language Environment supports PL/I applications with the following PSBGEN LANG options in the supported releases of IMS:

**IMS/ESA Version 4**

Table 11 shows support for PSB LANG options in IMS/ESA Version 4 Release 1, and later releases.

*Table 11. PSB LANG options for IMS/ESA Version 4 Release 1, and later*

| SYSTEM option | Entry point | LANG= |
|---|---|---|
| IMS | CEESTART, PLISTART | PLI or other values except PASCAL |
| IMS | PLICALLA[1] | PLI |
| Omitted | CEESTART, PLISTART | Illegal |
| Omitted | PLICALLA[1] | PLI |

**Note:**

1. Supported only for compatibility.

## Storage usage considerations

With IMS/ESA Version 3 Release 1, the parameters passed to the IMS interfaces are no longer restricted to the area below the 16M line.

The parameters will most likely be placed above the 16M line if you use the following methods:

- Use the ANYWHERE suboption of the HEAP runtime option. It applies to variables with the CONTROLLED or BASED attribute because their storage is obtained from the heap.
- Use the ANYWHERE suboption of the STACK runtime option. If you relink your OS PL/I application with Language Environment and your application does not use any edited stream I/O, or you recompile your application with PL/I for MVS & VM, you can use STACK(,,ANYWHERE) if your application is AMODE(31). In this case, the variables in automatic storage are placed above the 16M line.
- Place parameters in static storage and make sure the load module attribute used is RMODE(ANY).

## Coordinated condition handling under IMS

Language Environment and IMS condition handling is coordinated, meaning that if a program interrupt or abend occurs when your application is running in an IMS environment, the Language Environment condition manager is informed whether the problem occurred in your application or in IMS.

If the problem occurs in IMS, Language Environment, as well as any invoked HLL-specific condition handler, percolates the condition back to IMS.

With Language Environment runtime option TRAP(ON), Language Environment continues to support coordinated condition handling for the PLITDLI and ASMTDLI interface invoked from a PL/I routine.

With IMS/ESA Version 3 with PTF UN4928 or IMS/ESA Version 4, Language Environment also supports the coordinated condition handling for CEETDLI,

CTDLI from a C routine, CBLTDLI from a COBOL program, AIBTDLI from a PL/I program, and ASMTDLI from a non-PL/I program.

Note that if a program interrupt or abend occurs in your application outside of IMS, or if a software condition of severity 2 or greater is raised outside of IMS, the Language Environment condition manager takes normal condition handling actions described in the *z/OS Language Environment Programming Guide*. In this case, in order to give IMS a chance to do database rollback, you must do one of the following:

- Resolve the error completely so that your application can continue.
- Issue a rollback call to IMS, and then terminate the application.
- Make sure that the application terminates abnormally by using the ABTERMENC(ABEND) runtime option to transform all abnormal terminations into system abends in order to cause IMS rollbacks.
- Make sure that the application terminates abnormally by providing a modified assembler user exit (CEEBXITA) that transforms all abnormal terminations into system abends in order to cause IMS rollbacks.

  The assembler user exit you provide should check the return code and reason code or the CEEAUE_ABTERM bit, and requests an abend by setting the CEEAUE_ABND flag to ON, if appropriate. See the *z/OS Language Environment Programming Guide* for details.

### Performance enhancement with Library Retention (LRR)

If you use LRR, you can get an improvement in performance.

See "Improving CPU utilization" on page 131 for details.

## DB2 considerations

There are no special considerations for using DB2 other than the considerations described in "IMS considerations" on page 57.

# Part 4. Moving to the new compiler

© Copyright IBM Corp. 1999, 2017

# Chapter 10. Understanding the limitations of the new compiler

In addition to not supporting VM, the new compiler has various other limitations that you should understand. This chapter lists and explains these differences.

## Language Environment requirements

Enterprise PL/I V4R2 is supported only on Language Environment for z/OS 1.10 or later.

## Language not supported

The compiler will flag any language that is not supported.

**Multitasking**
The multitasking language supported by the old compilers is not supported by the new compiler.

However, the new compiler does support multithreading. But, in order to use the multithreading facilities, your code must run with the POSIX(ON) option.

For more information about the multithreading statements, see the PL/I Language Reference.

**CHECK**
PL/I for MVS & VM dropped support for the CHECK statement, the CHECK prefix, and the CHECK condition, and the new compiler also does not support these constructs.

**CHARSET(48) and CHARSET(BCD)**
Support for these options were dropped by OS PL/I Version 2. However, there is an IBM-supplied tool that will convert your source.

**EGCS** OS PL/I Version 1 supported EGCS, which was a precursor to the GRAPHIC support in OS PL/I Version 2, which dropped the support for EGCS. The new compiler also does not support EGCS.

**Fortran**
The new compiler does not support the remapping of Fortran parameters. In particular, a two-dimensional array passed from Fortran to PL/I will be seen by PL/I as if it were transposed.

**Invalid code**
The new compiler does not support invalid code even if it was sometimes accepted by the old compiler. For example, the old compiler would allow the CHAR built-in function to be applied to a FILE VARIABLE (even though the old compiler documented that the arguments to the CHAR built-in must have computational type). The new compiler will flag such invalid code with a severe message.

## Language restricted

Except where indicated, the compiler will flag the use of any language that is restricted.

© Copyright IBM Corp. 1999, 2017

## RECORD I/O

RECORD I/O is supported but with restrictions.

- REGIONAL(1) files larger than 2.1 Gigabytes are not supported.
- The EVENT clause on READ/WRITE statements is not supported.
- The UNLOCK statement is not supported.
- The following file attributes are not supported:
  - BACKWARDS
  - EXCLUSIVE
  - TRANSIENT
- The following options of the ENVIRONMENT attribute are not supported, but their use is flagged only under LANGLVL(NOEXT):
  - ADDBUFF
  - ASCII
  - BUFFERS
  - BUFOFF
  - INDEXAREA
  - NCP
  - NOWRITE
  - REGIONAL(2)
  - REGIONAL(3)
  - SIS
  - SKIP
  - TOTAL
  - TP
  - TRKOFL

Note that because the TOTAL option of the ENVIRONMENT attribute is not supported, I/O to files using the TOTAL option will generally not perform as well as under the old compilers.

However, the old implementation of the TOTAL option relied on the compiler generated code knowing both about the layout of the library's I/O control blocks and the layout of the DFSMS control blocks. This meant that if either of these changed, that code would be broken. Consequently, the library code could not be changed and had to continue to use (or fake the use of) a fixed level of the DFSMS control blocks. One consequence of this was that the old library could not use I/O buffers that were above the line - because the compiler generated code in the user's load modules knew they were below the line. (And if Enterprise PL/I were to start implementing the TOTAL option, then it, in turn, would never be able to use buffers that were above the line or above the bar). This rigid interdependence is poor design that hurts even those that do not use the TOTAL option.

Furthermore, the TOTAL option had inherent dangers because the code bypassed all of the library code and hence error handling was problematic at best.

## STREAM I/O

STREAM I/O is supported but with restrictions that apply to PUT/GET DATA statements.

- DEFINED is not supported if both of the following conditions are true:
  - The DEFINED variable is BIT or GRAPHIC.
  - The DEFINED variable has the POSITION attribute.
- DEFINED is not supported if its base variable is an array slice or an array with a different number of dimensions from the defined variable.

## Structure expressions

Structure expressions as arguments are not supported unless there is a parameter description that specifies all constant extents.

However, structure expressions are not supported in any GENERIC reference. Mismatched parameter and argument structures are also not supported in any GENERIC reference.

## Array expressions

An array expression is not allowed as an argument to a user function unless it is an array of scalars of known size. Consequently, any array of scalars of arithmetic type can be passed to a user function, but there might be problems with arrays of varying-length strings.

However, array expressions are not supported in any GENERIC reference. Mismatched parameter and argument arrays are also not supported in any GENERIC reference.

The following example shows a numeric array expression supported in a call:

```
dcl x entry, (y(10),z(10)) fixed bin(31);

call x(y + z);
```

The following unprototyped call would be flagged because it requires a string expression of unknown size:

```
dcl a1 entry;
dcl (b(10),c(10)) char(20) var;

call a1(b || c);
```

However, the following prototyped call would not be flagged:

```
dcl a2 entry(char(30) var);
dcl (b(10),c(10)) char(20) var;

call a2(b || c);
```

## DEFAULT statement

Factored default specifications are not supported.

For example, the following statement is not supported:

```
default ( range(a:h), range(p:z) ) fixed bin;
```

But you could change this statement to the following equivalent and supported statement:

```
default range(a:h) fixed bin, range(p:z) fixed bin;
```

The use of a "(" after the DEFAULT keyword is reserved for the same purpose as under the ANSI standard: after the DEFAULT keyword, the standard allows a parenthesized logical predicate in attributes.

## Extents of automatic variables

An extent of an automatic variable cannot be set by a function nested in the procedure where the automatic variable is declared, or by an entry variable unless the entry variable is declared before the automatic variable.

## Built-in functions

Some built-in functions are supported with exceptions and restrictions.

- The PLITEST built-in function is not supported.
- Pseudovariables permitted in DO loops are restricted to the following:
  - IMAG
  - REAL
  - SUBSTR
  - UNSPEC
- The POLY built-in function has the following restrictions:
  - The first argument must be REAL FLOAT.
  - The second argument must be scalar.
- The COMPLEX pseudovariable is not supported.
- Under the RULES(NOLAXDCL) option, the compiler will flag any declare of a name, such as DISPLAY, as a built-in function if there is no such PL/I built-in function. Even under the more forgiving RULES(LAXDCL) option, the compiler will flag any declare of a name, such as DISPLAY, as a built-in function if there is no such PL/I built-in function if the code attempts to use the name as a built-in function (rather than merely declare it).

## DEFINED BIT aggregates

If a DEFINED variable is a structure or union containing any elements that are UNALIGNED NONVARYING BIT, all array bounds and string lengths in the DEFINED variable must be specified as constants.

The compiler will issue the S-level message IBM1900I when this restriction is violated.

## OPTIONS(REENTRANT)

This option is a part of the OPTIONS for a PROCEDURE or BEGIN statement, but it is ignored.

On the z/OS platform, all programs compiled with the RENT compiler option are reentrant, and other programs are reentrant if they do not alter any static variables (which might require use of the NOWRITABLE compiler option).

## iSUB defining

Support for iSUB defining is limited to arrays of scalars.

## LABEL arrays

The Enterprise PL/I compiler does not require that arrays of statement labels be declared.

If such an array is declared, it should either be declared without a storage class (and without an active DEFAULT statement that would imply a storage class), or declared as STATIC. The old PL/I compiler would require either the former or that the array be declared as AUTOMATIC. Hence if you want your code to be accepted by both compilers, you must declare such an array, but you should declare it neither as AUTOMATIC nor as STATIC.

## DBCS

DBCS can be used only in G and M constants, identifiers, and comments.

G literals can start and end with a DBCS quotation mark followed by either a DBCS G or an SBCS G.

## GRAPHIC and POSITION

In OS PL/I, if a variable is declared with attributes GRAPHIC and POSITION, the POSITION value is interpreted as a count of bytes into the base variable. In Enterprise PL/I, the POSITION value is interpreted as a count of GRAPHIC characters into the base variable.

GRAPHIC and POSITION is similar to BIT and POSITION, where the POSITION value is interpreted as a count of bits into the base variable.

## Macro preprocessor

Suffixes that follow string constants are not replaced by the macro preprocessor—whether or not these are legal PL/I suffixes—unless you insert a delimiter between the ending quotation mark of the string and the first letter of the suffix.

Note that the OS PL/I V2R1 compiler introduced this change, and so this is not a difference between the Enterprise PL/I compiler and either the PL/I for MVS & VM compiler or the OS PL/I V2Rx compilers. This restriction is consequently not flagged.

As an example, consider the following code:

```
%DCL (GX, XX) CHAR;
%GX='||FX';
%XX='||ZZ';
DATA = 'STRING'GX;
DATA = 'STRING'XX;
DATA = 'STRING' GX;
DATA = 'STRING' XX;
```

Under OS PL/I V1, this produces the source:

```
DATA = 'STRING'||FX;
DATA = 'STRING'||ZZ;
DATA = 'STRING' ||FX;
DATA = 'STRING' ||ZZ;
```

Under Enterprise PL/I, it produces the following source:

```
DATA = 'STRING'GX;
DATA = 'STRING'XX;
DATA = 'STRING' ||FX;
DATA = 'STRING' ||ZZ;
```

# Options restricted

Some options or suboptions are ignored.

The following compiler options are restricted:

- INCLUDE

  The NOINCLUDE option is not supported, and the old INCLUDE option is essentially always enabled.

- LANGLVL

  The NOSPROG and SPROG suboptions are not supported - SPROG is always in effect.

  The SAA and SAA2 suboptions are not supported, because the SAA compilers are no longer in service.
- LIST

  The LIST option is supported, but no suboptions of the LIST option are supported - under the new compiler, the psuedo-assembly listing always appears in one column.
- STMT

  The STMT option is supported, but it currently has no effect on the output produced by the LIST, MAP, or OFFSET options.
- SYSTEM

  The CMS and CMSTPL options are not supported (because VM is not supported).

## Options not supported

This topic lists compiler options that are no longer supported by the compiler.

- CONTROL
- COUNT

  The COUNT option is not supported, and it is also not supported by the PL/I for MVS & VM compiler.
- DECK
- ESD

  The ESD option is not supported, but an External Symbol Dictionary is produced if either the LIST or MAP option is in effect.
- FLOW

  The FLOW option is not supported, and it is also not supported by the PL/I for MVS & VM compiler.
- GOSTMT
- HGPR
- IMPRECISE
- LMESSAGE
- SEQUENCE
- SIZE
- SMESSAGE

## Restrictions on other interfaces to the compiler

### Batch compilation

Compilation is not performed in PROCESS-delimited chunks.

This difference has the following consequences:

- Options on later sets of PROCESS statements are ignored.
- One TEXT deck or .o is produced.
- One listing file with one set of messages is produced.
- External variables with the same name must match.

The following example demonstrates a batch compilation. In this case, the mismatches in ***b*** and ***x*** would be flagged by the new compiler only.

```
*process opt(0);

 a: proc;
   dcl b ext entry(1,2 char(2), 2 char(2));
   dcl
     1 x ext,
       2 x1a char(2),
       2 x1b char(2);

   call b(x);
 end;

*process opt(2);

 b: proc(p);
   dcl p pointer;
   dcl
     1 x ext,
       2 x1a bit(16),
       2 x1b bit(16);

 end;
```

You can imitate how batch compilations worked by using a program like the one in Appendix D, "Batch processing sample," on page 189.

## Invoking the compiler from assembler

The new compiler cannot be invoked from the assembler by calling IEL0AA.

The new DD option can be used to specify a list of alternate ddnames for the compiler to use. This provides the key functionality offered by invoking the old compiler from the assembler and should alleviate the need to invoke the compiler from the assembler.

Note also that the compiler can be invoked from an Enterprise PL/I program by using the SYSTEM built-in function.

However, if you must invoke the compiler from the assembler, you can do so as long as your assembler code satisfies these requirements:

- The assembler code must be LE-enabled.
- It must load IBMZPLI using CEEFETCH.
- When it calls IBMZPLI, register 1 must point to the address of a varying string containing the options to be passed in the compilation.

## Compiling under TSO

There is no support for compilations under TSO.

This means that the ISPF 4.5 option is useless with Enterprise PL/I. You should probably disable this option or use it for another purpose.

However, you can invoke the compiler under z/OS UNIX by using the **pli** command. For more information about using the compiler under z/OS UNIX, see the *Enterprise PL/I for z/OS Programming Guide*.

### Specifying INCLUDE data set names

The DD statement corresponding to a %INCLUDE statement should specify the name of the PDS (or PDSE) containing the file to be included, but it must not specify the name of the member file.

For example, to include the file DEBUG from the data set INCLUDE.PLI by using the TEST DD statement, you can use the following the %INCLUDE statement and corresponding DD statement:

| %INCLUDE statement | DD statement |
|---|---|
| `%INCLUDE TEST(DEBUG);` | `TEST  DD DISP=SHR,DSN=INCLUDE.PLI` |

The following DD statement would not be accepted by the new compiler.

```
TEST  DD DISP=SHR,DSN=INCLUDE.PLI(DEBUG)
```

### Defining the SYSLIN data set

Output in the form of one or more object modules from the compiler is stored in the SYSLIN data set if you specify the OBJECT compile-time option. This data set is defined by the DD statement.

The SYSLIN DD must specify a sequential data set, not a PDS or PDSE.

## Compiler time and space requirements

The LRECL for the compiler SYSPRINT data set is 137. The new compiler can require much more time and use much more storage when generating your code.

This is especially true under OPT(2) or OPT(3), in which case some compiles might need a region greater than 100M and might possibly require several minutes to compile. Using the options OPT(2) or OPT(3) without the option DFT(REORDER) can easily lead to this problem and should be avoided.

When the region size is too small for a compile, the compilation will often end with this message:

```
IBM1936I S Invocation of compiler backend ended abnormally.
```

In these situations, you will also find in SYSOUT the following message from the compiler backend:

```
SEVERE ERROR IBM5002: Virtual storage exceeded.
```

If you see this combination of messages, you should either split your program into several smaller programs, or recompile using a larger region size.

The new compiler always runs with ALL31(ON) and with HEAP and STACK obtained from above the 16MB line.

## AMODE(24) restrictions

AMODE(31) and RMODE(ANY) are the default settings for Enterprise PL/I applications. To run an application in AMODE(24), you must complete some tasks.

1. Compile all the PL/I source with the compiler option NORENT.
2. Link with the SIBMAM24 dataset concatenated in front of the SCEELKED data set.

3. Run with the Language Environment runtime option ALL31(OFF) and STACK=(,,BELOW,,,).

**Notes:**

1. There is no support for AMODE(24) in ILC applications, including those involving both Enterprise PL/I and older PL/I. The single exception to this restriction is ILC between Enterprise PL/I and supported High-Level Assembler releases.
2. When you include the SIBMAM24 library in the SYSLIB concatenation for the binder, you are making available library modules that have mode switching capability. However, including the SIBMAM24 library will not by itself cause the resulting load module to be AMODE(24).
3. If you try to run an Enterprise PL/I program in AMODE(24) without linking the SIBMAM24 library before the SCEELKED data set in the SYSLIB concatenation for the binder, your application is invalid and can lead to obscure abends. For example, the first out-of-block GOTO will most likely cause an abend in the library SETJMP routine.

## EXTERNAL names restricted

You must not declare as EXTERNAL a variable whose name, unless it is the name of an IBM-provided function such as PLIXOPT or PLITDLI, begins with any of the following:

- @@
- CEE
- IBM
- PLI

The code generator for the new compiler uses C functions to perform some tasks, particularly under OPT(0). As a result, unless you are intending to invoke the C function directly, you must not declare as EXTERNAL a variable with any of the following names:

- LONGJMP
- MEMCCPY
- MEMCHR
- MEMCMP
- MEMCPY
- MEMMOVE
- MEMSET
- SETJMP
- STRLEN
- SYSTEM

The PLIXHD variable is no longer used as the heading in storage reports. Consequently, the identifier PLIXHD is no longer reserved, and you can declare it and use it as you would declare and use any other variable (as long as you do not declare it EXTERNAL).

## Listing differences

The new compiler produces a listing that is significantly different from the listing produced by the old compiler.

These are some of the differences:

- The LRECL for the listing is 137.

- The first line of the source will not be reflected in the first line of the first page, but the first 43 (or the first 25 if the DBCS option is in effect) characters from that line will be incorporated into the header line of the following pages (except for some parts of the pseudoassembler listing).

## Control block differences

The new compiler uses some different internal control blocks in its generated code from those used by the old compiler.

If you had code that knew the layout and meaning of such control blocks, that code is highly likely not to work now and will probably have to be changed. Some examples where these differences would require code changes:

- Assembler code that "knows" the layout of a PL/I label variable and uses that to try to branch back from assembler into PL/I code
- Assembler code that "knows" the layout of a PL/I file variable and associated file control block and uses that to try to get the DCB for a file

## ISAM support differences

The Enterprise PL/I compiler provides no support for ISAM datasets.

# Chapter 11. Understanding the new compiler's options

This section describes some important compiler options, and after a description of some important defaults, it describes choices you can make to improve compatibility, performance, quality, and test.

If you want to ignore all the discussion in this section and blindly try to maximize compatibility at all costs, follow these suggestions:

1. Use the following default options:
   - BACKREG(5)
   - BIFPREC(15)
   - CMPAT(V2)
   - EXTRN(FULL)
   - LIMITS(EXTNAME(7))
   - NORENT
2. Specify the following additional, nondefault options:
   - COMMON
   - DFT(NOBIN1ARG)
   - DFT(LINKAGE(SYSTEM))
   - DFT(OVERLAP)
   - NOREDUCE
   - NORESEXP
   - RULES(LAXCTL)
   - RULES(NOLAXINOUT NOLAXSEMI)
   - STATIC(FULL)
   - NOWRITABLE(PRV)

The rest of this section describes these and other options in detail so that you can understand the consequences of your choices.

Note that you can also change the IBM defaults for the compiler option by running the job IBMZWIOP when you install the compiler, or by applying a usermod to the module IBMZIOP after you have installed the compiler.

## Understanding the effect of default options on compatibility

This section describes some of the default settings for the compiler options and why you might want to use them.

### BACKREG(5)

The BACKREG option controls the backchain register, which is the register used to pass the address of a parent routine's automatic storage when a nested routine is invoked.

For best compatibility with PL/I for MVS & VM, OS PL/I V2R3 and earlier compilers, you should use BACKREG(5).

All routines that share an ENTRY VARIABLE must be compiled with the same BACKREG option, and it is strongly recommended that all code in application be compiled with the same BACKREG option.

© Copyright IBM Corp. 1999, 2017

Note that code compiled with VisualAge PL/I effectively used the BACKREG(11) option. Code compiled with Enterprise PL/I V3R1 or V3R2 also used the BACKREG(11) option by default.

## BIFPREC(15)

The BIFPREC option controls the precision of the FIXED BIN result returned by various built-in functions.

BIFPREC affects the following built-in functions:
- COUNT
- INDEX
- LENGTH
- LINENO
- ONCOUNT
- PAGENO
- SEARCH
- SEARCHR
- SIGN
- VERIFY
- VERIFYR

The effect of the BIFPREC compiler option is most visible when the result of one of the above built-in functions is passed to an external function that has been declared without a parameter list. For example, consider the following code fragment:

```
dcl parm char(40) var;
dcl funky ext entry( pointer, fixed bin(15) );
dcl beans ext entry;
call beans( addr(parm), verify(parm),' ') );
```

Suppose that the function `beans` actually declares its parameters as POINTER and FIXED BIN(15). If the code above is compiled with the option BIFPREC(31) and if it is run on a big-endian system such as z/OS, the compiler will pass a four-byte integer as the second argument and the second parameter will appear to be zero.

Note that the function `funky` would work on all systems with either option.

The BIFPREC option does not affect the built-in functions DIM, HBOUND, and LBOUND. The CMPAT option determines the precision of the FIXED BIN result returned by these three functions: under CMPAT(V1), these array-handling functions return a FIXED BIN(15) result, while under CMPAT(V2) and CMPAT(LE), they return a FIXED BIN(31) result.

**Related information**:

"CMPAT(V2)"
In Enterprise PL/I V3R2 or later, CMPAT(V2) is the default. This default will ease your migration.

## CMPAT(V2)

In Enterprise PL/I V3R2 or later, CMPAT(V2) is the default. This default will ease your migration.

CMPAT(V2) has the following effects:
- All descriptors will be the same as those generated by the OS PL/I V2R3 and PL/I for MVS & VM compilers.

- Functions returning a string will also use a string locator descriptor (as did the old compilers) for the return value.

CMPAT(V2) does not prevent the use of any new feature of Enterprise PL/I. However, if you have assembler code that examines or builds PL/I descriptors (even if only for strings), the CMPAT(V2) option must be used. For example, DB2 contains such assembler code where it invokes PL/I stored procedures, and therefore, your stored procedures that are written in PL/I must be compiled with CMPAT(V2).

**Note:** As of Enterprise PL/I V5R1, the CMPAT(V1) and CMPAT(LE) options are no longer supported.

## EXTRN(FULL)

By default, the Enterprise PL/I compiler will not discard unused EXTERNAL ENTRYs.

This would cause problems if the EXTRN for a discarded entry was used to force the linker to resolve other references. For example, this would cause problems if your program called the secondary entry point B inside a procedure called A, but contained a declare but no references for A itself.

Note, however, this option will cause EXTRNs to be emitted for all declared external ENTRYs. If you include a file with all the declares potentially used by your code, this can populate your text decks with a large number of EXTRNs.

## LIMITS(EXTNAME(7))

With Enterprise PL/I V3R2, LIMITS(EXTNAME(7)) became the default (previously, LIMITS(EXTNAME(100)) was the default).

This default will ease your migration because this will make the default under the new compiler match what the old compilers always did - they had a limit of 7 characters in an external name and no option that allowed for a higher limit.

Also note that any $n > 8$ in LIMITS(EXTNAME(*n*)) requires the prelinker to be used or your modules to be stored in PDSEs.

Additionally, under LIMITS(EXTNAME(7)) (and under all the old compilers), if an 8-character name is declared as EXTERNAL, the compiler will take the first 4 and last 3 characters to make a 7-character name which it will pass to the linker. However, under LIMITS(EXTNAME(8)), the full 8-character name would be passed to the linker, thereby creating an incompatibility with the code generated by the old compilers.

For example, if the name DEZEMBER is declared as EXTERNAL, under LIMITS(EXTNAME(7)), the linker will see the name DEZEBER, while under LIMITS(EXTNAME(8)), it would see DEZEMBER.

Consequently, for compatibility, do ***not*** use LIMITS(EXTNAME(8)) - use the default LIMITS(EXTNAME(7)).

Finally note that LIMITS(EXTNAME(7)) applies only to PL/I names; assembler and COBOL routines can have 8 characters (exactly as they could with the old compilers).

### NORENT and WRITABLE

In Enterprise PL/I V3R2 or later, NORENT is the default; in earlier releases, RENT was the default. The WRITABLE option is also the default because it gives you the best performance in combination with NORENT.

This default will ease your migration because now, by default, the new compiler, just like the old compilers, will not generate any extra code to make your static variables writeable and still REENTRANT (which is what the RENT option does).

Also note that using the RENT option requires the prelinker to be used or your modules to be stored in PDSEs.

But if you are using the NORENT option, you must also use the NOWRITABLE option if both of the following conditions are true:

1. Your code must be REENTRANT.
2. Your code uses CONTROLLED variables or FILEs.

In Enterprise PL/I V3R4 or later, the NOWRITABLE option has two suboptions, which can also make your code more (or less) compatible:

**FWS** The NOWRITABLE(FWS) option will make your code compatible with the code generated by earlier releases of Enterprise PL/I under the NOWRITABLE option, but it does not allow CONTROLLED variables to be shared between code generated by Enterprise PL/I and code generated by the PL/I for MVS & VM and earlier compilers.

**PRV** The NOWRITABLE(PRV) option will allow code compiled by Enterprise PL/I to share CONTROLLED variables with code compiled by the old PL/I compilers. However, it will also impose the same limits as imposed by those compilers on using CONTROLLED with FETCH.

### SYSTEM

The SYSTEM option generally effects only the way parameters are passed to MAIN. The default is SYSTEM(MVS).

Use this option for all programs except noted as follows:

**SYSTEM(CICS)**
The SYSTEM(CICS) option should be used for all CICS MAIN programs.

**SYSTEM(IMS)**
The SYSTEM(IMS) option should be used only for those IMS MAIN programs to which IMS will pass parameters by value (BYVALUE).

**SYSTEM(OS)**
The SYSTEM(OS) option should be used only for those z/OS UNIX MAIN programs that want to receive the parameter list built by z/OS UNIX. For more discussion of this option, see the *Enterprise PL/I for z/OS Programming Guide*.

## Choosing nondefault options for even more compatibility

This section describes some of the options that you can choose to increase the compatibility between the old and new compilers.

## COMMON

The COMMON option specifies that the compiler should generate CM linkage records for uninitialized EXTERNAL STATIC.

This option can it make it easier to migrate your code if you declare an EXTERNAL STATIC variable in more than one routine but initialize it in only one.

However, note that this option is valid only if both the NORENT and LIMITS(EXTNAME(7)) options are in effect.

## DFT(NOBIN1ARG)

The DFT(NOBIN1ARG) compiler option will generally increase the compatibility of your code, but at the expense of limiting the use of some new function.

For more information about this option, see “One-byte FIXED BIN” on page 118.

## DEFAULT(LINKAGE(SYSTEM))

DFT(LINKAGE(SYSTEM)) causes the parameter list to be built in the same way that it was built by the old compilers (including turning on the high-order bit of the address of the last parameter).

This is not the default linkage used by C or JAVA; their default linkage is what you get with the new compiler's default of DFT(LINKAGE(OPTLINK)). Under the OPTLINK linkage, the last parameter might not even be an address (for instance, if it is a BYVALUE FIXED BIN(31)) and its high-order bit will not turned on even when it is an address. Furthermore, under the OPTLINK linkage, the return value, if any, might be returned in Register 15.

The SYSTEM linkage is assumed for any OPTIONS(COBOL) or OPTIONS(ASM) routine.

When one PL/I routine calls another, it does not matter what linkage they use as long as they match. However, some non-PL/I routines are not declared as OPTIONS(ASM) but do use the SYSTEM linkage. So, for easiest compatibility and migration, you should probably use the DFT(LINKAGE(SYSTEM)) option.

However, if you make the SYSTEM linkage your default, you will need to add OPTIONS( LINKAGE(OPTLINK) ) to the declares of any functions (such as the C library function `fread`) that use that linkage. For example, you would declare `fread` as follows:

```
dcl fread ext entry(...) options( linkage(optlink) );
```

## DFT(OVERLAP)

The DFT(OVERLAP) compiler option will generally increase the compatibility of your code, but at some expense to performance.

For more information about this option, see “Source-target overlap” on page 108.

## NOREDUCE

The NOREDUCE compiler option will slightly increase the compatibility of your code, but at a significant expense to performance.

For more information about this option, see “REDUCE” on page 81.

## NORESEXP

The NORESEXP compiler option will increase the compatibility of your code if your code intentionally forces a ZERODIVIDE condition.

The RESEXP compiler option allows the compiler to evaluate all restricted expressions at compiler time. For example, programs with the following code would fail at compile-time with an S-level message:

```
if somevariable = goodvalue then;
    else
        put skip list( 1 / 0 );
```

Under the NORESEXP compiler option, the compiler will not flag this statement and the ZERODIVIDE condition would be raised at run time, as originally intended.

## RULES(LAXCTL)

The RULES(LAXCTL) compiler option will slightly increase the compatibility of your code, but at a significant expense to performance.

For more information about this option, see "RULES(NOLAXCTL)" on page 82.

## RULES(NOLAXINOUT NOLAXSEMI)

These suboptions of the RULES option have no effect on object compatibility because they do not change the code that is generated.

But if you specify them, the new compiler will act more like the old because it will then issue two messages that the old compiler would issue; in particular, the compiler will issue a W-level message under the following conditions:

**RULES(NOLAXINOUT)**
If it finds a possibly uninitialized scalar passed as an ASSIGNABLE BYADDR parameter

**RULES(NOLAXSEMI)**
If it finds a semicolon inside a comment

## NOWRITABLE

You should choose the NORENT option for the greatest compatibility with your old modules.

The NORENT WRITABLE options allow the compiler to use a static pointer as follows:

- As the base for the stack that tracks a CONTROLLED variable
- As the handle for the storage that represents a FILE

Under the NOWRITABLE option, the compiler will not use a static pointer for either of these purposes, but it has to generate more lines of code to provide the same function.

But you must use the NOWRITABLE option if both of the following conditions are true:

1. Your code must be REENTRANT.
2. Your code uses CONTROLLED variables or FILEs.

However, the NOWRITABLE(FWS) option can have a potentially very strong negative impact on performance, so do not use it if either of the above items does not apply to you.

## Choosing options for improved performance

This section describes some of the options that you can choose to improve the performance of the compiler generated code.

If you want to ignore all the discussion in this section and blindly try to improve performance at all costs, you should follow these suggestions:

1. Use the following default options:
   - REDUCE
   - NORENT
   - RULES(NOLAXCTL)
2. Specify the following additional, nondefault options:
   - ARCH(*n*), *n* is the highest ARCH level supported by your machines
   - BIFPREC(31)
   - DFT(NONASGN)
   - DFT(CONNECTED)
   - DFT(REORDER)
   - DFT(NOOVERLAP)
   - OPT(3)

However, while there are considerations (discussed in this section) that might make you choose not to use all of the above options, unless you use both DFT(REORDER) and at least OPT(2), you will not get good performance from the generated code.

### ARCH

The default in Enterprise PL/I V5R1 is ARCH(8).

Note that Enterprise PL/I requires Language Environment 1.13 (or later) and Language Environment 1.13 requires machines that support ARCH(5).

If you specify an ARCH level that is higher than the lowest level of any machine on which your code runs, your code might cause a program abend with a specification exception on those machines.

If you specify an ARCH value less than 8, the compiler resets it to 8.

### BIFPREC(31)

Specifying the BIFPREC(31) will make your code perform better if you use of the built-in functions to which it applies.

However, the BIFPREC(15) option will give you better compatibility if you use unprototyped ENTRY declarations.

**Related information**:

"BIFPREC(15)" on page 74
The BIFPREC option controls the precision of the FIXED BIN result returned by various built-in functions.

## DEFAULT(NONASGN)

The option DFT(NONASGN) will add the NONASSIGNABLE attribute to all STATIC variables not explicitly declared as ASSIGNABLE.

If your STATIC variables are, in fact, not altered, using this option will allow the compiler to put them in read-only storage and that will give you better performance (particularly if you use the RENT option).

## DEFAULT(CONNECTED)

The DEFAULT(CONNECTED) option sets the default for whether parameters are connected or nonconnected.

Nonconnected arrays are arrays whose elements do not occupy adjacent pieces of storage. Nonconnected arrays are passed by both of these calls:

```
dcl a(3,4) fixed bin;

dcl 1 x(5), 2 y fixed bin, 2 z fixed bin;

call f( a(*,1) );

call f( x.y );
```

The new and old compilers fully support nonconnected arrays, and in fact, the compilers assume that any array parameter is not connected - that there might be other bytes between successive array elements.

This assumption slows down the compiler and requires more code to be generated, which slows down your application.

If you use the new DFT(CONNECTED) compiler option, the compiler will assume that all arrays received are connected and will generate much better code. Hence, if you never pass a discontiguous slice of an array (such as a column), use this option for better performance.

## DEFAULT(REORDER)

To optimize the performance from the compiler generated code, use either OPTIMIZE(2) or OPTIMIZE(3) together with DFT(REORDER).

If you use OPTIMIZE(2) or OPTIMIZE(3) with DFT(ORDER) rather than DFT(REORDER), the runtime performance is less optimal and the compile time might be much longer.

## DEFAULT(NOOVERLAP)

While you might want to use the DFT(OVERLAP) option for compatibility, using the DFT(NOOVERLAP) option will give you much better performance.

For more information about this option, see “Source-target overlap” on page 108.

## OPTIMIZE(2)/OPTIMIZE(3)

To optimize the performance from the compiler generated code, use either OPTIMIZE(2) or OPTIMIZE(3) together with DFT(REORDER).

If you use OPTIMIZE(2) or OPTIMIZE(3) with DFT(ORDER) rather than DFT(REORDER), the runtime performance is less optimal and the compile time might be much longer.

Note that OPT(3) will produce slightly better code than OPT(2), but the compiler will take much longer to compiler programs (especially large programs) under OPT(3) than under OPT(2). For this reason, the compiler maps OPT(TIME) to OPT(2).

## REDUCE

The REDUCE option specifies that the compiler is permitted to reduce an assignment of a null string to a structure into fewer, simpler operations - even if that means padding bytes might be overwritten.

The REDUCE option will cause fewer lines of code to be generated for an assignment of a null string to a structure, and that will usually mean your compilation will be quicker and your code will run much faster. However, padding bytes might be zeroed out.

For instance, in the following structure, there is one byte of padding between `field11` and `field12`:

```
dcl
1 sample ext,
  5 field10 bin fixed(31),
  5 field11 dec fixed(13),
  5 field12 bin fixed(31),
  5 field13 bin fixed(31),
  5 field14 bit(32),
  5 field15 bin fixed(31),
  5 field16 bit(32),
  5 field17 bin fixed(31);
```

Now consider the assignment `sample = '';` .

Under the NOREDUCE option, it will cause eight assignments to be generated, but the padding byte will be unchanged.

However, under REDUCE, the assignment will be reduced to three operations.

With NOREDUCE, you get code that looks like the following:

```
00004C  5810  3056        00015 |      L     r1,=A(@CONSTANT_AREA)(,r3,86)
000050  58E0  305A        00015 |      L     r14,=A(SAMPLE)(,r3,90)
000054  4100  0000        00015 |      LA    r0,0
000058  D206  E004  1000  00015 |      MVC   FIELD11(7,r14,4),+CONSTANT_AREA(r1,0)
00005E  5000  E000        00015 |      ST    r0,<s9:d0:l4>(,r14,0)
000062  5000  E00C        00015 |      ST    r0,<s9:d12:l4>(,r14,12)
000066  5000  E010        00015 |      ST    r0,<s9:d16:l4>(,r14,16)
00006A  5000  E014        00015 |      ST    r0,<s9:d20:l4>(,r14,20)
000072  5000  E018        00015 |      ST    r0,<s9:d24:l4>(,r14,24)
000076  5000  E01C        00015 |      ST    r0,<s9:d28:l4>(,r14,28)
00007A  5000  E020        00015 |      ST    r0,<s9:d32:l4>(,r14,32)
```

But with REDUCE, you get code like the following:

```
00004C  5810  3042        00015 |      L     r1,=A(SAMPLE)(,r3,66)
000050  58E0  3046        00000 |      L     r14,=A(@CONSTANT_AREA)(,r3,70)
000054  D703  1000  1000  00015 |      XC    _shadow1(4,r1,0),_shadow1(r1,0)
00005A  D206  1004  E000  00015 |      MVC   _shadow1(7,r1,4),+CONSTANT_AREA(r14,0)
000060  D717  100C  100C  00015 |      XC    _shadow1(24,r1,12),_shadow1(r1,12)
```

Consequently, for best performance, use the REDUCE compiler option.

## NORENT

While the NORENT option is now one of the compiler defaults because its use increases the compatibility of the object code generated, it might also significantly improve the performance of your code - as long as you do not also use the NOWRITABLE option.

The reasons for this performance improvement are that, under the RENT option, the initialization of every load module takes more time and the code length is longer both for calls and for references to static variables.

However, note that if your code must be REENTRANT and if your code uses CONTROLLED variables or FILEs, you must use either the RENT option or both the NORENT and NOWRITABLE options.

If you use NOWRITABLE with NORENT and your application consists of many programs using CONTROLLED variables, you will get better performance if you use NOWRITABLE(PRV) than if you use NOWRITABLE(FWS). However, using NOWRITABLE(PRV) will also impose all the old limits on using CONTROLLED variables with FETCH.

## RULES(NOLAXCTL)

Using RULES(LAXCTL) can significantly slow the compiler and cause it to generate more copious and more time-consuming code.

For one large customer program, this reduced the compile time by 40% and run time by 50%.

To understand this option, consider the following declaration:

```
DCL
 01 VTAB(*) CTL,              /*                VALOREN-TABLE  */
  02  WA0102  CHAR(26),       /* MUTATIONSDATUM DB2-TIMESTAMP   */
  02  WA0104I BIN FIXED(31),  /* PKEY AKTIONSNR-ID:             */
  02  WA0104K CHAR(1),        /* PKEY VALOREN-KNZ:              */
  02  WA0104V DEC FIXED(15,0), /* PKEY VALORENNR                */
  02  WA0104L BIN FIXED(15),  /* PKEY VV_SEG_LFNR               */
  02  WA0104A CHAR(4);        /* PKEY TERM_ID                   */
```

The bounds of `VTAB` are clearly not known at compile time. But is the length of `WA0104K` really 1 ? The structure would normally be allocated with a statement like one of the following two statements:

```
ALLOC VTAB( 100 );

ALLOC VTAB( N + M );
```

After either of these allocations, `WA0104K` would have length 1.

But the structure could be allocated as follows:

```
ALLOC
  1 VTAB(17),
      2 WA0102,
      2 WA0140I,
      2 WA0104K CHAR(29);
```

But then `WA0104K` has length 29 !

The compiler option RULES(LAXCTL) permits allocations such as the one immediately above despite the fact that the original declared length for the string was a constant. However, using this option will also force the compiler to generate much longer code sequences.

In contrast, the compiler option RULES(NOLAXCTL) assumes that all lengths and bounds that are declared as constant are, in fact, constants, and any ALLOCATE statement that violated this assumption will be flagged with an S-level message `IBM2063`.

Consequently, using this option will not leave you with any runtime surprises, and it will give you much better performance, both at compile time and at run time.

## Choosing options for better quality

This section describes some of the options that you can choose to improve or ensure the quality of your code.

### RULES(NOLAXDCL)

RULES(LAXDCL) causes the compiler to emit only an I-level message for each undeclared variable. But, under RULES(NOLAXDCL), you get an E-level message.

If your code is to have any reasonable quality, you should **always** compile with RULES(NOLAXDCL).

However, when we made this the default on Windows, too many users objected, and it is now *not* the default.

Under RULES(NOLAXDCL), compiling the following code causes the compiler to issue an E-level message saying that *starring_role* is undeclared. The message would alert you that this name is almost certainly a typo, and this is an example of why you want to use this compiler option.

```
x: proc( starting_role ) returns( fixed bin(31) );
  dcl starting_role fixed bin(31);
  return( starring_role + 1 );
end;
```

The option RULES(NOLAXDCL) might also flag "working" code:

```
read_in = fileread( file_in, addr(buffer), stg(buffer) );

if read_in = 0 then
  leave;
```

If *read_in* is undeclared, the code will work; however, *read_in* will have FLOAT as an attribute and that is probably not what you want.

### RULES(NOLAXIF)

Under the compiler option RULES(NOLAXIF), the compiler flags with an E-level message any conditional expression that does not have the attributes BIT(1) NONVARYING.

The expressions in IF, WHILE, UNTIL, and undominated WHEN clauses should have the attributes BIT(1) NONVARYING; however, all the new and old compilers would allow any computational expression in these clauses. For example, you could write the following statement:

```
dcl x fixed bin(31);

if x then ...
```

You might have intended this IF statement to mean the same as the following statement:

```
if x ¬= 0 then
```

But the old and new compilers interpret the statement as:

```
if abs(x) ¬= 0 then
```

It would be much better to code this statement and similar statements so that the conditional expression was a Boolean.

So you can use RULES(NOLAXIF) to enforce this good coding practice.

Under RULES(NOLAXIF), the compiler flags an IF clause consisting of just a reference to a BIT(8) variable, say `Y`. In this case, the generated code treats the expression as true if any of the 8 bits is on, but it might be better to change this IF clause to `Y ^= ''b`.

Under RULES(NOLAXIF), the compiler also flags the assignments of the form:

```
x = y = z
```

Note that the RULES(NOLAXIF) option has effect on the code generated for any statement that it flags.

## RULES(NOLAXLINK)

Specifying the option RULES(LAXLINK) causes the compiler to ignore the LINKAGE and other options specified in the declarations of two ENTRY variables or constants when you assign or compare them.

For example, if you use the RULES(LAXLINK) option, the following incorrect program, which would almost certainly cause an abend if executed, would not be flagged:

```
dcl funtion ext entry returns( char(20) );
dcl subrtn  entry variable;

subrtn = function;

call subrtn;
```

You should use the RULES(NOLAXLINK) option to catch these errors and to enforce basic coding standards.

However, it is probably not a good idea to use the RULES(NOLAXLINK) option in programs containing EXEC CICS statements because the CICS preprocessor generates these declares:

```
DCL DFHEI0   ENTRY VARIABLE INIT(DFHEI01) AUTO;
DCL DFHEI01  ENTRY OPTIONS(INTER ASSEMBLER);
```

Because the variable DFHEI0 is then used in the code that the CICS preprocessor generates for EXEC CICS statements, the compiler will flag, under RULES(NOLAXLINK), the entry DFHEI01, which is declared with OPTIONS(INTER ASSEMBLER), but assigned to DFHEI0, which is declared without any OPTIONS attribute.

## RULES(NOLAXMARGINS)

Under the option RULES(NOLAXMARGINS), any line with non-blanks after the right margin will be flagged. This can help detect problems when code, especially an end-of-comment marker, has been accidentally shifted too far right.

However, because many source files have serial numbers or other data after the right margin, RULES(LAXMARGINS) is the default.

## RULES(LAXSTRZ)

The new compiler will flag any string assignment where the source has a known length, the target has a known maximum length, and the source length is greater than the maximum target length. Unfortunately, this will cause the compiler to flag even those assignments where the trailing bits or characters are "uninteresting". The compiler option RULES(LAXSTRZ) can help reduce this "noise".

Under RULES(LAXSTRZ), no message will be issued in an initial clause or an assignment if any of the following conditions are true:

- A bit variable has a source that is too long but whose excess bits are all 0's.
- A character variable has a source that is too long but whose excess characters are all blanks.

Consequently, under RULES(LAXSTRZ), only the second of the following statements will be flagged:

```
dcl a char(4) init('ok   ');
dcl b char(4) init('error');
```

The default option is RULES(NOLAXSTRZ), but using RULES(LAXSTRZ) might give you better quality by letting you focus on truly problematic assignments.

## RULES(NOMULTICLOSE)

The new and old compilers all allow you to close more than one DO, SELECT, BEGIN, or PROCEDURE group with one END statement, although the new compiler will issue an I-level message. However, closing multiple groups with one END statement is not a good programming practice, and the compiler option RULES(NOMULTICLOSE) allows you to force the compiler to flag such code with an E-level message.

For example, under this option, the compiler will object to the following code:

```
a: do i = 1 to 17;
   b: do j = 1 to 29;
      t = x(i,j);         /* transpose i and j
      x(i,j) = x(j,i);
      x(j,i) = t;
   end b;              /* end of loop */
end a;
```

Note that because the first comment is unclosed, the `end a;` closes both DO loops.

# Choosing options for test

This section describes some of the options that you can choose during development when you want to test your code.

## CHECK(CONFORMANCE)

Specifying the CONFORMANCE suboption of the CHECK option will cause the compiler to generate extra code in the prologue of some procedures to check that the parameters passed match what those procedures expect.

The Programming Guide describes in more detail when this option applies and what it will do, and it can be a very useful tool in development to test your code.

## GONUMBER

When you specify the compiler option GONUMBER, the compiler generates a "statement number table". This table allows the error handler, when it needs to produce a message for a condition that has been raised, to identify where the condition occurred not only by its offset within the containing procedure, but also by its location within your source program.

This extra information can be very useful in helping you analyze errors in your program. If you choose not to use this option, you should probably use the OFFSET option so that the compiler will produce a table that you can use to determine the source statement from the entry offset.

## PREFIX

The PREFIX compiler option allows you to enable PL/I conditions without editing your source.

The following three conditions are particularly useful to enable during test:
- SIZE
- STRINGRANGE
- STRINGSIZE
- SUBSCRIPTRANGE

However, all these conditions will cause the compiler to generate more code and will sometimes cause the performance of the generated code to be significantly worse. Enabling the SIZE condition for an entire compilation can be especially expensive, and it is not recommended that you use this option with production programs.

## TEST

If you are using Debug Tool, you should use the TEST option so that the compiler will generate symbol tables and other information needed for the debugger.

However, do not use this option with production programs.

# Chapter 12. Understanding the new compiler's messages

The new compiler issues many messages that are very similar to those issued by the old compilers. However, it also issues many new messages, some of which can be very important as you migrate to the new compiler. Paying attention to these messages can alert you to possible migration problems. This section explains some of the more important of these messages.

Many of the messages discussed here are I-level and W-level messages, but that does not mean you should ignore them. In fact, these messages are highlighting probable errors in your "working" code.

## IBM1044: one-byte FIXED BIN

This I-level message alerts you to a difference between Enterprise PL/I and the old PL/I compilers. The message produced by the new compiler looks like the following:

```
IBM1044I I FIXED BINARY with precision 7 or less is mapped to 1 byte.
```

This is a feature of Enterprise PL/I: it supports one-byte integers. This is a very useful feature, especially when data are being exchanged with C or JAVA.

However, this is also a difference between the old and the new compilers: under the old compilers, a variable declared as, for example, FIXED BIN(7), would have been allocated 2 bytes, which meant that unless SIZE was enabled, it could have assumed values in the range of -32768 through 32767 rather than the much smaller range of -128 through 127 allowed by a one-byte integer.

Unless you are intentionally exploiting this new feature, you should probably use the EXIT option to increase the severity of this message, and then change all code that produces the message.

## IBM1053: scaled FIXED BIN evaluation

When compiling some of your code, you might see the following message:

```
IBM1053I I Scaled FIXED operation evaluated as FIXED DECIMAL.
```

For an example of the code that will produce this message, and for an explanation of what to do, see “Arithmetic built-in functions with scale factors and FIXED BIN” on page 120.

## IBM1065: imprecise float constants

This I-level message alerts you to a potential source of problems with Enterprise PL/I:

```
IBM1065I I Float constant ... would be more precise if specified as a long float.
```

Floating point constants can represent binary fractions (such as .1E0b and .001E0b) very well, but in general, they cannot represent decimal fractions (such as .1E0 and 3.1415E0) precisely. This message alerts you to the fact that if such fractions were specified as long-floating point (for instance, by specifying more than 6 decimal digits), the fraction would be more precisely represented.

© Copyright IBM Corp. 1999, 2017

## IBM1091: FIXED BIN precision warning

This W-level message alerts you to what is at best poor programming and at worst a source of problems. The message produced by the new compiler looks like the following:

```
IBM1091I W FIXED BIN precision less than storage allows.
```

The Enterprise PL/I compiler will produce this message whenever a SIGNED FIXED BIN variable is declared with a precision other than 7, 15, 31, or 63, or whenever an UNSIGNED FIXED BIN variable is declared with a precision other than 8, 16, 32, or 64. The compiler will also issue this message if a built-in function such as BIN, ADD, DIVIDE, and so on has a FIXED BIN result but specifies one of the above precisions.

For example, if you declare a variable as FIXED BIN(5), the compiler will flag the declare, and you should probably change the declare to the intended FIXED BIN(15).

## IBM1099: mixed FIXED

When compiling some of your code, you might see messages such as this:

```
IBM1099I W  FIXED DEC(7,2) operand will be converted to FIXED
            BIN(25,7). Significant digits may be lost.
```

The attributes in your messages might vary, but here is a sample piece of code that would produce exactly this message:

```
DCL
   1 REC_OUT,
      03  AVAIL        FIXED BIN(31),
      03  TOTAL_SPARE  FIXED DECIMAL(7,2),
      03  WORK_TOTAL   FIXED DECIMAL(7,2);

 AVAIL = 17;
 WORK_TOTAL = 12.2;

 TOTAL_SPARE = AVAIL + WORK_TOTAL;
```

The new and old compilers implement the final assignment in exactly the same way and both would leave `TOTAL_SPARE` with the value of 29.19 (not 29.20 as you might expect). However, only the new compiler issues a message to tell you that you might want to examine this statement more closely.

To understand what this message is telling you and why the result of the statement above is correct when it seems to be wrong, you need to recall these PL/I rules for arithmetic operations other than exponentiation:

1. If either operand is FLOAT, any FIXED is converted to FLOAT.
2. If either operand is BINARY, any DECIMAL is converted to BINARY.
3. DECIMAL(p,q) is converted to BINARY( 1+log(10)*p, log(10)*q ).

So, adding the FIXED BIN(31,0) variable *AVAIL* to the FIXED DEC(7,2) variable *WORK_TOTAL* will force, by the above rules, the DEC(7,2) operand to be converted to BIN(25,7).

But 12.20 cannot be exactly represented as a BIN(25,7), and is actually converted as follows:

```
( bin(12.20,31,0) * 2**7 ) /  10**2
```

This yields a value that is approximately 12.195.

Then adding 17 and converting back results in 29.19.

The compiler behavior in all of the above is correct, but perhaps not what you want. If it is, in fact, not what you want, you could force the operation to be evaluated in DECIMAL by either applying the DECIMAL built-in function to the BINARY operand or by specifying the new compiler option RULES(ANS).

Under RULES(ANS), scaled FIXED BIN is not permitted and the conversion rules are more what a naive user might expect:

```
if both operands are FIXED, then

  if either has a non-zero scale, any BIN becomes DEC
```

So when adding BIN(31,0) to DEC(7,2), the BIN(31,0) is converted to DEC(10,0), and nothing is lost.

The same considerations as detailed above also apply to the following code fragment:

```
dcl a dec fixed(15,3) init(2500000);
dcl zero bin fixed(31) init(0);
if (a ¬= zero) then
   put skip edit('dec fixed ¬= Zero')(a);
else
   put skip edit('dec fixed  = Zero')(a);
```

The DEC(15,3) operand gets converted to BIN(31,10).

But BIN(31,10) can hold up to 2**21 or 2_097_152.

Consequently, this conversion cannot be made successfully, and the SIZE condition would be raised if it were enabled. When the SIZE condition is not enabled, this code is in error and the CVB instruction that is generated to perform the conversion raises the ZERODIVIDE condition.

Again, the new compiler issues an appropriate message:

```
IBM1099I W    FIXED DEC(15,3) operand will be converted
              to FIXED BIN(31,10). Significant digits may be lost.
```

Finally, using RULES(ANS) compiler option or applying the DEC built-in function to the BINARY operand would again fix this code.

## IBM1181: miscoded DO loops

Some programs that have always compiled cleanly might now produce this message:

```
IBM1181I W   A WHILE or UNTIL option at the end of a series DO specifications
             applies only to the last specification.
```

This message is produced for statements such as the following:

```
DO I = 1, 2 WHILE( X = 'Z' );
```

This message says that this DO-loop will be executed once with I equal to 1 (whether or not X = 'Z' is true), and then, if X = 'Z' is true, with I equal to 2. This DO statement is not the same as this statement (although this is probably what the author intended):

```
DO I = 1 WHILE( X = 'Z' ), 2 WHILE( X = 'Z' )
```

If this was what you intended, it would probably be best to code the statement as follows:

```
DO I = 1 TO 2 WHILE( X = 'Z');
```

And, if you did want to test if X = 'Z' only before the second iteration of the DO-group, it would be best to code the statement as follows:

```
DO I = 1 TO 2 UNTIL( X ^= 'Z' );
```

## IBM1206: misuse of BIT operators

This W-level message alerts you to likely coding errors. The message produced by the new compiler looks like the following:

```
IBM1206I W BIT operators should be applied only to BIT operands.
```

The code generated by the new compiler for statements where it produces this message is the same as the code generated by the old compiler, although the latter issued no warning message.

As examples of where this message could arise and the likely coding errors that led to them, consider this code:

```
dcl (x,y) fixed bin;

if x = ¬y then
  ...

if x ¬ y then
  ...
```

In the first IF statement, the bit prefix negation operator will be applied to the FIXED BIN variable *y*, and most likely that is not what was meant. Similarly, in the second IF statement, the bit infix exclusive-or operator will be applied to the FIXED BIN variables *x* and *y*, and most likely that is again not what was meant. In fact, both statements most likely contain typographical errors and were meant to test if the variables *x* and *y* were unequal.

Note also that if the bitwise operations were really intended here, it would probably be best to use the BIT built-in function (or possibly the INOT and IEOR built-in functions) to make that clear.

## IBM1208: incompletely initialized arrays

When compiling some of your code, you might also see the following message:

```
IBM1208I W   INITIAL list for the array WPPXS_TAB
                  contains only one item.
```

For instance, this message would be produced if the variable in the following declaration is used in your program:

```
DCL WPPXS_TAB(15) CHAR(3500) INIT((15)' ');
```

The *INIT( (15)' ' )* attribute does not specify 15 instances of a string consisting of one blank. The *15* is a string repetition factor, and so this INIT clause specifies only one string (of 15 blanks).

To initialize the whole array to blanks, you should code as follows:

```
DCL WPPXS_TAB(15) CHAR(3500) INIT( (*) ('') );
```

The new compiler will also produce this message for many other similar declares, such as these:

```
DCL LISTE(4,60:73)   CHAR(50)  INIT('');
DCL SPRACH_TAB(4)    CHAR(15)  INIT('');
```

Finally, if this array is part of a structure, the compiler will flag any subsequent occurrences of this problem in that structure with message IBM2603. Hence, you can use the EXIT option to reduce the number of times this problem is flagged to once per structure.

## IBM1215: incomplete declares

When compiling some of your old code, you might additionally see a message such as the following:

```
IBM1215I W  The variable I is declared without any data attributes.
```

The new compiler would issue this message, for example, for this declaration:

```
DCL  I, J FIXED BIN;
```

While the older compilers would produce no message for this declare, the new compiler issues the message above because this declare is not equivalent to `DCL (I,J) FIXED BIN;`: it is actually equivalent to `DCL I; DCL J FIXED BIN;`.

## IBM1216: incorrect structure declares

Similarly, consider the following declare:

```
DCL
  1 S,
    2 A    CHAR(10),
    2 B,
      2 C  CHAR(3),
      2 D  CHAR(3);
```

The older compilers would produce no message for this declare. However, the new compiler will produce the message:

```
IBM1216I W  The structure member B is declared without any data
                  attributes. A level number may be incorrect.
```

This message is pointing at some probable errors in the declare, namely that *C* and *D* should be declared at level 3 rather than level 2. But given the declare above, because they are at the same structure level as *B*, *B* is not their parent and gets the default attributes of FLOAT! This is almost certainly not what you intended, and this new message is directing your attention to this likely problem.

The compiler also issues this message for the following code:

```
DCL PARDIASE CHAR (20);
DCL 1 INDIASE1 BASED (PTPDIASE),
     2 C1CODIA CHAR      (1),
     2 C1FECDI DEC FIXED (9),
     2 C1DIADI CHAR (9),
     2 C1ABRDI CHAR (3),
     2 C1RESDI;
DCL PTPDIASE POINTER;
PTPDIASE = ADDR (PARDIASE);
. . .
INDIASE1 = '';
```

The message flags the fact that the variable C1RESDI is declared with out any data attributes. Hence it gets the default attributes of FLOAT DEC(6), and that means that the structure INDIASE1 then occupies 22 bytes. But because the structure is based on a pointer that has been assigned the address of a CHAR(20) field, the assignment `INDIASE1 = '';` will blank out 2 bytes of storage used by some other variable. In the code, this led to an abend in a library routine. Note that if C1RESDI had been declared as CHAR(2) or even as CHAR(0) (and CHAR(0) is legitimate PL/I), there would have been no problem.

So, even though this message, like many of the other messages discussed in this section, is not an E-level message, it would be very good to change your code so that your compilation is free of this message.

## IBM1220: pointless comparisons

When compiling some of your code, you might also see the following message:

```
IBM1220I W  Result of comparison is always constant
```

For example, the following code will cause the new compiler to produce this message:

```
DCL ZWSTRING     CHAR(80);
DCL ZWSTRING2    CHAR(8);

ZWSTRING = 'E R R O R';
.....
IF ZWSTRING2 = 'E R R O R' THEN
```

This message is produced because 'E R R O R' is CHAR(9) with its last character a nonblank, and hence it could never equal a CHAR(8) field.

Any code that produces this message is problematic and should be closely examined. In fact, ignoring this message when it points at a DO-loop statement means that your code could go into an infinite loop. For example, the compiler would produce this message for all three of these executable statements, and in the last case, the loop would run endlessly unless exited with a LEAVE statement:

```
DCL ZZ9 PIC'ZZ9';
DCL N  FIXED BIN(15);

IF ZZ9 < 0 THEN  ...
IF ZZ9 <= 999 THEN ...
DO N = 1 TO 32768;  ...; END;
```

Note that if you have a loop that you want to run "endlessly" until exited by a LEAVE (or GOTO) statement, it would be best to code that loop statement using DO FOREVER.

## IBM1927: SIZE condition

When compiling some of your "working" code, you might also see a message such as the following:

```
IBM1927I S  SIZE condition raised by attempt to convert
            32777 to SIGNED FIXED BIN(15)
```

For example, the following code will produce this message:

```
DCL I   BIN FIXED(15);

DCL
```

```
 1 S,
   2 A    CHAR(10),
   2 B    CHAR(32767);

I = STG(S);
```

Note the following in the assignment above:

- The source `STG(S)` is equal to 32777.
- The target `I` has the attributes FIXED BIN(15).

The old compilers would have issued no message.

The new compiler is telling you that 32777 is too large to be converted to FIXED BIN(15) (because a FIXED BIN(15) variable cannot hold values larger than 32767).

This message points to a problem you should not ignore, and because it is an S-level message, you will be forced to change your code.

## IBM1948: restricted expression evaluation

When compiling some of your code, you might see the following message:

```
IBM1948I S ZERODIVIDE condition with ONCODE=320 raised while
           evaluating restricted expression.
```

For an example of the code that will produce this message, and for an explanation of what to do, see "NORESEXP" on page 78.

## IBM2063: invalid ALLOCATE

When compiling some of your code, you might see the following message:

```
IBM2063I S Specification of extent for variable-name in
           ALLOCATE statement is invalid since it was declared
           with a constant extent.
```

For an example of the code that would produce this message, and for an explanation of what to do, see "RULES(NOLAXCTL)" on page 82.

## IBM2402: storage overlay

This message alerts you to a potentially important coding error:

```
IBM2402I E  <variable x> is declared as BASED on the ADDR of <variable y>,
            but <variable x> requires more storage than <variable y>.
```

The importance of this message depends on how the variables are used in your program. For instance, if X is a 100-byte structure and Y is declared as CHAR(200) BASED(ADDR(X)), the compiler issues this message; note that, in this example, the message is issued only when X is not subscripted. If your program also contains the statement `Y = ''`, you have a severe problem (because that the assignment wipes out 100 bytes of storage that the compiler is likely to be using for other purposes). You must correct this kind of problem.

However, your program might use Y only in the statements such as these:

- SUBSTR(Y,1,STG(X)) = '';
- SUBSTR(Y,1,STG(X)) = LOW(STG(X));

In this case, your code does not need to be changed.

However, in this case, you could change the declare of Y to eliminate these messages: if you declare Y after X, you could then declare Y as CHAR(STG(X)) BASED(ADDR(X)). This would eliminate this occurrence of the message without you having to make any changes to your code. But, if you wanted, you could also then simplify the above assignment statements as follows:

- Y = '';
- Y = LOW(STG(X));

## IBM2409: RETURN; in a function

This message alerts you what is probably a coding error:

```
IBM2409I E  RETURN statement without an expression is invalid inside a
            subprocedure that specified the RETURNS attribute.
```

The compiler issues this message when it finds a `RETURN;` statement inside a function (for example, inside a PROCEDURE that has the RETURNS options). If this statement were executed, the caller of the function would, if it used the result of the function, use an uninitialized value, and that could have unpredictable and arbitrarily bad consequences.

Code that produces this message should be corrected.

## IBM2410: No RETURN in a function

This message alerts you to another coding error:

```
IBM2410I E  Function F contains no valid RETURN statement.
```

The compiler issues this message when it finds no RETURN statement inside a function (for example, inside a PROCEDURE that has the RETURNS options). If this function were called, the caller of the function would, if it used the result of the function, use an uninitialized value, and that could have unpredictable and arbitrarily bad consequences.

Code that produces this message should be corrected.

## IBM2412: missing RETURNS option

This message alerts you to a related coding error:

```
IBM2412I E  Procedure has no RETURNS attribute, but contains a RETURN statement.
            A RETURNS attribute will be assumed.
```

This is the inverse problem to the problem that message IBM2409 flags: here there is a RETURN statement with an expression, but it is inside a PROCEDURE that is a subroutine rather than a function (for example, inside a PROCEDURE that does not have the RETURNS options). The compiler will assume a RETURNS attribute for the PROCEDURE, but these assumed attributes might not be what you intended. More importantly, if the invoker of this routine invoked it through a CALL statement, and if this RETURN statement were executed, it would assign the return value to storage allocated for other purposes, and that could have unpredictable and arbitrarily bad consequences.

Code that produces this message should be corrected.

## IBM2421: CLOSE in ENDFILE

This message alerts you to a subtle coding error:

```
IBM2421I E  A file should not be closed in its ENDFILE block.
```

While it might be tempting to close a file in your ENDFILE block for that file, you should not do this because internal library errors might occur. Instead, it is best to write your ENDFILE block so that it does nothing more than set a flag that will be tested after the READ or GET statement for the file. You should then close the file in the mainline code when it sees that this flag has been turned on.

Code that produces this message should be corrected.

## IBM2610: precision interpretation

This message alerts you to a possible misunderstanding of PL/I rules and, consequently, a possible source of problems:

```
IBM2610I W  One argument to BUILTIN X is FIXED DEC while the other is FIXED BIN
            or FLOAT. Compiler will not interpret precision as FIXED DEC.
```

This message applies to the MULTIPLY, DIVIDE, ADD, and SUBSTRACT built-in functions. You are most likely to see it under the following circumstances:

1. The built-in function has either 3 arguments or 4 arguments, the last of which is zero.
2. One argument is FIXED DEC(p1,0).
3. One argument is FIXED BIN(p2,0).

If, for instance, X is FIXED BIN(31), the compiler would flag the expression MULTIPLY( X, 1000, 15 ) with this message (even if this expression is assigned to a FIXED DEC(15) variable) because the result of this built-in has the attributes FIXED BIN(15). If you had intended that this built-in function produce a FIXED DEC(15) result (because, for example, you knew the result of the multiplication might be greater than 2G), this code would not perform the way you had intended and might result in the loss of significant data.

Note that if you want to force the result of this MULTIPLY to be FIXED DEC, you could apply the DECIMAL built-in to the FIXED BIN argument (as in MULTIPLY( DEC(X), 1000, 15 ). You can use the PRECTYPE compiler option to change how the compiler interprets the precision, but that would, of course, potentially change the interpretation of other statements.

## IBM2611, IBM2612: duplicate whens

When compiling some of your "working" code, you might also see a message such as one of the following:

```
IBM2611I W   The binary value ... appears in more than one WHEN clause.
IBM2612I W   The character string ... appears in more than one WHEN clause.
```

This message is easier to understand than some of the others discussed in this section and would be produced by code such as the following:

```
SELECT( OPT );
  WHEN( 'f','F' )
    BUFROM = ETOS(OPTARG);
  WHEN( 'T','T' )
    BUTO = ETOS(OPTARG);
```

```
  WHEN( 'n','N' )
    MAXRECIN = ETOL(OPTARG);
  WHEN( 'k','K' )
    KFLG = ^KFLG;
  WHEN( 'm','M' )
    MAXERR = ETOL(OPTARG);
  OTHERWISE;
    /* ungueltige Option */
END;
```

The message is indicating that the second WHEN clause above is probably meant to be coded as `WHEN( 't', 'T' )`.

The old compilers would have issued no message, and perhaps the code as written is not incorrect; however, it would probably be worthwhile to examine closely any code producing this message.

## IBM2617: passing labels out of PL/I

This message alerts you to a bad coding practice that might require you to edit some of your source code.

In general, the use of GOTO statements is a very poor programming practice, but if you pass a LABEL constant or variable to an ENTRY declared with OPTIONS(ASM), OPTIONS(COBOL) or OPTIONS(FORTRAN), you must not attempt to do a GOTO from that non-PL/I code back into your PL/I code by using the passed label. If you have code that is doing this, you must change it.

## IBM2621: missing ON ERROR SYSTEM

When compiling some of your "working" code, you might also now see this message:

```
IBM2621I W   ON ERROR block does not start with ON ERROR SYSTEM.
             An error inside the block may lead to an infinite loop.
```

The new compiler will produce this message for any ON ERROR block that does not start with the statement ON ERROR SYSTEM. If an ON ERROR block does not start with this statement, and if there is an error in the ON ERROR block, the block will most likely be reentered and an "infinite" loop would result.

Code that produces this message should be corrected.

## IBM2622: warning on poorly coded DO loops

When compiling some of your "working" code, you might also now see this more obscure message:

```
IBM2622I W   ENTRY used to set the initial value in a DO loop will
             be invoked after any TO or BY values are set.
```

The new compiler will produce this message for code such as the following:

```
dcl jx    fixed bin;
dcl last  fixed bin init(10);

do jx = f() to last;
  put skip list( jx );
end;
```

```
f: proc returns( fixed bin );
  last = 4;
  return( 2 );
end;
```

Note that in this code the function `f` that sets the initial value in the loop also changes the value of the variable *last* that sets the final value in the loop. The message is alerting you to the fact that this change to the variable *last* will be made after the compiler has already used that variable to set the final value for the loop. In the concrete terms of this example, the loop will run from 2 to 10, not 2 to 4.

This is different from what the old compiler would have done for such code: under the compiler, this loop would have run from 2 to 4.

So to make this code behave the same as it did under the old compiler, it would be necessary for you to change your source code. This would be a good idea in any case because it is not a good programming practice to have functions that have side effects such as changing other variables in their calling routine. The code as written above is also not very transparent, and unclear code with obscure effects is never good.

## IBM2626: SUBSTR with a zero length

If you have some especially poor code, you might also now see this message:

```
IBM2626I W   Use of SUBSTR with a third argument equal to 0 is
             somewhat pointless since the result will always be a
             null string.
```

If the compiler flags any of your code with this message, it has almost certainly found an error in your code that you should promptly fix.

## IBM2628: large BYVLAUE parameters

Because the old compiler had only very limited support for the BYVALUE attribute, you are not likely to see this message when compiling old code:

```
IBM2628I W   BYVALUE parameters should ideally be no larger than 32 bytes.
```

However as you start to use the BYVALUE attribute more, you might see this message, and in that case you should heed it. You should reserve the use of the BYVALUE attribute for small scalars and ideally for variables that could be passed in a register. Typically, these would be declared as:

- REAL FIXED BIN
- REAL FLOAT
- POINTER
- OFFSET
- HANDLE
- ORDINAL
- CHAR(1)
- ALIGNED BIT(1)
- ALIGNED BIT(8)

You should never use the BYVALUE attribute with strings or aggregates that are larger than 4096 bytes in size.

## IBM2801: introduction of scaled FIXED BIN

This message alerts you to a possible misunderstanding of PL/I rules and, consequently, a possible source of problems:

```
IBM2801I I  FIXED DEC(p1,q1) operand will be converted to FIXED BIN(p2,q2).
            This introduces a non-zero scale factor into an integer operation
            and will produce a result with the attributes FIXED BIN(r,s).
```

This message applies to arithmetic operations where one operand is scaled FIXED DEC and one is unscaled FIXED BIN. By PL/I rules, under the RULES(IBM) compiler option, if one operand in an arithmetic operation is DECIMAL and one is BINARY, the result is BINARY. This applies even if the DECIMAL operand is FIXED DEC with a nonzero scale factor and the BINARY operand is FIXED BIN with a scale factor of zero.

For example, if X is FIXED DEC(5,1) and Y is FIXED BIN(15), then in evaluating the expression X+Y, X will be converted to FIXED BIN(18,4), and the result will have the attributes FIXED BIN(20,4). The compiler will also issue the W-level message IBM1099I because a FIXED DEC(5,1) value whose fractional is not .0 or .5 cannot be exactly represented in FIXED BIN.

If you want to eliminate this message and avoid the problems it hints at, you might apply the DECIMAL built-in function to the FIXED BIN operand. For example, X+DEC(Y) would produce a result with the attributes FIXED DEC(8,1).

## IBM2804: suboptimal compares

This I-level message alerts you to a poor programming practice and possible error:

```
IBM2804I I Boolean is compared with something other than '1'b or '0'b.
```

A Boolean is a result of a comparison of two expressions or the result of anding, oring , or negating Booleans. As such, a Boolean can have only the values ’1’b or ’0’b. If your code compares a Boolean with something other than one of these values, it might reflect a problem (for instance, maybe the expression (a > b) = c was meant to be (a + b) = c ).

Note that the compiler will produce this message even if you compare a boolean to a value declared as BIT(1) STATIC INIT(’1’b). In this situation, there is no programming error, but the compiler cannot generate code as good as the code it would generate if the value were declared as BIT(1) VALUE(’1’b).

## IBM2810: conversion of scaled FIXED BIN

When compiling some of your code, you might see the following message:

```
IBM2810I I Conversion of FIXED BIN(31,16) to FIXED DEC(15,12) may
               produce a more accurate result than under the old
               compiler.
```

For an example of the code that will produce this message, and for an explanation of what to do, see “Conversions from scaled FIXED BINARY” on page 119.

## IBM2811: use of PICTURE as DO control variables

This message alerts you to a poor coding practice that might require you to edit some of your source code: even under OPT(0), the new compiler will flag any DO loop where the control variable has the PICTURE attribute. The compiler will issue this informational message:

```
IBM2811I I Use of PICTURE as DO control variable is not recommended.
```

In general, the use of PICTURE variables as DO loop control variables is a very poor programming practice (especially because it can lead to poor performance), and it would be best to change such code to use FIXED BIN variables as the loop control variables.

## IBM2812: poor TRANSLATE and VERIFY

This message alerts you to a coding practice that was ok under the old compiler, but for which there is a much better alternative with the new compiler: rather than declaring named constants with the attributes STATIC INIT, you can now with the attribute VALUE.

This change will particularly help code such as:

```
test: proc( c );

  dcl c char(20);

  dcl upper char(26) static init('ABCDEFGHIJKLMNOPQRSTUVWXYZ');
  dcl lower char(26) static init('abcdefghijklmnopqrstuvwxyz');

  c = translate( c, upper, lower );
end;
```

Because the named constants *upper* and *lower* are declared as STATIC INIT, both the old and new compilers will build the translate table at run time. This is expensive. However, the new compiler will also issue these informational messages:

```
IBM2812I I Argument number 2 to TRANSLATE built-in would lead to
               much better code if declared with the VALUE attribute.
    IBM2812I I Argument number 3 to TRANSLATE built-in would lead to
               much better code if declared with the VALUE attribute.
```

If you change the STATIC INIT in both declares to VALUE, these messages will be eliminated and the compiler will generate much better code.

## PLIXOPT messages

The PLIXOPT variable is a varying-length character string that contains runtime options that you can specify at compile time. The messages that the compiler produces to diagnose errors in these options are different from the messages produced by the old compilers. In most cases, the PL/I messages now list an associated Language Environment message that you should read for more information.

A module containing a PLIXOPT declare will also now contain a compiler-generated CEEUOPT CSECT that contains the Language Environment encoding of the runtime options specified in the PLIXOPT string. For small modules, this CSECT can cause a substantial increase in the object size of the modules.

## Using the compiler user exit

If you want any messages to have a higher severity, you can use the EXIT compiler option. This option makes it very easy for you to raise the severity of any informational, warning, or error message.

For more information about how to use this option, see the *Enterprise PL/I for z/OS Programming Guide*.

# Chapter 13. Understanding when working code must be changed

This chapter documents situations where the new compiler generates different code than the old compilers. The problems discussed in this chapter have been important to customers who have already migrated to Enterprise PL/I, and you should read this chapter closely to see if they might affect you.

Some of the problematic code discussed in the sections that follow is flagged by the compiler, and you should examine (and change as appropriate) the code associated with the messages issued. In particular, you should examine any compilations producing the following messages:

- IBM1063
- IBM1089

Note also that using the options DECIMAL(NOFOFLONASGN) DFT(OVERLAP) and STATIC(FULL) might eliminate some of these problems.

## Incorrect code

Your code must be correct code that conforms to the rules of PL/I. The Enterprise PL/I compiler might produce different results (including abends) from what the old compiler produces for code that is incorrect. You might get "lucky" in that some incorrect code does what you intended, but you must not rely on this. You must change your incorrect code.

These rules might seem obvious: for example, no user would expect to write to an element of an array using an index that is outside of the bounds of that array. However, in some cases, the fact that code is incorrect and needs to be changed might be less obvious. This section will attempt to describe some instances of incorrect code that must be changed; however, it is not a list of all incorrect code because the opportunities for writing bad code are endless.

### Relying on the order of declarations

If you declare one variable after another, you must not presume that they are contiguous in storage or even that the second variable is in storage after the first.

For example, in the following code, the storage allocated to the variable *a* might not immediately follow the storage allocated to the variable *b*, and hence the assignment could overlay 100 bytes of storage allocated to some other variable.

```
dcl a char(100);
     dcl b char(100);
     dcl c char(200) based;
     addr(a)->c = '';
```

In fact, if the variable *b* is unused, the compiler will most likely allocate no storage to it.

### Using invalid FIXED DECIMAL data

All FIXED DECIMAL variables that you use must be used only when they contain valid data.

© Copyright IBM Corp. 1999, 2017

If a FIXED DECIMAL variable contains invalid data (such as bad numeric digits or a bad sign nibble), any use of that variable might lead to a data exception. Even the assignment of such a variable to another variable with a similar precision and scale might lead to a data exception - even though the assignment could be done through a byte move.

Conversely, you should not presume that a data exception will be raised on the first use of such a variable: for example, the assignment described above might be done with a byte move under some circumstances, and in that case, a data exception would not occur until it was used in an arithmetic operation or a compare and so on.

## Using invalid SUBSTR references

Any SUBSTR reference that you use must be such that its use will not raise the STRINGRANGE condition if that condition is enabled.

If the STRINGRANGE condition is not enabled (and by default, it is not), a SUBSTR reference that is invalid can cause the compiled code to overwrite storage allocated for other purposes, and that, in turn, can lead to data corruptions or abends.

For example, in the following code, if the value in the variable *n* is larger than 100, the SUBSTR reference is invalid, and the generated code might overwrite storage allocated to other variables.

```
dcl f ext entry;
     dcl a char(100);
     call f( 'test' || substr(a,1,n) );
```

You can easily detect such bad code during test by compiling your programs with the PREFIX(STRINGRANGE) compiler option.

The SUBSTR suboption introduced in V3R8 to the USAGE compiler option can allow some of this incorrect code to be accepted. However, it would be best not to use this option; instead, you should correct your code. For the above example, you correct the code by changing the declare of *a* to have a length at least as large as the largest value that *n* could assume; if the maximum value for *n* is unknown, you can change the declare of *a* to have a length of 32767.

In some situations, the old compiler also generated code for SUBSTR references of the form `SUBSTR(X,1,N)`, where *X* was CHAR and *N* was greater than 32767. However, such references are invalid and would have raised STRINGSIZE if it were enabled. The new compiler enforces the restriction that the length of a SUBSTR reference must be less than 32768 for CHAR and BIT references and less than 16384 for GRAPHIC and WIDECHAR references, and you must correct any code that does not conform to these rules.

## Using dissimilar EXTERNAL declares

If you declare an EXTERNAL variable in more than one compilation unit, those declares must match. In particular, all the attributes in the two declares must match.

For example, if you declare an EXTERNAL FILE in one compilation unit with the attributes KEYED ENV( VSAM ), you must declare it with the same attributes in any other program linked with the first.

## Using an incorrect PLITABS declare

If your code contains a declare for PLITABS, the pagesize, linesize, and other values must be valid; in addition, the first field in the PLITABS structure must also be valid.

This field is supposed to hold the offset to the field specifying the number of tabs set by the structure, and the Enterprise PL/I library code will not work correctly if this is not true.

# Initializing variables

You must not use a variable before it has been initialized. Any program that uses an uninitialized variable is invalid and must be corrected.

The best way to correct such code is to add the INITIAL attribute to those variables that need it. However, there are also some other ways to initialize your variables, and this section discusses these alternative ways.

## Initializing AUTOMATIC

This topic describes how the compiler initializes AUTOMATIC when various options are in effect.

The compiler option INITAUTO will add an appropriate INITIAL attribute to any AUTOMATIC variable that does not have an INITIAL attribute if the variable has one of the following attributes:

- FIXED or FLOAT
- PICTURE, CHAR, BIT, GRAPHIC or WIDECHAR
- POINTER or OFFSET

See the *Programming Guide* for more details.

The compiler option DFT(INITFILL) will fill all AUTOMATIC storage with a specified byte value (or to '00'x if no byte value is specified). This can be used to initialize variables with these attributes:

- FIXED BIN
- FLOAT
- VARYING or VARYINGZ
- POINTER or OFFSET

The compiler option INITFILL will also fill all other AUTOMATIC variables with the specified (or default) byte value, but these variables would not really be properly initialized. For example, use of a FIXED DEC variable initialized through DFT(INITFILL) will lead immediately to a data exception.

Setting the third suboption of the runtime option to 00 (as in STORAGE(,,00)) will fill all AUTOMATIC storage in all the routines (including library routines) with the hex value 00. This has the same effects and validity as the DFT(INITFILL) compiler option except that it applies to all routines in the application and has a dreadfully bad impact on performance. Furthermore, because the compiler does not know if this option is being used, it might not have the desired effect for code compiled with OPT(2) or OPT(3): the fact that a variable is uninitialized makes the code invalid and might lead the optimizer to make choices about how to optimize the code that cannot be repaired by using this runtime option.

Setting the third suboption of the runtime option to CLEAR (as in STORAGE(,,CLEAR)) will fill all AUTOMATIC storage with the hex value 00 before MAIN is invoked. This has the same effects and validity as the DFT(INITFILL) compiler option except that it applies only to the MAIN routine. Furthermore, because the compiler does not know if this option is being used, it might not have the desired effect for code compiled with OPT(2) or OPT(3): the fact that a variable is uninitialized makes the code invalid and might lead the optimizer to make choices about how to optimize the code that cannot be repaired by using this runtime option.

### Initializing BASED

The compiler option INITBASED does for BASED what INITAUTO does for AUTOMATIC.

### Initializing CONTROLLED

The compiler option INITCTL does for CONTROLLED what INITAUTO does for AUTOMATIC.

### Initializing STATIC

The compiler option INITSTATIC does for STATIC what INITAUTO does for AUTOMATIC.

Without this option, all uninitialized STATIC storage will be filled with binary zeros. Of course, as was true with the compiler DFT(INITFILL) and runtime STORAGE options discussed above, this means that many variables such as FIXED DEC variables would have invalid values.

## Retaining unused declarations

This section describes how the new compiler handles storage allocation for unused declarations.

### Retaining unused INTERNAL STATIC

If an INTERNAL static variable is unused, the compiler will not allocate any storage for it.

For example, if the following declaration is the only reference to the variable *build_data*, no storage would be allocated for this variable and its initial value would not be in the generated text.

```
dcl build_data char(30) var static
               init('Compiled in build 17');
```

If the ABNORMAL attribute is specified on a level-1 static variable, the compiler will allocate storage for the variable. For example, to keep the variable above, you could change the declaration above as follows:

```
dcl build_data char(30) var static abnormal
               init('Compiled in build 17');
```

Do not apply the ABNORMAL attribute indiscriminately to all variables or all static variables - this will both slow down your compilation and worsen the performance of the generated code.

If you specify the compiler option STATIC(FULL), the compiler will apply the abnormal attribute to all static. This is a coarse solution and is not recommended.

## Incorrect code that will now raise exceptions

### FIXEDOVERFLOW when SIZE is disabled

Under both the old and new compilers, if you try to assign a source to a numeric target and the source is too big, the SIZE condition will be raised if it is enabled. However, if the SIZE condition is NOT enabled, your program is in error and what happens is unpredictable. You should correct such a program.

Under the old compiler, sometimes no condition will be raised. For example, consider the following program:

```
dcl A fixed dec(3);
dcl B pic'9';

A = 123;
B = A;
```

The value in the source *A* is too large to fit into *B*, and if SIZE is enabled, it will be raised. However, when SIZE is disabled, the old compiler raises no condition. That does ***not*** mean your program is correct - in fact, it is incorrect and should be changed. For instance, if you wished to set *B* to just the ones digit of *A*, you could change the above code as follows:

```
dcl A fixed dec(3);
dcl B pic'9';

A = 123;
B = mod(A,10);
```

Moreover, under the old compiler, sometimes a condition will be raised for very similar code. For example, consider the following program:

```
dcl X fixed dec(5);
dcl Y fixed dec(4);
dcl Z fixed dec(5);

X = 99999;
Y = X + 1;
Z = X + 1;
```

The value of the expression *X + 1* is too large to fit into either *Y* or *Z*, and if SIZE is enabled, it will be raised for both statements. However, when SIZE is disabled, the old compiler raises no condition for the assignment to *Y* and raises FIXEDOVERFLOW for the assignment to *Z*. Again your program is incorrect and should be changed.

The new compiler handles these statements consistently, but the results depend on the target attributes and the compiler options in effect:

- When the SIZE condition is disabled, if the target has the PICTURE attribute, the generated code will not raise the FIXEDOVERFLOW condition.

  More precisely, the generated code will not raise the FIXEDOVERFLOW condition when SIZE is disabled when assigning a source expression with any of the following data types to a non-floating point PICTURE target:

  – FIXED BIN
  – FIXED DEC
  – Non-floating point PICTURE
- When the SIZE condition is disabled, if the target has the FIXED DEC attribute, then the following applies:

- If the default compiler option DECIMAL(FOFLONASGN) is in effect, the generated code will raise the FIXEDOVERFLOW condition.
- If the compiler option DECIMAL(NOFOFLONASGN) is in effect, the generated code will not raise the FIXEDOVERFLOW condition.

Note that the above discussion applies to assignments only: if an operation such as addition or multiplication produces a result requiring more than 15 digits (or more than 31 if the LIMITS(FIXEDDEC(15,31)) option is in effect), an exception will be raised. The exception raised will usually be FIXEDOVERFLOW, but depending on the machine instructions generated, other exceptions, such as a specification exception, might be raised.

Similarly, when a BIT variable is assigned to a FIXED BIN variable, if the BIT variable was too large to be validly converted and if SIZE was not enabled (and hence the program was invalid), the following might happen:

- The old compiler would sometimes raise no condition and simply assign 0 to the target.
- The new compiler will raise the SIZE condition if the conversion is done via a library call.

For example, consider the following program:

```
dcl A bit(32) aligned;
dcl B fixed bin(31);

A = '80000000'bx;
B = A;
```

The value in the source *A* is too large to fit into *B*, and if SIZE is enabled, it will be raised. The new compiler will raise the SIZE condition for this code even when it is disabled. However, when SIZE is disabled, the old compiler raises no condition. That does not mean your program was correct - in fact, it was always incorrect and should be changed.

## Invalid allocations

Under the old compilers, the following piece of code "worked":

```
dcl vdptr  pointer;
dcl vdcom  char(2000) based(vdptr);

dcl
  1 vdcommarea based(addr(vdcom)),
    2 vda  char(1000),
    2 vdb  char(1000),
    2 vdz  char(1);

alloc vdcom;

vdcommarea = '';
```

This code is not valid PL/I code because you must not use a 2001 byte area to overlay a 2000 byte allocated piece of storage. By luck, this "worked" under the OS PL/I V2R3 run time, but under the Language Environment run time, this code fails miserably.

## Incorrect code that will now loop endlessly

This section describes incorrect code that will cause endless loops. The new compiler alerts you to such situations with messages. You should closely examine and probably change any code that causes the compiler to issue these messages.

### Even precision PICTURE loop control variables

Consider the following program that initializes an array:

```
winter: proc;
   dcl n pic'99';
   dcl a(0:99) fixed bin ext;
   do n = 0 to 99;
      a(n) = n;
   end;
end;
```

This code is not valid PL/I code. If the SIZE condition is enabled, it will be raised after *n* becomes equal to 99 (because the next value, 100, it would assume is too large for a PIC'99' variable).

For best performance, using a PICTURE variable for a loop control variable is usually not a good idea. However, for the code above it is a very bad idea because the new compiler will generate code that will make this loop run endlessly.

By the definition of the DO statement, this loop is equivalent to the following code, which will loop infinitely under both the old and new compiler.

```
n = 0;
if n > 99 then go to loop_exit;
loop_body:;
   a(n) = n;
n = n + 1;
if n <= 99 then go to loop_body;
loop_exit:;
```

However, for the original code using the DO-loop, the old compiler cheats and generates code that is, strictly speaking, incorrect.

The new compiler will try to alert you to this situation by issuing the following messages:

```
IBM1089I W    Control variable in DO loop cannot
              exceed TO value, and loop may be infinite.
IBM1220I W    Result of comparison is always constant.
IBM1220I W    Result of comparison is always constant.
```

You should closely examine (and probably change) any code that causes the compiler to issue message IBM1089. You could also use the EXIT option to raise the severity of this message.

To correct your code, you could change the attributes for the DO-loop control variable from PICTURE to FIXED BIN(31).

Finally, note that this problem will occur in any loop where the DO-loop control variable is PICTURE'(*n*)9' when *n* is an even number and the loop limit is equal to 10***n*-1.

This problem could also occur in forms that will not be flagged by the compiler. For example, consider the following program that initializes an array:

```
sommer: proc;
   dcl n pic'999';
   dcl a(0:999) fixed bin ext;
   do n = 0 to 998 by 2;
      a(n) = n;
   end;
end;
```

In this case, because the TO value of 998 is less than the maximum value that *n* could assume, the compiler will not issue message IBM1089. However, after *n* assumes the value 998, the next time through the loop *n* will be assigned the value 0 and the loop will repeat.

This problem could also occur when the BY value was negative:

```
eiki: proc;
   dcl n pic'999';
   dcl a(0:99) fixed bin ext;
   do n = 79 to 1 by -2;
      a(n) = n;
   end;
end;
```

However, after *n* assumes the value 1, the next time through the loop *n* will be decremented by 2 and assigned the value 1 and the loop will repeat.

## Assignments that will produce different results

This section describes assignments for which you might see different results from the old and new compilers. You might need to change your code or use specific compiler options to make your code compatible.

### Source-target overlap

Consider the assignment `P->Z = Q->Z;` where Z is CHAR(6) BASED.

Under OPT(0), the old compiler would assign the source first to a 6-byte temporary and then assign the temporary to the target.

However, under OPT(2), the old compiler would perform the assignment with one MVC.

These different implementations lead to different results if the source and target overlap.

The new compiler controls this behavior through the OVERLAP suboption of the DEFAULT compiler option:

- Under DFT(NOOVERLAP), the compiler will assume the source and target never overlap.
- Under DFT(OVERLAP), the compiler will generate more conservative code whenever necessary.

For example, for the assignment `SUBSTR(A,4,6) = SUBSTR(A,3,6);`, if `A = 'abcdefghijklm'`, the compiler sets the value of A as follows:

- The old compiler sets A = 'abccdefghjklm'.
- The new compiler, under DFT(OVERLAP), sets A = 'abccdefghjklm'.
- The new compiler, under DFT(NOOVERLAP), sets A = 'abccccccjklm'.

Consequently, for the most compatibility with the least work, you might want to specify the compiler option DFT(OVERLAP).

But specifying this option will also force the compiler to generate slower code in situations where you know the source and target do not overlap, and it will also cause the compiler to forego some other optimizations. You would be much better off if you changed your code to avoid source and target overlap and then use DFT(NOOVERLAP).

For instance, consider the following assignment:

```
SUBSTR(A,4,6) = SUBSTR(A,3,6);
```

This assignment can be replaced by the following assignments:

```
temp_Char6 = SUBSTR(A,3,6);
SUBSTR(A,4,6) = temp_Char6;
```

## Float-to-float assignments

The new compiler converts a FLOAT DECIMAL literal, such as 3.1415926E0 or 1E-02, to its internal floating-point representation solely by examining the literal's attributes and not by examining the context in which it used.

For example, 3.1415296E0 has the attributes FLOAT DEC(8), and hence the new compiler will convert it to long floating point. But, 1E-02 has the attributes FLOAT DEC(1), and hence the new compiler will convert it to short floating point.

If the literal is used in an assignment or an initial clause, the compiler will then convert, if necessary, its floating-point value to the attributes of the target of the assignment or initialization.

However, the old compilers would examine the context in which such a literal is used and convert the literal directly to the attributes of its target. The behavior of the old compilers does not strictly follow the rules for PL/I expression evaluation and can lead to different results than those produced by the new compiler. Consider this code fragment:

```
dcl z float dec(06) init(0);
dcl s float dec(06);
dcl q float dec(17);

s = 1e-2;
q = s;
put skip data( q );
q = 1e-2;
put skip data( q );
q = 1e-2 + z;
put skip data( q );
```

In all three assignments to *q* above, the attributes of the source are those of a short floating-point number, and the value of the source should be the same. However, the results of the three PUT statements under the old compilers are as follows:

```
Q= 9.999997913837432861E-03;
Q= 9.99999999999999999E-03;
Q= 9.99999999999999999E-03;
```

The results of the three PUT statements under the new compiler are as follows:

```
Q= 9.99999791383743286OE-03;
Q= 9.99999791383743286OE-03;
Q= 9.99999791383743286OE-03;
```

This kind of difference occurs only for float literals that cannot be exactly represented (such as fractions like 1E-2 that cannot be equated to a binary fraction).

To alert you to situations such as the above, the compiler will issue message IBM1065I when it detects short-floating point literals that cannot be exactly represented.

To get the same results as under the old compilers, you would have to change your source in one of the following ways:

1. Specify the constant by using the FLOAT built-in function applied to a FIXED DECIMAL literal and with the precision you want.

   For example, you would specify 1E-2 as FLOAT(.01,7) to make it a long floating-point value and as FLOAT(.01,17) to make it an extended floating-point value.
2. Add enough zeroes to the literal to give it the precision you want.

   For example, you would specify 1E-2 as 1.000000E-2 to make it a long floating-point value and as 1.0000000000000000E-2 to make it an extended floating-point value.
3. Use the new D or Q format to indicate the desired precision.

   For example, you would specify 1E-2 as 1D-2 to make it a long floating-point value and as 1Q-2 to make it an extended floating-point value.

Note that the first two changes in the above list would be accepted by the old and new compilers (and would produce the same results under each), but the third change would work only under the new compiler.

## Other statements that will produce different results

This section provides information about some other statements for which you might see different results from the old and new compilers.

### STREAM I/O with unprintable characters

If a character with the value '00'x, '0C'x through '0F'x, or '15'x is part of the output of a PUT FILE statement, a period ('4B'x) will be output instead under some scenarios.

The scenarios are listed below:

- The code is running under batch and is compiled with the STDSYS option, the file is SYSPRINT, and SYSPRINT is directed to SYSOUT.
- The code is running under z/OS UNIX, and the file is a STREAM OUTPUT file being written to the command window.
- The code is running under TSO, and the file is a STREAM OUTPUT file being written to the TSO terminal.

### Uninitialized EXTERNAL STATIC

The old and new compilers are different with respect to storage allocation for uninitialized EXTERNAL STATIC variables.

Under the old compiler, a variable declared as EXTERNAL STATIC but with no INITIAL value(s) specified for it was not allocated any storage (and a linkage editor ESD of type CM was issued). The storage for it must have been defined in some other program object that will be linked with it. In fact, the storage that was actually allocated might be bigger than what its declare specified (or implied). For example, consider the following code declares:

```
dcl testpcl ext static, pcl char(16) based(addr(testpcl));
```

The variable *testpcl* has the (implied) attributes of FLOAT DEC(6) and hence would seem to be allocated only 4 bytes of storage. If all the programs linked with this one also declare *testpcl* with the same PL/I declare, then exactly 4 bytes will be allocated to it. However, if it is linked with, say, an assembler that defined *testpcl* as a 16-byte CSECT, the linker will allocate 16 bytes to it.

The new compiler will currently allocate 4 bytes to such a variable (and issue a linkage editor ESD of type SD and length 4). An attempt to use it as the base for, say, a 16-byte area will lead to errors.

If you want to declare a variable but have its storage allocation be determined by its declaration in another module, you should declare it with the RESERVED option. For example, the declare above should be changed as follows:

```
dcl testpcl ext static reserved, pcl char(16) based(addr(testpcl));
```

If, however, you compile with the option COMMON, the Enterprise PL/I compiler will also issue a linkage editor ESD of type CM, and the code would work as it did with the old compiler.

## Incompletely declared FILEs

You must be aware of the difference in how the old and new compilers handle incompletely declared FILEs.

Under the old compiler, if you declared an EXTERNAL FILE in one routine with some attributes, such as RECORD, but in another routine linked with the first routine, you did not declare the file or declared it with no attributes (other than FILE), the second routine would use the attributes declared in the first routine even if the second routine opened the file first.

Enterprise PL/I would handle this differently: the second routine would not "see" the attributes from the first routine and would instead apply the default attributes, such as STREAM, to the file. This can lead to problems.

You should correct this code by declaring the FILE identically in all routines; in fact, all EXTERNAL variables should be declared identically in all routines.

## Dummy arguments and alignment

According to the *PL/I Language Reference*, a dummy argument will be created if an argument differs from its parameter description in its alignment. However, the old compiler followed this rule for CHARACTER NONVARYING, but not for CHARACTER VARYING. The new compiler applies the rule consistently.

For example, consider the following code:

```
dcl x entry( unaligned char(8) );
dcl y entry( unaligned char(8) varying );
dcl a aligned char(8);
dcl b aligned char(8) varying;
call x( a );
call y( b );
```

Dummy arguments should be created for both CALL statements, but only the Enterprise PL/I compiler will create a dummy argument for the second CALL.

Note that you can use the DEFAULT(DUMMY(UNALIGNED)) compiler option to make the compiler ignore alignment mismatches when deciding when to create dummy arguments. If this option were in effect, the compiler would not create a dummy argument for either of the CALLs in the above example.

### Dummy arguments and CONTROLLED

According to the PL/I Language Reference, a dummy argument will be created if an argument is a CONTROLLED string or area (because an ALLOCATE statement could have changed the length or extent, hence causing the string length or area size to be different from what is required by the called routine). The new compiler applies this rule consistently, but the old compiler does not.

Under Enterprise PL/I, this rule applies unless the RULES(NOLAXCTL) option is in effect and the string length or area size is a constant. However, the old compiler was not always consistent about following this rule (which should always have applied because the old compiler had no equivalent to the RULES(NOLAXCTL) option).

For example, consider the following code:

```
dcl x entry( char(8) );
dcl a controlled char(8);
dcl 1 b(2) controlled, 2 c char(8);
call x( a );
call y( b(1).c );
```

Dummy arguments should be created for both CALL statements, but only the Enterprise PL/I compiler will create a dummy argument for the second CALL.

### Pointer arithmetic

In expressions involving pointer arithmetic, it is presumed that the pointers are addresses. Consequently, when adding a value to a pointer, the result pointer might not have the high order bit on, even though the source pointer did have the high order bit on.

## Code that will not perform as well

This section describes code that you might consider changing for better performance.

### FIXED DEC as a loop control

A DO-loop that has a FIXED DECIMAL or PICTURE control variable will perform much worse than a loop that has a FIXED BINARY control variable.

You can get significantly better code if you change the declarations for your loop control variables from FIXED DEC to FIXED BIN(31).

## FIXED BIN(15) as a loop control

A DO-loop that has a FIXED BIN(15) control variable will perform worse than a loop that has a FIXED BIN(31) control variable.

Under OPT(2) or OPT(3), the compiler will issue I-level message IBM1063 to flag code that uses FIXED BIN(15) control variables. You can get better code if you change the declarations for your loop control variables from FIXED BIN(15) to FIXED BIN(31).

## I/O using TOTAL

Because the TOTAL option of the ENVIRONMENT attribute is not supported, I/O to files using it will generally not perform as well.

# Chapter 14. Understanding when working code may need to be changed

This chapter documents more situations where the new compiler generates different code than the old compilers. But unlike the previous chapter, these differences are somewhat obscure; they are included in this document for completeness and because they could potentially affect you.

## Code that will now raise an exception

This section provides information about code that will raise an exception under the new compiler.

### ZERODIVIDE and OVERFLOW promoted to ERROR

The new compiler improves the way it handles ZERODIVIDE or OVERFLOW.

Under the old compiler if ZERODIVIDE or OVERFLOW were raised and there was an ON-unit for the condition, and if the END statement for the ON-unit was reached, your program would continue with the next machine instruction after the one that raised the condition.

If the condition was raised by a hardware exception, this meant that your program continued on with some unknown value as the result of the operation, and this often led to more errors.

Under the new compiler, if either ZERODIVIDE or OVERFLOW is left unhandled by an ON-unit, the condition will be promoted to ERROR.

### Conditions raised when disabled

Under the old compiler, if a condition such as CONVERSION or SUBSCRIPTRANGE was disabled, the condition would almost never be raised. Under the new compiler, disabling a condition asserts that the condition will not occur. However, the condition might still be raised.

For some code sequences, this allows the compiler to generate faster code. For example, for an assignment of a CHAR(1) to a FIXED BIN, if CONVERSION is enabled, the conversion will be done by a library call. But if CONVERSION is disabled, the conversion will be done by very simple inline code that "ands out" the left nibble in the CHAR(1) value. This code is possible only because NOCONVERSION asserts that a conversion condition could not occur in this statement. If this assertion is not true, your program is invalid.

However, for an assignment of a CHAR(2) to a FIXED BIN, the conversion will always be done by a library call (because there are too many possibilities for what might be held in those two characters), and even if NOCONVERSION is in effect, the CONVERSION condition will be raised if the source does not contain a valid numeric value. (Note also that if you know that the CHAR(2) source contains only numeric digits, you could avoid this library call by using an appropriate picture string in either the EDIT built-in function or in a variable declared as based on or unioned with the source.)

© Copyright IBM Corp. 1999, 2017

Similarly, if SUBSCRIPTRANGE is disabled, you are asserting that all subscripts are valid. For most statements, this means the compiler will not generate any code to check the validity of the subscripts, and if any subscripts are invalid, your program is in error. However, if a subscripted reference is used in a PUT DATA statement, a library routine will evaluate that reference, and if any subscript is invalid, the SUBSCRIPTRANGE condition will be raised - even if disabled.

## Invalid RETURNs

If a piece of code attempts to return a value when no value should be returned, no condition will be raised under the old compiler, but the new compiler alerts you to this error by raising an ERROR condition.

Consider the following somewhat senseless, but illustrative, code fragment:

```
call y;

x: proc returns( pointer );
  y: entry;
  return( sysnull() );
end;
```

This program fragment is in error because when the procedure is entered at Y, no value should be returned, but the code attempts to return a value nonetheless.

Under the old compiler, no condition would be intentionally raised when the invalid return was attempted, and the program might fail in any number of ways (and it might even complete "successfully").

Under the new compiler, the ERROR condition would be intentionally raised by the generated code with ONCODE=9004.

## GOTO holes

If GOTO holes are detected in your code, the old compiler raises a protection exception, while the new compiler raises an ERROR condition.

Consider the following code fragment:

```
dcl x(4) label;

goto x(n);
x(4):;
put skip list( n );
x(3):;
put skip list( n );
x(2):;
put skip list( n );
x(1):;
put skip list( n );
```

Note that if $n < 1$ or if $n > 4$, and if the SUBSCRIPTRANGE condition is not enabled, your program was in error.

Under the old compiler, a protection exception usually resulted.

Under the new compiler, the ERROR condition will be raised with ONCODE=9003 with the following message:

```
IBM0751S  ONCODE=9003  A GOTO was attempted to an element of a label constant
          array, but the subscripts for the element were not those of any
          label in that array.
```

### The scope of NOFOFL

Under Enterprise PL/I, the FIXEDOVERFLOW/NOFIXEDOVERFLOW (or FOFL/NOFOFL) prefix applies only to FIXED DECIMAL operations. However, note that the (NO)FOFL prefix, when applied to a PROCEDURE or BEGIN statement, applies only to that block and to the blocks statically contained within it. The prefix does not apply to any other code dynamically called from within these blocks.

Similarly, if the (NO)FOFL prefix is applied to a CALL statement or a statement containing a function invocation, the setting in the prefix does not apply to the code in the invoked routine: it applies only to any FIXED DECIMAL calculations before or after the routine is invoked.

## Code that will now not raise exceptions

This section provides information about code that will no longer raise exceptions under the new compiler.

### FIXEDOVERFLOW for FIXED BIN

Under the old compiler, the FIXEDOVERFLOW (or FOFL) condition would be raised if any FIXED BIN operation produced a result that required more than 31 binary digits. Under the new compiler, the FOFL condition will not be raised for any FIXED BIN computation (but it will still be raised for FIXED DEC computations when needed).

For example, under the old compiler, if you multiplied a FIXED BIN variable equal to 100_000 by itself, the FOFL condition would be raised.

This change in the new compiler makes the PL/I language match the C and JAVA languages, and it also enables the compiler to generate inline code to perform adds and subtracts on 8-byte integers.

In fact, during runtime initialization, the bit in the PSW that enables integer FOFL will not be set if all your code has been compiled by the C or by the new PL/I compilers. It will be set on if there is any old PL/I code in the main module, and that can have some negative performance consequences for some of your new code.

### CONVERSION when assigning blanks to numeric variables

The old and new compilers react differently to the assignment of blanks to numeric variables.

Under the old compiler, the CONVERSION (or CONV) condition would be raised if a character string consisting of one or more blanks (and nothing else) was assigned to a numeric variable. However if a varying character string of length zero was assigned to a numeric variable, CONVERSION would not have been raised (even though a zero-length character string compares equal to character string consisting only of blanks).

Under the new compiler, the CONVERSION condition will not be raised by the assignment to a numeric variable of any character string that would compare equal to a blank string.

### ERROR when mapping excessively large aggregates

If your code declares an aggregate with adjustable extents, its size will be determined at runtime. If its size would be greater than 2G and the compiler generates a call to a library routine to map the variable, the ERROR condition will be raised.

However, for a simple aggregate with adjustable extents, the compiler will generate inline code to determine the variable's size - unless the SIZE condition is enabled. If such a variable had a size greater than 2G and SIZE was not enabled, no condition would be raised and your program would be invalid. Of course, if your aggregates are reasonable in size, you will get far better performance if SIZE is not enabled.

## Storage mapped differently

This section describes situations where the old and new compilers differ with respect to storage. These differences affect how your program runs.

### One-byte FIXED BIN

If you have any variables declared as FIXED BIN with a precision of 7 or less, they occupy one byte of storage under Enterprise PL/I instead of two as under PL/I for MVS & VM and earlier. If the variable is part of a structure, this usually changes how the structure is mapped, and that could affect how your program runs.

For example, if the structure were read in from a file, fewer bytes would be read in under Enterprise PL/I than under PL/I for MVS & VM or earlier PL/I releases.

To avoid this difference, you could change the precision of the variable to a value between 8 and 15 (inclusive).

To help you locate where you might have problems because of this difference, the compiler will flag any FIXED BIN with precision <= 7 with message IBM1044.

The (NO)BIN1ARG suboption of the DEFAULT compiler option controls how the compiler handles one-byte REAL FIXED BIN arguments passed to an unprototyped function:

- Under BIN1ARG, the compiler will pass a FIXED BIN argument as is to an unprototyped function.
- But under NOBIN1ARG, the compiler will assign any one-byte REAL FIXED BIN argument passed to an unprototyped function to a two-byte FIXED BIN temporary and pass that temporary instead.

Consider the following example:

```
dcl f1 ext entry;
dcl f2 ext entry( fixed bin(15) );

call f1( 1b );
call f2( 1b );
```

If you specified DEFAULT(BIN1ARG), the compiler would pass the address of a one-byte FIXED BIN(1) argument to the routine `f1` and the address of a two-byte FIXED BIN(15) argument to the routine `f2`. However, if you specified DEFAULT(NOBIN1ARG), the compiler would pass the address of a two-byte FIXED BIN(15) argument to both routines.

Note that if the routine f1 was a COBOL routine, passing a one-byte integer argument to it would cause problems because COBOL has no support for one-byte integers. In this case, using DEFAULT(NOBIN1ARG) might be helpful; but it would be better to specify the argument attributes in the entry declare.

So, while BIN1ARG is the default suboption, you might find it useful to specify the NOBIN1ARG suboption for increased compatibility.

## Declarations handled differently

This section provides information about declarations that the old and new compilers handles differently.

### AREA with INITIAL

The new compiler ignores the INITIAL attribute for AREAs, and you should convert any INITIAL clauses for AREAs into assignment statements.

For example, in the following code fragment, the elements of the array are not initialized to a1, a2, a3, and a4:

```
dcl (a1,a2,a3,a4) area;
dcl a(4) area init( a1, a2, a3, a4 );
```

However, you can rewrite the code as follows so that the array is initialized:

```
dcl (a1,a2,a3,a4) area;
dcl a(4) area;

a(1) = a1;
a(2) = a2;
a(3) = a3;
a(4) = a4;
```

The compiler will flag any declare of AREA with INITIAL with message IBM1196.

## Conversions handled differently

This section provides information about conversions where the old and new compilers might produce different results.

### Conversions from float to character

In some conversions from FLOAT (BIN or DEC) to CHARACTER, there might be a difference of one in the last digit between the result produced under the old and new compilers.

This difference does not reflect a difference in the underlying floating-point value or in the calculations that led up to it. It is generally safe to ignore this difference.

### Conversions from scaled FIXED BINARY

It is generally best to avoid scaled FIXED BINARY because its use generally causes the compiler to produce less efficient code. Additionally, in some conversions of scaled FIXED BINARY to FIXED DECIMAL, the new compiler might produce a different (but more accurate) result than the old compiler.

For example, consider the following code:

```
dcl i fixed bin(15) init(290);
dcl s fixed bin(31,16);
dcl d fixed dec(15,12);

d = i / 365;
put skip data( d );
s = i / 365;
d = s;
put skip data( d );
```

The results of the two PUT statements under the old compilers are as follows:

```
D= 0.794509887700;
D= 0.794509887700;
```

The results of the two PUT statements under the new compiler are as follows:

```
D= 0.794509887695;
D= 0.794509887695;
```

Note that while the second assignment above clearly involves a conversion of scaled FIXED BIN to FIXED DEC, the first assignment also involves such a conversion because the attributes of the expression *i / 365* are, by the PL/I rules for expression evaluation, FIXED BIN(31,16).

To understand what is happening here, it will help to look at the contents of the variable *s* after it is assigned the result of the divide. *s* will then hold the hex value 0000CB65. If viewed as a FIXED BIN(31,0) number, this would be the value 52069, but because it has scale factor 16, it represents the value 52069/2**16. That value is mathematically equivalent to 52069*5**16/10**16. So, to convert it from base 2 to base 10, the compiler multiplies the value by 5**16 (or 152587890625). That would produce a FIXED DEC value with scale factor 16; hence, to produce the target result with scale factor 12, the last 4 digits are dropped.

As can be verified on a calculator, 52069 times 5**16 is 7945098876953125, and dropping the last digits yields the result produced by the new compiler.

The reason why the old compiler produced a different result is that its generated code multiplied *s* not by 152587890625, but by 152587890626. This leads to a less accurate result.

You can avoid this problem entirely by ensuring that all divisions that could yield a fractional result are performed in decimal. Using the DECIMAL built-in function is one easy way to do this. For example, if, in the first assignment above, the expression *i / 365* were changed to *dec(i) / 365* , the result of the assignment would be 0.794520547945.

To alert you to situations such as the above, the compiler will issue message IBM2810I when it detects conversions of scaled FIXED BIN to FIXED DEC.

## Built-in functions handled differently

This section provides information about built-in functions for which there is difference in the support by the old and new compilers.

### Arithmetic built-in functions with scale factors and FIXED BIN

Under the RULES(IBM) compile-time option, which is the default, variables can be declared as FIXED BIN with a nonzero scale factor. Infix, prefix, and comparison operations are performed on scaled FIXED BIN using the same semantics as the

old compilers. However, the ADD, DIVIDE, or MULTIPLY built-in functions will not produce FIXED BIN results with nonzero scale factors.

The new compiler evaluates these built-in functions as FIXED DEC rather than as FIXED BIN as the old compilers did if either of the following is true:

- Their arguments are FIXED BIN with nonzero scale factors.
- Their arguments are FIXED BIN with zero scale factors but a nonzero value is specified as their fourth argument.

For example, the new compiler would evaluate the DIVIDE built-in function in the assignment statement below as a FIXED DEC expression:

```
dcl (i,j) fixed bin(15);
dcl x     fixed bin(15,2);

...

x = divide(i,j,15,2);
```

Note that in this example, the result is FIXED DEC(6,1) instead of FIXED DEC(15,2). In the general case, for (p,q), the result is (t,s) where t = 1 + ceil(p/3.32) and s = ceil(q/3.32). To get a result that has the attributes FIXED DEC(p,q), apply the DECIMAL built-in function to all of the FIXED BIN arguments; so in this example, the expression would become DIVIDE( DEC(X), DEC(Y) 15, 2 ). You can also use the PRECTYPE compiler option to change the way the compiler interprets the precision, but that potentially changes the interpretation of other statements.

The compiler will flag this difference with message IBM1053.

## String-handling built-in function for conversion of DBCS character strings

The Enterprise PL/I compiler still supports the CHAR built-in function although it now views CHAR as an abbreviation for CHARACTER. The results of the CHAR built-in function are the same as in the old PL/I for MVS & VM compiler except when the first argument has the GRAPHIC type.

- Under the PL/I for MVS & VM compiler, the result was the byte value of that argument enclosed in shift codes.
- Under the Enterprise PL/I compiler, the result is the string produced by converting the GRAPHIC string to CHARACTER. If the conversion is not possible, the CONVERSION condition will be raised.

For example, if X is GRAPHIC(3) and holds the byte .A.B.C, the results are as follows:

- Under the PL/I for MVS & VM compiler, GRAPHIC(X) yields <.A.B.C>.
- Under the Enterprise PL/I compiler, GRAPHIC(X) yields ABC.

### Examples

The following examples show how to change your code to obtain the results produced by the old compiler.

**Example 1**

```
add
    dcl so char(1) value ('0e'x), si char(1) value('0f'x);
then replace
```

```
    A = CHAR(X);
by
    UNSPEC(A) = UNSPEC(SO) || UNSPEC(X) || UNSPEC(SI);
```

**Example 2**

```
replace
    CHAR(X)
by
    OLDCHAR(X)
where OLDCHAR is defined by
    oldchar: proc(x) returns( char(32767) var );
      dcl x graphic(*);
      dcl a char(32767) var;
      dcl d char(2*length(x));
      a = '0e'x;
      unspec(d) = unspec(x);
      a = a || d;
      a = a || '0f'x;
      return( a );
    end;
```

For more information about the CHARACTER and CHARGRAPHIC built-in functions, refer to the *Enterprise PL/I for z/OS Programming Guide*.

# MACRO preprocessor differences

This topic describes situations where you might need to change your code because of differences between the old and new MACRO preprocessors.

## MACRO preprocessor and strings

Under the old compiler, the MACRO preprocessor would uppercase all text except for text enclosed in strings and comments. But the old compiler recognized only text delimited by '...' as strings: text delimited by "..." was not recognized as a string and was uppercased.

The new compiler will, under the default preprocessor option of CASE(UPPER) also uppercase all text except for text enclosed in strings and comments. However, the new compiler recognizes both text delimited by '...' and text delimited by "..." as strings and will not uppercase either.

This difference could cause a problem if you were running the MACRO preprocessor before the SQL preprocessor and if you also had code in your SQL statements such as the following:

```
WHERE "system" = 'Wilmer'
```

Under the old compiler, this would have become `WHERE "SYSTEM" = 'Wilmer'`.

But under the new compiler, this becomes `WHERE "system" = 'Wilmer'`.

The latter would probably not produce the results you want from DB2. If this is the case, you must change your source so that the text delimited by "..." is in uppercase (before any preprocessing).

# SQL preprocessor differences

You might need to change your code because of differences between the old and new SQL preprocessors.

Starting with Enterprise PL/I for z/OS V4R2, the SQL preprocessor no longer supports the LOB option. If your program relies on how the preprocessor translates LOB declarations, you must change it. See “Migrating from Enterprise PL/I V4R1” on page 144 for detailed information about that and other SQL preprocessor changes.

# Chapter 15. Linking your new objects

This chapter describes factors you must consider when you link-edit an object module produced by the new Enterprise PL/I compiler.

For more information about linking your code, see the *Enterprise PL/I for z/OS Programming Guide*.

## Prelinker and PDSE considerations

As long as you use the Enterprise PL/I default compiler options NORENT and LIMITS(EXTNAME(7)), you do not need to use either the prelinker or PDSEs.

## AMODE(24) considerations

For AMODE(24) support, you must link your application program with SIBMAM24 concatenated in front of SCEELKED.

For more information about building AMODE(24) applications, see "AMODE(24) restrictions" on page 70.

## Using PLICALLA or PLICALLB entry

If you use PLICALLA or PLICALLB as a main entry point in an Enterprise PL/I program, you must concatenate SIBMCAL2 in front of SCEELKED.

## CHANGE cards

Enterprise PL/I does not support the use of CHANGE cards during link-edit if the RENT option is specified, or if the LIMITS(EXTNAME(*n*)) option is specified with *n* greater than 8.

© Copyright IBM Corp. 1999, 2017

# Chapter 16. Using Language Environment with the new compiler

When you use the Language Environment run time with Enterprise PL/I, be aware of the differences between Language Environment and the old OS PL/I run time.

Many of the same considerations that were discussed in Chapter 6, "Considerations before migrating," on page 35 apply to the new compiler as well. See that chapter for runtime considerations when you use the new compiler.

Chapter 13, "Understanding when working code must be changed," on page 101 also contains useful information about differences between Enterprise PL/I runtime results and previous versions of PL/I.

## Using the right runtime options

Under Language Environment, some of the options available under the OS PL/I run time are no longer available, and some have been renamed, redefined, or merged with other options. In addition, some important new options are now available.

### Dropped options

- COUNT
- FLOW

### Renamed and merged options

- HEAP redefines HEAP
- NATLANG replaces LANGUAGE
- RPTSTG replaces REPORT
- STACK merges ISASIZE and ISAINC
- TRAP merges SPIE and STAE

### Important new options

- ABTERMENC
- ALL31
- DEPTHCONDLMT
- ERRCOUNT
- MSGFILE
- STORAGE
- XUFLOW

For more and complete information about runtime options, see the *z/OS Language Environment Programming Reference*, but note the following key points:

- For compatibility with OS PL/I, use the following options:
  - ABTERMENC(RETCODE)
  - DEPTHCONDLMT(0)
  - ERRCOUNT(0)
  - TRAP(ON)
  - XUFLOW(ON)
- You must specify the following options in your AMODE(24) applications:
  - ALL31(OFF)
  - STACK(,,BELOW)

© Copyright IBM Corp. 1999, 2017

- Never use RPTSTG(ON) in any performance-critical application.
- Never use STORAGE(,,00) in any performance-critical application.
- You must specify POSIX(ON) in any multithreaded application.

## Calling PL/I from assembler main programs

There are three ways by which Language Environment-conforming assembler routines can pass control to a Enterprise PL/I subroutine.

1. Branch to a statically-linked Enterprise PL/I subroutine.
2. Use the Language Environment macro CEEFETCH to branch to a separately-linked Enterprise PL/I subroutine.
3. Use assembler instructions such as LOAD and BALR to branch to a separately-linked Enterprise PL/I subroutine.

   In this case, your must explicitly link in the Language Environment-Enterprise PL/I signature CSECT, CEESG011, to ensure that the Language Environment-PL/I-specific runtime environment is initialized.

For information about other assembler issues, see "Differences in assembler support" on page 46.

## Understanding when your results might vary

Under Language Environment, some return codes, runtime messages, dumps, and storage reports are different from what you get under the old OS PL/I run time. In addition, Enterprise PL/I now invokes Language Environment-provided math built-in routines, which are more precise than those of OS PL/I.

### Return codes

The PLIRETC built-in subroutine will now accept a FIXED BIN(31) argument and does not require the value to be <= 999.

Correspondingly, the PLIRETV built-in function will now return a FIXED BIN(31) value.

The Language Environment run time adds 3000 to the user return code for severity 3 conditions, and Language Environment classifies all PL/I conditions as severity 3 except the following:

- ATTENTION (when raised by a SIGNAL statement)
- CONDITION
- ENDPAGE
- FINISH
- NAME
- PENDING
- STRINGRANGE
- STRINGSIZE
- UNDERFLOW

### When the run time issues messages

Under Language Environment, there is a small difference in the timing of when some runtime messages are issued for conditions with ON-units.

- Without Language Environment, if a condition such as ZERODIVIDE or ERROR occurred, the run time will issue a message before invoking the ON-unit for the condition.

- With Language Environment, if a condition such as ZERODIVIDE or ERROR occurs, the run time will issue a message only if the END statement in the ON-unit is executed.

This change gives you the chance to handle a condition (and issue your own message if you wish) and to continue your application through a GOTO without the run time also issuing its own message.

There is no change to the runtime behavior when there is no ON-unit.

Also, the SNAP traceback message produced by ON ERROR SNAP continues to be issued before the ERROR ON-unit receives control.

When running Enterprise PL/I programs under Language Environment, some file I/O errors are now detected during the OPEN process, which results in a different but more meaningful error message and error code. As a result, the error will result in an UNDEFINEDFILE condition instead of a TRANSMIT or other condition that was received with older PL/I.

## What the runtime messages say

PL/I for MVS & VM and Enterprise PL/I share the same set of runtime messages, and this can lead to messages that should be read with understanding and flexibility.

For example, when the run time issues a message for UNDEFINEDFILE in an Enterprise PL/I program, the message will mention both MVS and VM constructs even though Enterprise PL/I does not currently support VM. The meaning should be clear nonetheless.

Also, if you compile with the compiler GONUMBER option, the runtime messages will refer to a "statement" where your exception has occurred. This "statement" is, for Enterprise PL/I, the line number in the source program of the statement that raised the exception.

Finally, the format and content of runtime messages are different under the Language Environment run time from that under the OS PL/I run time. You can find complete descriptions of the runtime messages in *z/OS Language Environment Run-Time Messages*.

## Where the runtime messages go

Under Language Environment, runtime messages go to the destination specified in the runtime option MSGFILE.

The default MSGFILE destination is SYSOUT, not SYSPRINT, as it was under the old run time. MSGFILE(SYSPRINT) is supported under Enterprise PL/I only after you apply the PTFs for the runtime APAR PQ78307.

## Math built-ins

The new compiler invokes the Language Environment-provided routines to evaluate the mathematical built-in functions (such as SIN or COS) and for float exponentiation. These routines are more precise than the routines provided with the OS PL/I V2R3 library and can sometimes produce results with a different last digit.

As an example of this difference, consider the following program, which produces the kind of table seen at the back of trigonometric textbooks:

```
trigtab: proc options(main);

  dcl degrees   fixed dec(5,1);
  dcl minutes   fixed dec(3,1);

  do degrees = 0 to 359;
    put skip edit( degrees ) ( f(5) );
    do minutes = 0 to .9 by .1;
      put edit( sind(degrees+minutes) ) ( f(9,4) );
    end;
  end;

end;
```

The output of this program is as follows:

```
 0   0.0000   0.0017   0.0035   0.0052   0.0070   ...
 1   0.0175   0.0192   0.0209   0.0227   0.0244   ...
```

The table produced depends on which math library is used, and even then there are only 5 different values. For instance, with the old compilers using the pre-LE math library, the result for 140.1 is 0.6414, while with the old compilers using the Language Environment math library, the result is 0.6415. Because the new compiler uses only the Language Environment math library, the result with it is also 0.6415.

## Dumps

Calling PLIDUMP still produces a dump, but the format, contents, and destination of dumps are now controlled by Language Environment.

For more information about the many resultant, but mostly small, differences, see "Differences in PLIDUMP" on page 44.

## Storage reports

The format, contents, and destination of the runtime storage report have changed.

For more and complete information about the runtime storage report, see the description of the RPTSTG option in the *z/OS Language Environment Programming Reference*.

Note that Language Environment does not use the PLIXHD declaration to provide the heading for the runtime storage report. You can, however, specify the heading by using the Language Environment's callable service CEE3RPH.

# Prerequisite Language Environment PTFs

This topic lists PTFs that are required to compile and run PL/I applications using Enterprise PL/I.

- For z/OS, Version 1 Release 4: PTF UQ70042 (APAR PQ66155) and PTF UQ88264 (APAR PQ88268).
- For z/OS, Version 1 Release 5: PTF UQ80236 (APAR PQ78173) and PTF UQ88263 (APAR PQ88065).
- For z/OS, Version 1 Release 6: PTFs UQ92073 and UQ92088 (APARs PQ92870 and PQ93118).
- For z/OS, Version 1 Release 7: PTF UK06652 (APAR PK10630).

# Chapter 17. Tuning for better CPU and storage utilization

After you migrate to Language Environment, you should retune your applications to maximize the performance. It is not always possible to maximize CPU and storage at the same time. Often, to obtain better CPU, you need to use more storage, or vice versa. This section provides general tips to help you to retune your applications under Language Environment.

For information about choosing compiler options for improved performance see "Choosing options for improved performance" on page 79.

For more information about tools you can use to improve performance for your applications, see the *z/OS Language Environment Programming Guide*, *z/OS Language Environment Installation and Customization under OS/390*, or *z/OS Language Environment Customization*, and *Enterprise PL/I for z/OS Programming Guide*.

## Improving CPU utilization

This topic provides tips that help you obtain better CPU utilization.

- Reduce the number of GETMAINs and FREEMAINs issued by Language Environment.

  Use the Language Environment RPTSTG(ON) option to produce the storage report. Specify the reported storage amount in the corresponding Language Environment storage runtime options.
- Reduce the number of LOADs and DELETEs issued by Language Environment.

  Put the commonly used Language Environment library routines in (E)LPA. The following list shows the recommended candidates for PL/I:
  - CEEBINIT (LPA)
  - CEEPLPKA (ELPA)
  - CEEEV010 (ELPA) if you still have OS PL/I applications
  - CEEEV011 (ELPA) for Enterprise PL/I applications
  - CEEBLIIA (LPA) for OS PL/I applications not relinked
  - IBMRLIB1 (LPA)

  See *z/OS Language Environment Installation and Customization under OS/390* or *z/OS Language Environment Customization* for a complete list of library routines that can be put in (E)LPA.
- Avoid AMODE switching between library routines.

  Use AMODE(31) for your application, if possible, so you can specify Language Environment ALL31(ON) option. If ALL31(ON) is in effect, there will be no AMODE switching among library routines.
- Avoid overuse of PL/I conditions.

  All PL/I condition handling is expensive and should only be used where appropriate. Overuse of PL/I condition handling will degrade the performance of your application.
- Use DF/SMS-provided system-determined BLKSIZE.

  On MVS, use BLKSIZE(0) for an output file that can be blocked. DF/SMS determines the optimal block size for you, which can improve the file performance.
- Use Language Environment Library Routine Retention facility (LRR).

© Copyright IBM Corp. 1999, 2017

You can get a better CPU performance if you use LRR. When LRR is used, Language Environment keeps certain Language Environment resources in storage when an application ends. Subsequent invocations of programs that use LRR is much faster because the Language Environment resources left in storage are reused.

For example, you can use LRR for your IMS/DC environment to improve performance.

Note that because LRR leaves Language Environment resources in the storage for a long period of time, you must assess your storage availability to accommodate the situation.

## Improving storage utilization

This topic provides tips that help you obtain better storage utilization.

- Relink with Language Environment if you have not recompiled your OS PL/I programs.

  The relinked OS PL/I load module has a smaller size because it contains the Language Environment stubs only.
- Make your application AMODE(31) and RMODE(ANY).

  Most likely the application will be loaded above the 16M line. You can specify the Language Environment ALL31(ON) option, which allows Language Environment to allocate some of its control blocks above the 16M line.
- Avoid use of the HEAPPOOLS(ON) option.

  The HEAPPOOLS option applies to Enterprise PL/I (although not to PL/I for MVS & VM and earlier PL/I code). Specifying the HEAPPOOLS(ON) option might lead to a very large amount of storage being allocated to ANYHEAP.
- Use Language Environment option HEAP(,,ANY) option, if possible.

  For PL/I, Language Environment will allocate the heap storage above the 16M line if the following is true:
  - The requestor is in AMODE(31).
  - HEAP(,,ANY) is in effect.
  - The main program is in AMODE(31).
- Use Language Environment STACK(,,ANY) option, if possible.

  Your application must be in AMODE(31). For PL/I, Language Environment will allocate the stack storage above the 16M line if your application is relinked with Language Environment and contains no edited stream I/O.
- Analyze the IBM-supplied default values in Language Environment storage options and change them, if possible and as necessary, to make them optimal for your applications.

  Note that specifying a smaller value is not always better: if you use a smaller value, Language Environment will allocate less storage initially, but this could result in more GETMAINs and FREEMAINs being issued over the life of the application - and GETMAINs are very expensive.
- Put commonly used Language Environment library modules in the (E)LPA.

  The library routines in (E)LPA do not occupy storage in your application region, so your application has more storage to use. See the recommended library routines for (E)LPA in "Improving CPU utilization" on page 131.

## Improving performance under subsystems

This topic provides tips that help you obtain better performance under specific subsystems.

## Under CICS

Use the PL/I FETCH/CALL statement instead of EXEC CICS LINK. The PL/I FETCH/CALL statement has a much shorter path length than the path length of EXEC CICS LINK.

## Under IMS

Use Language Environment Library Routine Retention (LRR) facility to reduce the number of LOADs/DELETEs and GETMAINs/FREEMAINs issued by Language Environment for each transaction.

Preload commonly used Language Environment library modules and frequently used top-level applications.

In particular, it is especially beneficial for programs with any I/O to preload the module IBMPOIOA or to put IBMPOIOA in the LPA.

# Chapter 18. Adding Enterprise PL/I programs to existing PL/I applications

When you add an Enterprise PL/I program to an existing application, you are either recompiling an existing program with Enterprise PL/I or including a newly written Enterprise PL/I program. When you add Enterprise PL/I programs to your existing applications, you have the ability to upgrade your existing programs incrementally, as your shop's needs dictate. This chapter provides information about object and load module considerations and about condition handling in mixed applications.

**Important:**

- After you add an Enterprise PL/I program to an existing application, that application must run under Language Environment.
- You cannot mix new and old object code if the old code does any multitasking.
- If you mix old and new code, you cannot do any FETCH from FETCH.

## Object and load module considerations

It is strongly recommended that you recompile all your PL/I source. However, if you do not recompile your source, and if you want to mix Enterprise PL/I code with older PL/I objects, you must be aware of the options that are recommended for compiling new code and of the restrictions on mixing old and new object code.

You must use the following options when compiling Enterprise PL/I code that will be mixed with older PL/I objects:

- CMPAT(V2)

  **Note:** If your old code was compiled with CMPAT(V1), you must recompile it with CMPAT(V2) or CMPAT(V3).
- LIMITS( EXTNAME(7) )
- NORENT
- BACKREG(5)
- BIFPREC(15)

In addition, as discussed in Chapter 11, "Understanding the new compiler's options," on page 73, you might also want to use some or all of these options:

- COMMON
- DEFAULT( LINKAGE( SYSTEM ) )
- DEFAULT( OVERLAP )
- EXTRN( FULL )
- NOWRITABLE(PRV)

Note that unless you use the NOWRITABLE(PRV) option, CONTROLLED variables cannot be shared between old and new code.

Even if all the options listed above are used, there are some restrictions on mixing old and new object code:

© Copyright IBM Corp. 1999, 2017

- FILE variables and constants cannot be shared between old and new code with one exception: SYSPRINT can be shared by old and new code if the old code was linked under LE. However, a file written out by old code can be read by new code - and vice-versa.
- Whenever old code is used, all fetch/release restrictions from the older product apply. In particular, if a new MAIN does successfully FETCH and CALL an old module, the old module cannot perform a subsequent FETCH of another module.
- If any old code is present in an application, DLL code cannot be invoked.
- There is no support for mixing OS PL/I V1R4 object with Enterprise PL/I objects.
- For old code compiled with OS PL/I V2R3 or earlier (but later than V1R4), these restrictions apply:
  - An old MAIN not linked with Language Environment cannot FETCH a new module.
  - A new MAIN cannot CALL or FETCH an old module unless under either of the following conditions:
    - The new MAIN has the signature CSECT CEESG010 linked in.
    - The old module has been relinked with SCEELKED either after you have applied the PTF for APAR PK23270, or with an explicit INCLUDE SYSLIB(CEESG010).

Previously, Enterprise PL/I had the restriction that if your old code did any I/O, MAIN must have been compiled with an old compiler. This restriction no longer applies if you are using Language Environment 1.10 or later.

## Sharing SYSPRINT

With the enhancement delivered with co-req APAR PK01919 (Enterprise PL/I) and PK016197 (PL/I for MVS & VM), SYSPRINT can be shared between Enterprise PL/I and PL/I for MVS & VM at the enclave level and also in a multi-enclave environment.

Below are the restrictions and the extent to which this shared SYSPRINT is supported:

- The compiler option STDSYS must not be used.
- SYSPRINT must have the default or declared attributes: EXTERNAL, STREAM, OUTPUT, or PRINT.
- The shared SYSPRINT could be directed to SYSOUT or to a permanent data set.
- Shared SYSPRINT is supported when MSGFILE(SYSPRINT) is specified provided that there are no preinitialized programs and/or stored procedures in the mix.
- In a multi-enclave environment, the first SYSPRINT that is opened will determine the attributes of SYSPRINT. The second and subsequent SYSPRINT will inherit all attributes from the first SYSPRINT.
- SYSPRINT will remain opened during the entire application. At enclave termination, all other files will get closed except for SYSPRINT which will only be closed at process termination.
- An explicit close of the shared SYSPRINT by either Enterprise PL/I or PL/I for MVS & VM is honored. Any attempt to write to SYSPRINT afterward requires

SYSPRINT to be explicitly or implicitly opened again. If SYSPRINT was routed to a data set that is reused for the second open, data previously written might be lost.

- SYSPRINT can only be opened (explicitly or implicitly) by the initial thread (Enterprise PL/I multithreading) or by the main task (PL/I for MVS & VM multitasking). Secondary thread and subtask should not explicitly or implicitly open SYSPRINT and should not explicitly close SYSPRINT.
- SYSPRINT cannot be shared with older PL/I under TSO.

With support for shared SYSPRINT, the overriding of attributes has changed in the following ways:

- When SYSPRINT is routed to SYSOUT, the SYSPRINT attributes specified in the ENVIRONMENT option or the OPEN statement are allowed to override those options specified on the DD statement.
- When SYSPRINT is routed to a data set (either TEMPORARY, NEW, or OLD), any mismatch between the attributes specified by the program and those specified on the DD statment will cause the UNDEFINEDFILE condition to be raised.

To aid in migration, APAR PK63659 introduces a new temporary environment variable, PLI_SYSPRINT_ATTR_OVERRIDE. To get the same behavior as before the shared SYSPRINT changes, specify PLI_SYSPRINT_ATTR_OVERRIDE=YES in the PARM parameter or in the PLIXOPT string. This will allow attribute overrides when SYSPRINT is routed to a TEMPORARY or NEW data set. Note that attribute overriding is never allowed when SYSPRINT is routed to an existing or OLD data set, and that it is always allowed when SYSPRINT is routed to SYSOUT.

Also note that support for this new environment variable is only temporary. Starting with Language Environment 1.10Z this environment variable will be ignored. Affected programs and JCL will need to be changed or the UNDEFINEDFILE condition will be raised.

## Runtime option considerations

The HEAPPOOLS option cannot be used in mixed old and new PL/I code if the old PL/I code tries to free storage allocated by new PL/I code.

## Condition handling considerations

For the purposes of condition handling, you must consider old PL/I programs and Enterprise PL/I programs as separate languages.

Both old PL/I and Enterprise PL/I have their own signature CSECTs (CEESG010 for OS PL/I and PL/I for MVS & VM, and CEESG011 for VisualAge PL/I and Enterprise PL/I), and separate runtime libraries in Language Environment.

This implies that when software conditions are raised in a PL/I source program on one side (either old or new PL/I) and is expected to be handled by a PL/I source program on the other side (new PL/I if it was raised in old PL/I, or old PL/I if it was raised in new PL/I), the program that is supposed to handle the exception will not even know about it because it uses a completely separate runtime library than the program that raised the condition.

Hardware conditions (such as ZERODIVIDE) have a better chance of being handled correctly across the old PL/I/new PL/I boundary because Language Environment gets involved and bridges the gap between the two separate PL/I runtime libraries.

## Partitioning PL/I source programs into units of execution

You will need to partition your PL/I source programs into units of execution to accommodate the restrictions on mixing and condition handling between old and new PL/I modules.

Pay careful attention when partitioning your PL/I source programs into units of execution. Your goal is to contain any restrictions on mixing old and new PL/I modules within the boundaries of the units of execution that you define. For example, if Program A defines a CONTROLLED EXTERNAL variable, Program B references this variable, and Program B also creates a file variable that it shares with Program C, then you must compile all three Programs A, B, and C with Enterprise PL/I so that they can work correctly.

Finally, note that when mixing old and new code, you must pay attention to the differences between how the new and old compilers handle various language constructs, as described in Chapter 13, “Understanding when working code must be changed,” on page 101.

# Chapter 19. Migrating from earlier releases of Enterprise PL/I to Enterprise PL/I V5R1

This *Migration Guide* concentrates on the migration effort in migrating from OS PL/I or PL/I for MVS & VM to Enterprise PL/I V5R1. If you have already moved to Enterprise PL/I Version 3 Releases, V4R1, V4R2, V4R3, V4R4, or V4R5, migration to Enterprise PL/I V5R1 is relatively easy.

This chapter focuses on differences in the compiler options and in the compiler messages, but there are some other differences in the compiler output that might possibly affect users of earlier release of Enterprise PL/I:

- The compiler itself is compiled with ARCH(8) and any use of it on a machine with older hardware causes the compiler to stop.
- The macro preprocessor now leaves %include %xinclude, %inscan, and %xinscan in the compiler listing as comments.
- Listings now include 7 columns for the line number in a file.
- The MAP output now also includes a list in order of storage offset (per block) of the AUTOMATIC storage used by the block.
- The length of the mnemonic field in the assembler listing is increased to allow for better support of the new z/OS instructions that have long mnemonics.
- More of the right margin is used in the attributes, cross-reference, and message listings.
- There are some small changes in the SYSADATA produced by the compiler:
  - The chaining of the procedure records and their associated statements are changed so that the block structure of a compilation is readily determined (more details are available in the appendix in the *Programming Guide*).
  - The edition and sysadata level numbers are updated (and these values could be used to allow code to handle both the old and new chaining of the procedure recodes).
- The MAXNEST option can flag some old code in which the nesting of DO, IF, or PROCEDURE statements is too deep.
- Under OPT(3), optimization will no longer be turned off for programs where the limit in the MAXSTMT option is exceeded, and this might cause some compiles (particularly V4 compilers) to consume much greater CPU time and use much more storage as well.

## Migrating from Enterprise PL/I V4R5

The V5R1 compiler, like the V3R9, V4R1, V4R2, V4R3, V4R4, V4R5 compilers, must be installed in a PDSE. Also, the Language Environment runtime option XPLINK must be ON whenever you start the compiler. If you start the compiler under batch by using IBMZPLI or under z/OS UNIX System Services by using the **pli** command, the compiler itself ensures that it runs with XPLINK(ON). However, if you are starting the compiler in some other way, you must ensure that XPLINK(ON) is in effect.

The compiler now issues a warning message if the option strings in the IBMZIOP are the same as each other and if a user specifies an option that conflicts with one of those non-overridable options.

© Copyright IBM Corp. 1999, 2017

To make it easier to change the severity of the DSNH030 SQL preprocessor message, it is now given its own preprocessor message IBM3317.

Enterprise PL/I V5R1 contains some new options, some options with new suboptions, and some options with changed defaults. In particular, the default for the RULES suboption LAXDCL has been changed to NOLAXDCL, which will cause some compilations that previously ended with a return code of 0 to end with a return code of 8.

## New or changed options in Enterprise PL/I V5R1

**New options**

- BRACKETS
- DECOMP | NODECOMP
- LP (LP(32) is the default.)
- NULLDATE | NONULLDATE
- OFFSETSIZE (OFFSETSIZE(4) is the default.)

**Existing options with new suboptions**

- The (NO)TRUNCFLOAT suboption of DECIMAL
- The FIRST and ALL suboptions of INSOURCE
- The (NO)LAXSTMT and (NO)UNREFBASED suboptions of RULES
- The NOPROCENDONLY(ALL|SOURCE) suboption of RULES
- The EXPLICIT and IMPLICIT suboptions of XREF

**Existing options with changed defaults**

| Option | Default in V5R1 | Default in V4R5 |
|---|---|---|
| ARCH | 8 | 7 |
| LIMITS | FIXEDDEC(15,31) | FIXEDDEC(15) |
| RULES | NOLAXDCL | LAXDCL |
| | NOLAXIF | LAXIF |
| | NOLAXSCALE | LAXSCALE |
| | NOMULTICLOSE | MULTICLOSE |

**Existing options with dropped suboptions**

- The 7 suboption of ARCH
- The AFP and NOAFP suboptions of FLOAT
- The V1 and LE suboptions of CMPAT

**Existing options with changed equivalents**

LIMITS(FIXEDBIN(31)) used to be equivalent to LIMITS(FIXEDBIN(31,31)) in V4R5, but it is equivalent to LIMITS(FIXEDBIN(31,63)) in V5R1. LIMITS(FIXEDBIN(31,31)) is no longer supported in V5R1.

**Dropped options**

- HGPR | NOHGPR

All the compiler option default settings for a release is listed in the member with the name PXOPT*vr*, where *v* is the version number and *r* is the release number (for example, PXOPT51 for V5R1 and PXOPT45 for V4R5), in the SIBMZSAM data set. The data set contains PXOPT*vr* members for all supported PL/I releases. To see what has changed in the default settings and what new options are added from one

release to the next, you can compare the PXOPTvr files for the two releases. You can also use the PXOPTvr files as templates for creating your own options files with your preferred settings.

## Migrating from Enterprise PL/I V4R4

The V4R5 compiler, like the V3R9, V4R1, V4R2, V4R3, and V4R4 compilers, must be installed in a PDSE. Also, the Language Environment runtime option XPLINK must be ON whenever you start the compiler. If you start the compiler under batch by using IBMZPLI or under z/OS UNIX System Services by using the **pli** command, the compiler itself ensures that it runs with XPLINK(ON). However, if you are starting the compiler in some other way, you must ensure that XPLINK(ON) is in effect.

Enterprise PL/I V4R5 contains some new options and some old options with new suboptions. However, the defaults for these options, with the exceptions of the CMPAT(V3) option and the FIXEDBIN suboption of the LIMITS option, produce executable code that is compatible with the code that is produced by the Enterprise PL/I V4R4 compiler or any of the releases since V3R3.

All code compiled with earlier releases and CMPAT(V3) must be recompiled.

In some conversions, the value of M2 in the LIMITS(FIXEDBIN(M1,M2)) option partly determines the attributes of the results. Hence, the change in the default for this option could cause the compiler to generate code that would produce different results. Here are some examples:

1. When converting from FIXED DEC(p,q) to FIXED BIN, the source is converted to FIXED BIN with a precision equal to 1 + MIN( CEIL(p*3.32), M2 ). So, if a FIXED DEC(15) is converted to FIXED BIN, under LIMITS(FIXEDBIN(31,31)), it is converted to FIXED BIN(31). However, under LIMITS(FIXEDBIN(31,63)), it is converted to FIXED BIN(51).
2. When converting from FIXED DEC(p,q) to BIT, the source is first converted to FIXED BIN with a precision equal to MIN( CEIL((p-q)*3.32), M2 ). So, if a FIXED DEC(11) is converted to BIT, under LIMITS(FIXEDBIN(31,31)), it is converted to BIT(31), but under LIMITS(FIXEDBIN(31,63)), it is converted to BIT(37).
3. When converting from BIT to FIXED BIN, the source is converted to FIXED BIN(M2).

If you use with PL/I V4R5 the same settings for your compiler options as you used with the V4R4, V4R3, V4R2, V4R1, and Version 3 releases, you can mix code compiled with V4R5 and earlier releases. You do not need to recompile all your code unless you change the setting of a compiler option that changes the program semantics. For example, you can change the ARCH or RULES option when mixing objects, but you cannot mix objects if you change the BACKREG, BIFPREC, or CMPAT options.

Some programs that used to compile might now fail for these reasons:

- The compiler now flags any compilation unit that has a FETCH statement that tries to fetch itself.
- The compiler now objects if preprocessor suboptions are not enclosed in quotation marks. For example, the compiler objects to specifying PP(MACRO(CASE(ASIS))), but it still accepts the correct PP(MACRO('CASE(ASIS)')).

### New or changed options in Enterprise PL/I V4R5

**New options**

- FILEREF | NOFILEREF
- MAXBRANCH

**Existing options with new suboptions**

- The FORCE suboption of the RULES(NOLAXQUAL) option.
- The ALL and SOURCE suboptions of the RULES(NOLAXNESTED) and RULES(NOPADDING) options.

## Migrating from Enterprise PL/I V4R3

The V4R4 compiler, like the V3R9, V4R1, V4R2, and V4R3 compilers, must be installed in a PDSE. Also, the Language Environment runtime option XPLINK must be ON whenever you start the compiler. If you start the compiler under batch by using IBMZPLI or under z/OS UNIX System Services by using the **pli** command, the compiler itself ensures that it runs with XPLINK(ON). However, if you are starting the compiler in some other way, you must ensure that XPLINK(ON) is in effect.

Enterprise PL/I V4R4 contains some new options and some old options with new suboptions. However, the defaults for these new options and suboptions produce executable code that is compatible with the code that is produced by the Enterprise PL/I V4R3 compiler or any of the releases since V3R3.

If you use with PL/I V4R4 the same settings for your compiler options as you used with the V4R3, V4R2, V4R1, and Version 3 releases, you can mix code compiled with V4R4 and earlier releases. You do not need to recompile all your code unless you change the setting of a compiler option that changes the program semantics. For example, you can change the ARCH or RULES option when mixing objects, but you cannot mix objects if you change the BACKREG, BIFPREC, or CMPAT options.

### New or changed options in Enterprise PL/I V4R4

**New options**

- INCLUDE | NOINCLUDE

**Existing options with new suboptions**

- The STRICT suboption of the NULLSTRPTR suboption of the DEFAULT option.

**Existing options with new functions**

- When the STMT option is specified, the source and message listings will include both the logical statement numbers and the source file numbers.

### Changes to the return code

Starting from V4R4, the compiler flags the declarations for variables that do not have the STATIC attribute but have more than 100 INITIAL items. The compilation of the program that includes these declarations ended with a return code of 0, but now it ends with a return code of 4.

The CICS and SQL preprocessors have never supported the DFT(ASCII) compiler option, and they now flag the use of the DFT(ASCII) compiler option with an S-level message. This new S-level message will lead the compilation to end with a return code of 12.

Under the SQL preprocessor, WIDECHAR is always assigned a CCSID value of 1200.

## Migrating from Enterprise PL/I V4R2

The V4R3 compiler, like the V3R9, V4R1, and V4R2 compilers, must be installed in a PDSE. Also, the Language Environment runtime option XPLINK must be ON whenever you start the compiler. If you start the compiler under batch by using IBMZPLI or under z/OS UNIX System Services by using the **pli** command, the compiler itself ensures that it runs with XPLINK(ON). However, if you are starting the compiler in some other way, you must ensure that XPLINK(ON) is in effect.

Enterprise PL/I V4R3 contains some new options and some old options with new suboptions. However, the defaults for these new options and suboptions produce executable code that is compatible with the code that is produced by the Enterprise PL/I V4R2 compiler (or any of the releases since V3R3).

If you use with PL/I V4R3 the same settings for your compiler options as you used with the V4R2, V4R1, and Version 3 releases, you can mix code compiled with V4R3 and earlier releases. You do not need to recompile all your code unless you change the setting of a compiler option that changes the program semantics. For example, you can change the ARCH or RULES option when mixing objects, but you cannot do so if you change the BACKREG, BIFPREC, or CMPAT options.

### New or changed options in Enterprise PL/I V4R3

**New options**

- CASERULES
- DEPRECATENEXT
- MSGSUMMARY

**New SQL options**

- ONEPASS | TWOPASS. The ONEPASS SQL option is restored in V4R3. It was dropped in V4R2.
- DEPRECATE

**Existing options with new suboptions**

- The 10 suboption of the ARCH option
- The STMT suboption of the DEPRECATE option
- The ASSERT suboption of the IGNORE option
- The NULL370 suboption of the RTCHECK option
- The (NO)CONTROLLED, (NO)LAXNESTED, and (NO)RECURSIVE suboptions of the RULES option
- The SOURCE | ALL suboption of the RULES(NOUNREF) option

**Existing options with dropped suboptions**

- The 5 suboption of the ARCH option.
- The (NO)STOP suboption of the RULES option. You can use DEPRECATE(STMT(STOP)) to achieve the same functionality.

- The XMI suboption of the XINFO option.

### Preprocessor message number changes

With PL/I V4R3, the numbers of the messages that report CICS and SQL backend errors are changed. The new numbers are listed as follows. Make appropriate updates if you use the EXIT option to change the severity of these messages or to track their occurrences.

- IBM3000I I for informational messages from the back end.
- IBM3250I W for warning messages from the back end.
- IBM3500I E for error messages from the back end.
- IBM3750I S for severe messages from the back end.

## Migrating from Enterprise PL/I V4R1

The V4R2 compiler, like the V3R8, V3R9, and V4R1 compilers, must be installed in a PDSE. Also, the Language Environment runtime option XPLINK must be ON whenever you start the compiler. If you start the compiler under batch by using IBMZPLI or under z/OS UNIX System Services by using the **pli** command, the compiler itself ensures that it runs with XPLINK(ON). However, if you are starting the compiler in some other way, you must ensure that XPLINK(ON) is in effect.

Enterprise PL/I V4R2 contains some new options and some old options with new suboptions. However, the defaults for these new options and suboptions produce executable code that is compatible with the code that is produced by the Enterprise PL/I V4R1 compiler (or any of the releases since V3R3).

If you use with PL/I V4R2 the same settings for your compiler options as you used with the V4R1 and Version 3 releases, you can mix code compiled with V4R2 and earlier releases. You do not need to recompile all your code unless you change the setting of a compiler option that changes the program semantics. For example, you can freely change the ARCH or RULES option when mixing objects, but you cannot mix objects if you change the BACKREG, BIFPREC, or CMPAT options.

### New or changed options in Enterprise PL/I V4R2

**New options**

- PPLIST
- UNROLL

**Existing options with new suboptions**

- CHECK supports (NO)STORAGE as a suboption.
- DEFAULT supports (NO)PSEUDODUMMY as a suboption.
- RULES supports NOLAXENTRY(LOOSE | STRICT), (NO)LAXRETURN, and (NO)SELFASSIGN as suboptions.

### SQL preprocessor differences from Enterprise PL/I V4R1 and Version 3

Note the following differences between the SQL preprocessors from the new and the old compiler:

**Dropped SQL preprocessor options**

The V4R2 compiler dropped support for the following SQL preprocessor options:

- LOB(DB2 | PLI)
- ONEPASS | TWOPASS

  The preprocessor now always acts as if the TWOPASS option were on.
- SCOPE | NOSCOPE

  In effect, SCOPE is always on. However, the restrictions imposed previously when using SCOPE have been removed. The SQL preprocessor now resolves names using the same rules as the compiler.

**Handling of LOB declarations**

The SQL preprocessor no longer supports the LOB option. If your program relies on how LOBs are represented in the code that is generated by the SQL preprocessor, you must change the program.

You can now use SQL TYPE anywhere other PL/I data attributes can be used. Therefore, you can eliminate any dependency on how the preprocessor translates LOB declarations in your code, and thus write simpler and cleaner code.

For instance, the variable XML_DOC_STRUC in the following example depends on a particular implementation of the CLOB type. Therefore, the compiler cannot compile the code if you use the SQL preprocessor from V4R2.

```
DCL
  1 DOCM_STRUC,
    2 MODEL_EXECN_ID_STRUC  FIXED BIN(31),
    2 DOCM_TYPE_CD_STRUC    CHAR(1),
    2 XML_DOC_STRUC,
      3 XML_DOC_ARRY_LENGTH FIXED BIN(31),
      3 XML_DOC_ARRY_DATA,
        4 XML_DOC_DATA1(3)  CHAR(32767),
        4 XML_DOC_DATA2     CHAR(4099);

DCL MODEL_EXECN_ID_ARRY(5)  FIXED BIN(31);
DCL DOCM_TYPE_CD_ARRY(5)    CHAR(1);
DCL XML_DOC_ARRY(5)         SQL TYPE IS XML AS CLOB(100K);

EXEC SQL FETCH NEXT ROWSET FROM DOCM_CSR FOR 5 ROWS
          INTO  :MODEL_EXECN_ID_ARRY
               ,:DOCM_TYPE_CD_ARRY
               ,:XML_DOC_ARRY;
XML_DOC_STRUC = XML_DOC_ARRY(I);
```

You can change the code in the previous example to the following code by using SQL TYPE. The compiler can compile it with the SQL preprocessor or the precompiler from V4R2, but you cannot compile it with the preprocessor from V4R1 or earlier releases.

```
DCL
1 DOCM_STRUC,
  2 MODEL_EXECN_ID_STRUC  FIXED BIN(31),
  2 DOCM_TYPE_CD_STRUC    CHAR(1),
  2 XML_DOC_STRUC         SQL TYPE IS XML AS CLOB(100K);

DCL MODEL_EXECN_ID_ARRY(5)  FIXED BIN(31);
DCL DOCM_TYPE_CD_ARRY(5)    CHAR(1);
DCL XML_DOC_ARRY(5)         SQL TYPE IS XML AS CLOB(100K);

EXEC SQL FETCH NEXT ROWSET FROM DOCM_CSR FOR 5 ROWS
         INTO  :MODEL_EXECN_ID_ARRY
              ,:DOCM_TYPE_CD_ARRY
              ,:XML_DOC_ARRY;

XML_DOC_STRUC = XML_DOC_ARRY(I);
```

**Invalid host variable references**

The new SQL preprocessor flags with warning messages the following two kinds of invalid host variable references that the old preprocessor did not flag:

- Arrays of structures and structures that contain arrays are not valid as host references. Take the following declarations as an example:

```
dcl 1 A(<bounds>), 2 B <data-type>;
dcl 1 A, 2 B (<bounds>) <data-type>;
```

  If you use A as a host reference, the old SQL preprocessor would accept the reference as valid. However, the new SQL preprocessor flags it with a warning message. The reference should be changed to A.B.

- Structures that contain structures are not valid as a host reference. Take this declaration as an example:

```
dcl 1 X, 2 X, 3 Y <data-type>, 3 Z <data-type>, ... /* and no other level-2 items */
```

  If you use X as a host reference, the old SQL preprocessor would accept it as valid. The new SQL preprocessor flags it with a warning message. The reference should be changed to X.X.

**Handling of SQL preprocessor messages**

Before Enterprise PL/I for z/OS V4R2, the facility ID was SQL for messages that were produced by the back end of the SQL preprocessor. You could use the IBM-supplied compiler user exit (IBMUEXIT) to suppress these messages or to change the severity of them.

As of V4R2, the facility ID of all messages is `IBM`, and you cannot change the severity of these messages by using the IBM-supplied IBMUEXIT.

However, you can change the severity of DB2 messages or suppress them entirely by modifying IBMUEXIT. For details on how to do this, see "Chapter 22. Using user exits" in the *Enterprise PL/I for z/OS V4R2 Programming Guide*.

# Migrating from Enterprise PL/I Version 3 (all releases)

Enterprise PL/I V4R1 contains some new options and some old options with new suboptions. However, the defaults for these new options and suboptions produce executable code that is compatible with the code that is produced by the Enterprise PL/I V3R9 compiler (or any of the releases since V3R3).

If you use with PL/I Version 4 the same settings for your compiler options as you used with the Version 3 releases, you can mix code compiled with V4R1 and earlier releases. You do not need to recompile all your code unless you change the setting of a compiler option that changes the program semantics. For example, you can freely change the ARCH or RULES option when mixing objects, but you cannot mix objects if you change the BACKREG, BIFPREC, or CMPAT options.

## New or changed options in Enterprise PL/I V4R1

The new options and changed options are listed as follows. They are described fully in the *Programming Guide*.

**New options**

- DEPRECATE
- XREF

**Existing options with added suboptions**

- ARCH supports 9 as a suboption.
- GONUMBER supports (NO)SEPARATE as a suboption.
- RULES supports (NO)GLOBALDO and (NO)PADDING as suboptions.

**Existing options with dropped suboptions**

- The STORAGE suboption of the CHECK option
- The SAA and SAA2 suboptions of the LANGLVL option

**Dropped SQL options**

- (NO)OPTIONS. The SQL preprocessor options are always listed.

## Changes in Enterprise PL/I Version 3 releases

This information lists some changes that have been made into earlier Enterprise PL/I Version 3 releases:

**Enterprise PL/I V3R9**

- Since Enterprise PL/I V3R9, REORDER rather than ORDER is the default suboption for the DEFAULT option.
- The V3R9 compiler also dropped support for the COMPACT and TUNE options.

**Enterprise PL/I V3R8**

Since Enterprise PL/I V3R8, because of the new V3 suboption to CMPAT, some of the message inserts generated by the compiler is 8-byte integers of type FIXED BIN(63). This change has no effect unless you write your own routine to be invoked by the EXIT compiler option. In this case, if you have a SELECT statement for the possible types of message inserts, you would probably have to add a new WHEN clause to that SELECT statement.

**Enterprise PL/I V3R7**

Since Enterprise PL/I V3R7, the documentation for the following built-in functions is removed and since V3R8 they are no longer supported:

- ACOSF
- ASINF
- ATANF
- COSF
- EXPF
- LOGF
- LOG10F
- SINF
- TANF

**Enterprise PL/I V3R6**

Note that only under V3R6, the default for CEESTART option is CEESTART(LAST). This makes the compiler place the CEESTART CSECT at the end of its generated object deck. Though this is required if you are using linker CHANGE cards, it is different from what was done under earlier releases of the compiler.

Moreover, if you do not use an ENTRY CEESTART linker card when binding your objects, this causes your code to behave incorrectly. You might prefer to use the CEESTART(FIRST) option.

**Enterprise PL/I V3R5**

Since Enterprise PL/I V3R5, when you specify the PP option more than once, the compiler behavior is changed. Before V3R5, the last specification would replace any previous specification, but since V3R5, the option is additive (as are the RULES and other options). So, if you specify PP(CICS) PP(SQL), it is the same as if you specify PP(CICS SQL).

# Messages that are introduced with V5R1

This topic includes new or changed messages that are introduced with V5R1, which includes both compiler messages and preprocessor messages.

## New and changed compiler messages

New and changed messages introduced with V5R1 are listed as follows. Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM1329: flags RULES(NOMULTIENTRY) violations
- IBM1479: flags RULES(NOMULTIEXIT) violations
- IBM2297: flags use of ALLOC31 without LP(64)
- IBM2298: flags use of ALLOCSIZE without CHECK(STG)
- IBM2299: flags invalid ranges in BETWEEN
- IBM2300: flags disabling of SMF
- IBM2301: flags problems with registration of SMF
- IBM2302: flags options unsupported under LP(64)
- IBM2303: flags unsupported codepage values
- IBM2304: flags attributes not supported under CMPAT(V1)
- IBM2305: flags invalid compare operators in ASSERT COMPARE
- IBM2306: flags invalid compare operators in ASSERT COMPARE
- IBM2307: flags invalid BINSEARCH
- IBM2308: flags invalid BINSEARCH
- IBM2309: flags invalid BINSEARCH
- IBM2310: flags invalid BINSEARCHX
- IBM2467: flags RULES(NOYY) violations
- IBM2468: flags RULES(NOYY) violations
- IBM2469: flags RULES(NOYY) violations
- IBM2470: flags RULES(NOYY) violations
- IBM2471: flags RULES(NOYY) violations
- IBM2472: flags RULES(NOYY) violations
- IBM2473: flags RULES(NOLAXINTERFACE) violations
- IBM2606: flags poor choice of attributes for REFER (used to be IBM1045)
- IBM2464: flags RULES(NOLAXSTMT) violations
- IBM2465: flags invalid assignments under NULLSTRPTR option
- IBM2466: flags invalid comparisons under NULLSTRPTR option
- IBM2654: flags INITIAL with BASED on ADDR
- IBM2655: flags if options conflict with overridable options
- IBM2656: flags occurrences of simple DEFINED

- IBM2825: flags BIT conversions by library calls
- IBM2826: flags BIT conversions by library calls
- IBM2831: flags unreachable ASSERT statements

### New and changed preprocessor messages

New and changed messages introduced with V5R1 are listed as follows. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM3317: flags the DB2 message DSNH03
- IBM3605: flags duplicate type definitions
- IBM3606: flags duplicate identifiers in a structure definition
- IBM3607: flags unsigned ordinals with negative
- IBM3608: flags ordinals with a precision too small for their values
- IBM3661: flags invalid use of question marks in preprocessors
- IBM3849: flags undefined type
- IBM3850: flags use of a type before its definition
- IBM3851: flags ordinals with values that are not increasing
- IBM3852: flags ordinals with values too large for a 4-byte integer
- IBM3857: flags run-on structure descriptions
- IBM3858: flags ordinals with no names not previously defined
- IBM3859: flags structure definitions with storage attributes
- IBM3860: flags structure definitions that specify arrays of structures
- IBM3950: flags invalid use of asterisk iteration factors
- IBM3951: flags invalid use of asterisk iteration factors
- IBM3952: flags use of INITIAL with macro parameters
- IBM3953: flags INITIAL attributes that contain too many initial values

## Messages that are introduced with V4R5

This topic includes new or changed messages that are introduced with V4R5, which includes both compiler messages and preprocessor messages.

### New and changed compiler messages

New and changed messages introduced with V4R5 are listed as follows. Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM1700: flags AREAs if BYVALUE
- IBM1894: flags REINIT reference if it has subscripts
- IBM2120: flags AREAs in RETURNS
- IBM2121: flags AREA arguments with size unequal to the parameter size
- IBM2281: flags invalid arguments to BETWEEN and INLIST
- IBM2282: flags REINIT reference that is not level 1
- IBM2283: flags REINIT reference with invalid storage class
- IBM2284: flags type mismatches in LOCNEWVALUE
- IBM2285: flags invalid arguments to PLISTCK

- IBM2286: flags invalid arguments to PLISTCKE
- IBM2292: flags attempts to FETCH oneself
- IBM2293: flags use of JSON functions under CMPAT(V1)
- IBM2294: flags use of long strings under CMPAT(V1) or CMPAT(V2)
- IBM2295: flags use of long strings under BIFPREC(15)
- IBM2442: flags structures with padding
- IBM2642: flags use of OPTIONS(REENTRANT)
- IBM2649: flags duplicate binary arguments in INLIST
- IBM2650: flags duplicate ordinal arguments in INLIST
- IBM2651: flags violations of the MAXBRANCH option
- IBM2652: flags REINIT when there is no INIT
- IBM2653: flags preprocessor options that are not enclosed in quotation marks

### New and changed preprocessor messages

New and changed messages introduced with V4R5 are listed as follows. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM3309: flags compares of SYSPOINTERSIZE to values other than 4 or 8
- IBM3334: flags DEPRECATENEXT(ENTRY) in the MACRO preprocessor
- IBM3660: flags DEPRECATE(ENTRY) in the MACRO preprocessor
- IBM3896: flags VALUE reference not reducible to a character literal
- IBM3897: flags VALUE reference not reducible to a numeric literal
- IBM3898: flags VALUE reference with unsupported type
- IBM3899: flags ambiguous reference
- IBM3900: flags unknown dot-qualified reference
- IBM3901: flags use of unsupported built-in function
- IBM3902: flags built-in argument that is not a structure
- IBM3903: flags numeric VALUE reference that is not REAL FIXED
- IBM3993: flags assertions that fail in a preprocessor

## Messages that are introduced with V4R4

This topic includes new or changed messages that are introduced with V4R4, which includes both compiler messages and preprocessor messages.

### New and changed compiler messages

New and changed messages introduced with V4R4 are listed as follows. Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM2261: flags the use of the overpunch and currency symbols in WIDEPIC
- IBM2262: flags the use of the A and X symbols in WIDEPIC
- IBM2263: flags the use of COMPLEX WIDEPIC as a REFER object
- IBM2264: flags invalid attributes in the LOCATES descriptor
- IBM2265: flags non-constant extents in the LOCATES descriptor
- IBM2266: flags built-in functions when the sole argument does not have the LOCATES attribute

- IBM2267: flags built-in functions when the first argument does not have the LOCATES attribute
- IBM2268: flags pseudovariables when the argument does not have the LOCATES attribute
- IBM2269: flags the LOCATES attribute when it is used on anything but OFFSET
- IBM2270: flags the LOCATES attribute that is used with more than one description
- IBM2271: flags built-in functions when the first argument is not a scalar
- IBM2272: flags built-in functions when the second argument is not a scalar
- IBM2273: flags built-in functions when the OFFSET argument is not qualified
- IBM2274: flags built-in functions when the second argument does not have the LOCATES attribute
- IBM2275: flags built-in function when the third argument is not an AREA
- IBM2276: flags built-in function when the argument must contain the LOCATES attribute
- IBM2277: flags the use of %INCLUDE under NOINCLUDE
- IBM2648: flags the non-STATIC declarations with too many INITIAL items

### New and changed preprocessor messages

New and changed messages introduced with V4R4 are listed as follows. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM3775: flags the use of DFT(ASCII) with PP(CICS) or PP(SQL)
- IBM3878: flags problems during SQL initialization
- IBM3967: flags the use of the CALL statement outside of a procedure
- IBM3968: flags the undefined CALL references
- IBM3969: flags a CALL reference that is not an entry
- IBM3970: flags a CALL reference that is a function
- IBM3971: flags a CALL reference that has the STATEMENT option
- IBM3985: flags semicolons that are used between left and right parentheses in a statement
- IBM3986: flags IF statements that have invalid syntax
- IBM3987: flags statements that have invalid start

## Messages that are introduced with V4R3

This topic includes new or changed messages that are introduced with V4R3, which includes both compiler messages and preprocessor messages.

### New and changed compiler messages

New and changed messages introduced with V4R3 are listed as follows. Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM2643: flags violations of the DEPRECATENEXT(BUILTIN) option
- IBM2644: flags violations of the DEPRECATENEXT(INCLUDE) option
- IBM2645: flags violations of the DEPRECATENEXT(ENTRY) option
- IBM2646: flags violations of the DEPRECATENEXT(VARIABLE) option

- IBM2647: flags violations of the DEPRECATENEXT(STMT) option
- IBM2450: flags bad arguments to Y4DATE, Y4JULIAN, and Y4YEAR
- IBM2451: flags assignments of the form x = y = z under the RULES(NOLAXIF) option
- IBM2452: flags the use of the ROUND built-in function with a negative scale factor under the RULES(NOLAXSCALE) option
- IBM2453: flags violations of the RULES(NOLAXNESTED) option
- IBM2454: flags violations of the DEPRECATE(STMT) option
- IBM2455: flags violations of the CASERULES(KEYWORD) option
- IBM2456: flags violations of the RULES(NORECURSIVE) option
- IBM2457: flags the conflict between the DFT(RECURSIVE) and RULES(NORECUSIVE) options
- IBM2458: flags the violation of the RULES(NOCONTROLLED) option
- IBM2000: flags failed assertion by using the ASSERT statements within the compiler
- IBM2237: flags the third argument to ALLCOMPARE if the argument is not a CHAR(2) constant
- IBM2238: flags the third argument to ALLCOMPARE if the argument has invalid values
- IBM2239: flags the invalid use of unspecified STRUCT types
- IBM2240: flags arithmetic operations on handles for unspecified structures
- IBM2241: flags the use of the SIZE and NEW type functions with unspecified structures
- IBM2242: flags the subtraction of handles to different types
- IBM2243-IBM2254: flag mismatches of the attributes derived from the PROCEDURE statement for the ENTRY constant with those in its explicit declaration.
- IBM2255: flags invalid argument types to UTF8
- IBM2256: flags UTF8 results that are too long
- IBM2257: flags the UTF8 source that is invalid UTF-16
- IBM2258: flags the invalid argument type to the UTF8TOCHAR or UTFTOWCHAR built-in function
- IBM2259: flags the UTF8TOCHAR or UTFTOWCHAR source that is invalid UTF-8
- IBM2260: flags the unsupported use of INITIAL expressions in DEFINE STRUCT

## New and changed preprocessor messages

New and changed messages introduced with V4R3 are listed as follows. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM3000: now used to issue informational-level messages from the CICS or DB2 preprocessor back end
- IBM3024: now used to issue informational-level %NOTE messages
- IBM3250: now used to issue warning-level messages from the CICS or DB2 preprocessor back end
- IBM3259: now used to issue warning-level %NOTE messages
- IBM3261: flags the incomplete syntax in the DEPRECATE option
- IBM3262: flags invalid keywords in the DEPRECATE option

- IBM3326: flags violations of the DEPRECATENEXT(INCLUDE) option
- IBM3500: now used to issue error-level messages from the CICS or DB2 preprocessor back end
- IBM3501: now used to issue error-level %NOTE messages
- IBM3659: flags violations of the STMT suboption of DEPRECATE
- IBM3750: now used to issue severe-level messages from the CICS or DB2 preprocessor back end
- IBM3771: now used to issue severe-level %NOTE messages

## Messages that are introduced with V4R2

This topic includes new or changed messages that are introduced with V4R2, which includes both compiler messages and preprocessor messages.

### New and changed compiler messages

New and changed messages introduced with V4R2 are listed as follows. Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM2211: flags the lack of a closing shift code on a line
- IBM2212: flags the INDICATORS built-in function when it is applied to an element that is not a structure
- IBM2213: flags procedures and BEGIN blocks with too many label arrays
- IBM2214: flags the use of XMLATTR on parent structures
- IBM2215: flags the use of XMLATTR on unnamed elements
- IBM2216: flags the use of XMLATTR on arrays
- IBM2217: flags the use of XMLATTR on an element when the previous element at that logical level does not also have the XMLATTR attribute
- IBM2218: flags the use of XMLOMIT on non-native float elements
- IBM2219: flags the use of INONLY with ASSIGNABLE
- IBM2220: flags the use of OUTONLY with only NONASSIGNABLE
- IBM2221 - IBM2228: flag invalid non-constant extents in BASED
- IBM2230: flags invalid POPCNT arguments
- IBM2231: flags the use of XMLCHAR with a non-native character set
- IBM2232: flags multiple targets in BY DIMACROSS assignments
- IBM2233: flags non-structure targets in BY DIMACROSS assignments
- IBM2234: flags the use of arrays in BY DIMACROSS assignments
- IBM2235: flags invalid targets in BY DIMACROSS assignments
- IBM2419: flags the use of an option with an ARCH level that is too low
- IBM2449: flags violations of RULES(NOSELFASSIGN)
- IBM2820: flags options supported only on other platforms

### New and changed preprocessor messages

New and changed messages introduced with V4R2 are listed as follows. For a fuller and more comprehensive explanation, see *Enterprise PL/I for z/OS Messages and Codes*.

- IBM3024: transmits DB2 I-level messages

- IBM3259: transmits DB2 W-level messages
- IBM3314: flags host variables that must be fully qualified
- IBM3315: flags host variables that are arrays of structures
- IBM3316: flags host variables that are structures of arrays
- IBM3501: transmits DB2 E-level messages
- IBM3502: flags K constants that have too many digits
- IBM3503: flags K constants whose values are too large
- IBM3504: flags M constants that have too many digits
- IBM3505: flags M constants whose values are too large
- IBM3506: flags G constants that have too many digits
- IBM3507: flags G constants whose values are too large
- IBM3508: flags variables with a precision of zero in the DECLARE statement
- IBM3509: flags a DECLARE statement with invalid syntax
- IBM3515: flags scale factors that are greater than 127
- IBM3516: flags scale factors that are less than -128
- IBM3520: flags structure level values equal to 0
- IBM3521: flags structure levels that are too large
- IBM3528: flags a DECLARE statement with more than one precision value
- IBM3529: flags scale factors in a float declaration
- IBM3571: flags the inconsistent use of SQL and PL/I float options
- IBM3572: flags structure declarations in DECLARE statements with an initial level value greater than 1
- IBM3573: flags structure declarations with missing level values
- IBM3574: flags declarations of nameless scalars
- IBM3575: flags duplicate specifications of attributes
- IBM3576: flags empty EXEC SQL statements
- IBM3577: flags INCONLY when it is preceded by other options
- IBM3640: flags declarations with invalid level values
- IBM3641: flags declarations with an invalid LIKE attribute
- IBM3751: flags a missing reference after a colon in an EXEC SQL statement
- IBM3752: flags too many dots in a reference in an EXEC SQL statement
- IBM3753: flags an overly large length value in SQL TYPE IS
- IBM3754: flags a missing left parenthesis in SQL TYPE IS
- IBM3755: flags a missing integer in SQL TYPE IS
- IBM3756: flags a missing right parenthesis in SQL TYPE IS
- IBM3757: flags a missing left parenthesis in SQL TYPE IS XML AS
- IBM3758: flags a missing integer in SQL TYPE IS XML AS
- IBM3759: flags a missing right parenthesis in SQL TYPE IS XML AS
- IBM3766: flags declarations of structures with too many levels
- IBM3767: flags a length value of 0 in SQL TYPE IS
- IBM3771: transmits DB2 S-level messages
- IBM3782: flags a missing AS after SQL TYPE IS XML
- IBM3783: flags a missing type after SQL TYPE IS XML AS
- IBM3784: flags a missing LIKE after SQL TYPE IS TABLE
- IBM3785: flags a missing table-name after SQL TYPE IS TABLE LIKE

- IBM3786: flags a missing AS in SQL TYPE IS TABLE
- IBM3787: flags a missing LOCATOR in SQL TYPE IS TABLE
- IBM3788: flags an invalid type after SQL TYPE IS
- IBM3795: flags the lack of a closing shift code on a line
- IBM3799: flags host variables that are not declared in the SQL DECLARE SECTION
- IBM3805: flags a missing LARGE in SQL TYPE IS XML
- IBM3806: flags a missing OBJECT in SQL TYPE IS XML
- IBM3807: flags a missing LARGE in SQL TYPE IS CHARACTER
- IBM3808: flags a missing LARGE in SQL TYPE IS BINARY
- IBM3809: flags a missing OBJECT in SQL TYPE IS BINARY LARGE
- IBM3877: reports an internal error during an SQL back-end initialization
- IBM3880: flags undefined host variable references
- IBM3881: flags ambiguous host variable references
- IBM3882: flags an indicator array with more than one dimension
- IBM3883: flags an indicator array with nonconstant bounds
- IBM3884: flags an indicator reference that is not an array
- IBM3885: flags a host variable reference with more than one dimension
- IBM3886: flags a host variable array with nonconstant bounds
- IBM3887: flags a host variable array that is not CONNECTED
- IBM3888: flags a host variable reference with no corresponding DB2 type
- IBM3889: flags a host variable reference that is a union
- IBM3890: flags a host variable reference that is an array of structures
- IBM3891: flags a host variables reference that contains an array
- IBM3892: flags a host variable reference that contains structures
- IBM3893: flags a host variable reference that contains unnamed subelements
- IBM3894: flags an indicator reference that is not FIXED BIN(15)
- IBM3895: flags an indicator reference that is a scalar used with an array
- IBM3929: flags EXEC SQL statements not enclosed in a procedure
- IBM3934: flags invalid syntax in EXEC SQL INCLUDE
- IBM3935: flags failure to fetch the SQL back end
- IBM3936: flags an SQL back end that is not at the latest level
- IBM3937: flags EXEC SQL statements that are too long
- IBM3938: flags EXEC SQL statements with too many host variables

## Compiler messages that are introduced with V4R1

This topic includes new and changed messages introduced with V4R1.

Many of these messages are produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

**New messages**

- IBM2210: flags invalid use of the VALUE type function
- IBM2442: flags violations of the RULES(NOPADDING) option
- IBM2443: flags violations of the RULES(NOGLOBALDO) option
- IBM2444: flags violations of the DEPRECATE(BUILTIN) option

- IBM2445: flags violations of the DEPRECATE(INCLUDE) option
- IBM2446: flags violations of the DEPRECATE(ENTRY) option
- IBM2447: flags violations of the DEPRECATE(VARIABLE) option
- IBM2640: flags assignments to the REFER object
- IBM2641: flags syntax errors in options
- IBM2642: flags the PROC(REENTRANT) statement when code might not be reentrant
- IBM3658: flags violations of the DEPRECATE(INCLUDE) option

**Changed messages**

- IBM1204: does not flag the use of the static label arrays, when they are declared as NONASGN
- IBM2016: flags only the use of the VALUE type function in the DEFINE STRUCTURE statement

## Compiler messages that are introduced with V3R9

This topic includes new messages introduced with V3R9.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM1985: includes C runtime message when a file open fails
- IBM1986: reports system (or user) abends occurring during compiles
- IBM2200: traps and flags DFP conversion errors when DFP hardware is absent
- IBM2201: flags invalid arguments to the ROUNDDEC/ROUNDAWAYFROMZERO built-in function
- IBM2202: flags use of MEMCU built-in functions without ARCH(7)
- IBM2203: flags invalid use of VALUE in structures
- IBM2204: flags invalid use of VALUE in structures
- IBM2205: flags invalid use of VALUE in structures
- IBM2206: flags invalid use of VALUE in structures
- IBM2207: flags invalid use of VALUE in structures
- IBM2208: flags invalid use of VALUE in structures
- IBM2209: flags BASED with variable extents
- IBM2435: flags FIXED(p,q) declares with q less than 0
- IBM2436: flags FIXED(p,q) declares with q greater than p
- IBM2437: flags duplicate invocation of PP(SQL)
- IBM2438: flags violations of RULES(NOSTOP)
- IBM2439: flags violations of RULES(NOPROCENDONLY)
- IBM2440: flags violations of RULES(NOLAXQUAL(STRICT))
- IBM2441: flags violations of RULES(NOGOTO(LOOSE))
- IBM2635: flags operations producing FIXED(p,q) with q greater than p
- IBM2636: flags duplicate ORDINALs in SELECT statements
- IBM2637: flags ENTRYs declared without RETURNS and used as functions
- IBM2638: flags lines where the MAXGEN limit is exceeded
- IBM2639: flags statements where the MAXGEN limit is exceeded
- IBM2815: flags inappropriate use of BYVALUE
- IBM2816: flags inappropriate use of BYVALUE

- IBM2817: flags inappropriate use of BYVALUE
- IBM2818: flags FIXED DEC add operations that may raise FOFL
- IBM2819: flags FIXED DEC multiply operations that may raise FOFL
- IBM3518: flags violations of the NAMEPREFIX option
- IBM3810: flags too many labels on one statement during CICS preprocessing

## Compiler messages that are introduced with V3R8

This topic includes new messages introduced with V3R8.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM2189: flags arrays with bounds greater than 2G-1
- IBM2190: flags arrays with bounds less than -2G
- IBM2191: flags OR, NOT or QUOTE with no valid characters
- IBM2192: flags invalid PLISAXC event structures
- IBM2193: flags invalid PLISAXC event structures
- IBM2194: flags invalid PLISAXC event structures
- IBM2195: flags invalid PLISAXC event structures
- IBM2196: flags invalid PLISAXC event structures
- IBM2197: flags invalid arguments to some UTF functions
- IBM2198: flags invalid arguments to some UTF functions
- IBM2199: flags code generation without XPLINK(ON)
- IBM2429: flags CMPAT(V3) without LIMITS(FIXEDBIN(*,63))
- IBM2430: flags mismatches between LINESIZE and RECSIZE
- IBM2431: flags options invalid with GOFF
- IBM2432: flags INITIAL with PARAMETER
- IBM2433: flags INITIAL with DEFINED
- IBM2434: flags unprototyped ENTRYs under RULES(NOLAXENTRY)
- IBM2630: flags operations producing FIXED(p,q) with q larger than p
- IBM2631: flags built-in functions mixing FIXED DEC and FLOAT BIN
- IBM2632: flags built-in functions mixing FIXED DEC and FLOAT DEC
- IBM2633: flags POINTER or OFFSET variables based on FIXED BIN variables
- IBM2634: flags FIXED BIN variables based on POINTER or OFFSET variables
- IBM2814: flags allocations where an aggregate is mapped via a library call

## Compiler messages that are introduced with V3R7

This topic includes new messages introduced with V3R7.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM2184: flags source files with too many lines
- IBM2185: flags invalid arguments to ISFINITE etc
- IBM2186: flags DFP arguments to SQRTF etc
- IBM2187: flags DFP literals with too large exponents
- IBM2188: flags DFP literals with too small exponents
- IBM2420: flags FLOAT(DFP) without ARCH(7)

- IBM2421: flags CLOSE of a file in its ENDFILE block
- IBM2422: flags use of HEX attribute with FLOAT DEC under FLOAT(DFP)
- IBM2423: flags use of IEEE attribute with FLOAT DEC under FLOAT(DFP)
- IBM2424: flags scale factors in FLOAT declarations
- IBM2425: flags ELSE-IF statements when RULES(NOELSEIF) applies
- IBM2426: flags excessive nesting of DO statements
- IBM2427: flags excessive nesting of IF statements
- IBM2428: flags excessive nesting of BEGIN/PROC statements
- IBM2621: flags ON ERROR blocks not starting with ON ERROR SYSTEM
- IBM2622: flags use of function to set the initial value in a DO loop
- IBM2623: flags mixing of FLOAT DEC and FIXED BIN under DFP
- IBM2624: flags mixing of FLOAT DEC and BIT under DFP
- IBM2625: flags mixing of FLOAT DEC and FLOAT BIN under DFP
- IBM2626: flags SUBSTR where third argument is 0
- IBM2627: flags REFER structures not supported by XINFO(XMI)
- IBM2628: flags BYVALUE parameters larger than 32 bytes
- IBM2629: flags variables for which no symbol table information is generated
- IBM2812: flags use of AUTO (and STATIC) variables as tables in TRANSLATE and VERIFY
- IBM3325: flags %DECLARE without any data attributes
- IBM3820: flags invalid use under INCONLY suboption of PP(MACR0) of INCLUDE or XINCLUDE as a macro procedure name
- IBM3821: flags invalid use under INCONLY suboption of PP(MACR0) of INCLUDE or XINCLUDE as a macro statement label
- IBM3822: flags invalid use under INCONLY suboption of PP(MACR0) of INCLUDE or XINCLUDE as a macro variable name

## Compiler messages that are introduced with V3R6

This topic includes new messages introduced with V3R6.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM2180: flags use of KEYED DIRECT files without a KEY/KEYFROM clause
- IBM2181: flags invalid first argument to PICSPEC
- IBM2182: flags invalid second argument to PICSPEC
- IBM2183: flags mismatching arguments in PICSPEC
- IBM2619: flags unreferenced INCLUDE files
- IBM2620: flags structure assignments that would alter REFER objects
- IBM2811: flags use of PICTURE as a loop control variable

## Compiler messages that are introduced with V3R5

This topic includes new messages introduced with V3R5.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM2177: flags using a PDS member as the SYSADATA data set

- IBM2178: flags %INCLUDE statements when the LINEDIR option is in effect
- IBM2179: flags %LINE directives that are too large for the margins
- IBM2416: flags using the LINEDIR option with the TEST(SEPARATE) option
- IBM2417: flags ALLOCATE and FREE of non-PARAMETER CONTROLLED in FETCHABLE using PRV
- IBM2418: flags unreferenced variables
- IBM2615: flags one-time DO loops
- IBM2616: flags use of SIZE against CHAR(*) NONVARYING parameter in an OPTIONS(NODESCRIPTOR) procedure
- IBM2617: flags passing a label as a parameter to a non-PL/I routine
- IBM2618: flags invalid suboptions of compiler suboptions
- IBM2805: flags conversions done by library call when the target can be named
- IBM2806: flags passing a label as a parameter
- IBM2809: flags implicit FIXED DEC to 8-byte integer conversions
- IBM2810: flags difference in conversion of scaled FIXED BIN to FIXED DEC

## Compiler messages that are introduced with V3R4

This topic includes new messages introduced with V3R4.

Many of these messages will be produced only when certain compiler options are in effect. For a fuller and more comprehensive explanation, see *Messages and Codes*.

- IBM2165: flags use of NOWRITABLE(PRV) with LIMITS(EXTNAME(n)) if n is bigger than 7
- IBM2166: flags use of NOWRITABLE(PRV) with RENT
- IBM2167: flags use of NOWRITABLE(PRV) with CMPAT(LE)
- IBM2170: flags too many instances of INTERNAL CONTROLLED
- IBM2171: flags any FETCHABLE ENTRY declared at the PACKAGE level if the NOWRITABLE option is in effect
- IBM2172: flags any FILE CONSTANT declared at the PACKAGE level if the NOWRITABLE option is in effect
- IBM2173: flags any CONTROLLED VARIABLE declared at the PACKAGE level if the NOWRITABLE option is in effect
- IBM2174: flags REPLACEBY2 built-in function references where the result would be longer than CHAR(32767)
- IBM2175: flags REPLACEBY2 built-in function references where the second and third arguments are not restricted expressions
- IBM2176: flags HEX and HEXIMAGE built-in function references where the result would require more than 32767 characters
- IBM2402: flags the declaration of one variable as based on the address of a second variable when the second variable is not large enough to contain the first variable
- IBM2403: flags the use of *PROCESS statements
- IBM2404: flags the declaration of one variable as based on the address of a second variable when the structure containing the second variable is not large enough to contain the first variable
- IBM2405: flags declares and built-in functions that specify an even FIXED DEC precision

- IBM2406: flags arithmetic precision specified in a DEFAULT statement but outside of a VALUE clause
- IBM2407: flags string length specified in a DEFAULT statement but outside of a VALUE clause
- IBM2408: flags AREA size specified in a DEFAULT statement but outside of a VALUE clause
- IBM2409: flags RETURN; statements in functions
- IBM2410: flags the lack of any RETURN statements inside a function
- IBM2411: flags STRING of GRAPHIC aggregates that contain VARYING strings or NONCONNTECT array slices
- IBM2412: flags RETURN statements that specify an expression if the containing PROCEDURE statement does not specify the RETURNS option
- IBM2413: flags use of CONNECTED apart from on parameters and in descriptor lists
- IBM2604: flags FIXED DEC assignments that could raise SIZE
- IBM2605: flags invalid carriage control characters
- IBM2607: flags PIC to FIXED DEC assignments that could raise SIZE
- IBM2608: flags PIC to PIC assignments that could raise SIZE
- IBM2609: flags semicolons in comments
- IBM2610: flags MULTIPLY, DIVIDE, ADD and SUBTRACT built-in function references where one operand is FIXED DEC and the other is FIXED BIN or FLOAT
- IBM2611: flags duplicate binary or bit WHEN values and identifies the duplicate value
- IBM2612: flags duplicate character WHEN values and identifies the duplicate value
- IBM2613: flags possibly uninitialized scalars used as ASGN BYADDR arguments
- IBM2614: flags expressions where the results of two compares are compared
- IBM2801: flags any arithmetic operation where one operand is FIXED BIN with zero scale factor and the other is FIXED DEC with non-zero scale factor, thus producing a FIXED BIN result with nonzero scale factor
- IBM2802: flags aggregate mapping done by library call
- IBM2803: flags statements where GET/PUT STRING EDIT has been optimized
- IBM2804: flags suboptimal compares
- IBM3270: flags EXEC CICS statements when the CICS option is not in effect
- IBM3271: flags EXEC CSPM statements when the CSPM option is not in effect
- IBM3272: flags EXEC DLI statements when the DLI option is not in effect

## Object compatibility

If you want to maintain object compatibility with code generated by VisualAge PL/I or earlier Enterprise PL/I releases, it is imperative that you use, with this Enterprise PL/I release, the same value from each of the following set of options that you used with the earlier compiler.

- BACKREG(5) or BACKREG(11)
- BIFPREC(15) or BIFPREC(31)
- CMPAT(V2) or CMPAT(V1) or CMPAT(LE)
- CSECT or NOCSECT
- LIMITS(EXTNAME(n))

- NORENT or RENT
- WRITABLE or NOWRITABLE

The PTF for APAR PQ66252 changed VisualAge PL/I 2.2.1 (and corresponding PTFs changed Enterprise PL/I 3.1 and 3.2) so that the results of conversions of FLOAT to FIXED DEC and PICTURE would match those produced by the old compilers.

This can cause a small difference in some conversions. For example, consider the following code:

```
dcl f          float dec(16);
dcl d2         dec(15,2);

f = 1.4417e+04;
f = f / 100;
d2 = f;
```

All the compilers will now assign the value 144.17 to *d2*, while before this PTF, the new compilers would have assigned the value 144.16 to *d2*.

With APAR PK17575 (which applied to V3R3, V3R4, and V3R5), the compiler generated code will set a flag in the CAA if MAIN contains an ON FINISH block. With a corresponding library APAR, the library will check for this flag, and unless it is on, it will not raise FINISH. This pair of changes can yield significant performance improvements. However, this also means that after you apply this library APAR, you must recompile any old Enterprise PL/I objects that have an ON FINISH block; otherwise, the ON FINISH block will not be entered.

Apart from these changes, there is complete object compatibility between code compiled by the Enterprise PL/I V3R2 compiler and code compiled by either the VisualAge PL/I or the Enterprise PL/I V3R1 compiler as long as you adhere to these limitations:

- You must not mix code compiled with different CMPAT options.
- You can mix RENT and NORENT code subject to the same restrictions as before:
  - Code compiled with RENT cannot be mixed with code compiled with NORENT if they share any EXTERNAL STATIC variables.
  - Code compiled with RENT cannot call an ENTRY VARIABLE set in code compiled with NORENT.
  - Code compiled with RENT cannot call an ENTRY CONSTANT that was FETCHed in code compiled with NORENT.
  - Code compiled with RENT can FETCH a module containing code compiled with NORENT if one of the following conditions is true:
    - All the code in the FETCHed module was compiled with NORENT.
    - The code containing the entry point to the module was compiled with RENT.
  - Code compiled with NORENT code cannot FETCH a module containing any code compiled with RENT.
  - Code compiled with NORENT WRITABLE cannot be mixed with code compiled with NORENT NOWRITABLE if they share any external CONTROLLED variables or any external FILEs.

It is recommended that all code be compiled with the same settings for the RENT/NORENT and WRITABLE/NOWRITABLE options.

## Runtime changes

The only change to the run time (apart from bug fixes and performance enhancements) that affects the behavior of your code is that the SIZE condition is no longer promoted to the ERROR condition if unhandled. However, some of the compiler changes made in release V3R5 and later releases require corresponding library changes.

So, if you are using code compiled by one of these releases, you must have the PTFs for these APARs installed:

- PQ97843 - for support of TEST(NOHOOK)
- PQ98938 - for support of less code for REFER
- PK03093 - for support of DebugTool starting after MAIN
- PK04110 - for support of PLITABS
- PK11161 - for support of alternate packed decimal signs in FIXED DEC(31) operations
- PK12504 - for support of the DB2 date-time patterns
- PK12833 - for support of TEST(SEPARATE)
- PK50199 - for support of Turkish code pages
- PK50714 - for support of ONOFFSET built-in function
- PK50715 - for support of DFP built-in functions PRED, SUCC etc
- PK50717 - for support of DFP conversions via library code
- PK50718 - to report DFP setting in PLIDUMP output
- PK68704 - for support of the PLISAXC and ONLINE built-in functions
- PK68705 - for support of the UTF handling built-in functions
- PK68708 - for support of the UTF handling built-in functions
- PK72146 - for support of the DFP math built-in functions and CMPAT(V3)
- PK36059 - for support of LRECL=X
- PK74015 - for support of dynamic file allocation
- PK86153 - for support of dynamic file allocation
- PK90903 - for support of opening of SYSPRINT on the IPT
- PM11241 - for support of the new PLISAXD built-in function
- PM13775 - for enhanced support of AUTOMON in DebugTool
- PM19445 - for support of GONUMBER(SEPARATE)
- PM19344 - for support of the new ONAREA built-in function
- PM18574 - for tolerance of DB2 coprocessor XREF option in DB2 V8 and V9 systems

# Part 5. Subsystem and other language considerations

© Copyright IBM Corp. 1999, 2017

# Chapter 20. Assembler considerations for PL/I applications

This chapter contains information for applications that contain mixed PL/I programs and assembler programs.

It includes the following information:

- Considerations for assembler programs mimicking PL/I main procedures
- Calling PL/I from the assembler and the Language Environment conforming assembler
- Condition handling and assembler programs
- Considerations for using assembler user exits

The new compiler uses some different internal control blocks in its generated code than did the old compiler. If you had assembler code that knew the layout and meaning of such control blocks, that code is highly likely not to work now and will probably have to be changed. Some examples where these differences will require code changes:

- Assembler code that "knows" the layout of a PL/I label variable and uses that to try to branch back from assembler into PL/I code
- Assembler code that "knows" the layout of a PL/I file variable and associated file control block and uses that to try to get the DCB for a file

## Considerations for assembler programs mimicking PL/I main procedures

If you have an assembler program mimicking a PL/I MAIN procedure, you must convert that assembler program to an Language Environment-conforming assembler program that is MAIN.

An assembler program that is not Language Environment-conforming cannot call a non-MAIN PL/I procedure (unless it was called from a PL/I MAIN procedure).

For more information, see the *z/OS Language Environment Programming Guide*.

## Calling PL/I from assembler and Language Environment conforming assembler

With Language Environment, assembler programs that call a PL/I routine must follow the calling conventions defined by Language Environment.

For example, Register 13 pointing to a save area, save areas properly back-chained, and the first word of the save area being zero. For detailed information, see the *z/OS Language Environment Programming Guide*.

If your PL/I main program is called by an assembler program and you want to convert your assembler program to use the Language Environment-conforming assembler, you must do either of the following:

- Recompile your PL/I program with a newer PL/I compiler without OPTIONS(MAIN).

© Copyright IBM Corp. 1999, 2017

- Ensure that the entry point receiving control is the real entry point of the PL/I program.

In either case, the called PL/I program is treated as a subroutine. Either of these programs run under the same Language Environment enclave where the assembler program is the main program and the called PL/I program is a subroutine.

There are three ways by which the Language Environment-conforming assembler can pass control to an Enterprise PL/I subroutine:

1. Branch to a statically linked PL/I subroutine.
2. Use the Language Environment macro CEEFETCH to branch to a separately linked Enterprise PL/I subroutine.
3. Use assembler instructions such as LOAD and BALR to a separately linked Enterprise PL/I subroutine.

If you recompile PL/I subroutines that use method 1 or 2 with Enterprise PL/I, you do not need to include CEESG011 with your assembler program. If your assembler program uses instructions as described in method 3, you must always include CEESG011 with your assembler program, even if you recompile your PL/I subroutine with Enterprise PL/I.

## Condition handling and assembler programs

The condition-handling behavior of the LINK from the assembler is now clearly defined.

For detailed information, see the *z/OS Language Environment Programming Guide*.

## Considerations for using assembler user exits

The only Assembler user exit supported by Enterprise PL/I is the Language Environment user exit CEEBXITA.

IBMBXITA and IBMFXITA are not supported.

**Specific considerations:**

- The PL1DUMP, PLIDUMP, or CEEDUMP file for the dump output is treated as a process resource and must not be cleared during enclave termination.
- The OS PL/I abend exit IBMBEER is ignored under Language Environment. See "Differences in condition handling" on page 37 for information about how to force an abend under Language Environment.

For a detailed parameter description for CEEBXITA and more information about assembler language user exits, see the *OS/390 Language Environment Programming Guide*.

# Chapter 21. CICS considerations for PL/I applications

This chapter explains the source language considerations for programs that run under CICS.

It describes the actions that you need to take for applications that use either CICS source or Enterprise PL/I source and involve the following functions:

- General CICS considerations
- Compiler options for programs that run under CICS
- Linking CICS applications and runtime considerations
- Migrating to the integrated CICS preprocessor

## General CICS considerations

The CICS Storage Protect facility was introduced under CICS 3.3. This provides more data integrity and security for the application program and especially for the entire CICS region. Because of the new feature, you might discover that some of your successfully running PL/I applications start to fail with ASRA(0C4) abend and the CICS message DFHSR0622.

If the above problem occurs in your PL/I applications, set the CICS system initialization parameter RENTPGM=NOPROTECT. This sets the protection of the user program in user key. The default for RENTPGM is PROTECT.

If PUT statements are used in your Enterprise PL/I CICS application, especially the `PUT DATA` statement, it might trigger the above error.

Remember also that in CICS programs these PUT statements are intended for debugging purposes only. They have a negative impact on performance, and it is recommended that you do not use them in production programs.

If you mix old and new object code under CICS, you must adhere to all the rules and restrictions described in "Object and load module considerations" on page 135.

### Updating CICS System Definition (CSD) file

When you bring up a CICS region with Language Environment, you must ensure the module names listed in Language Environment CEECCSD are defined in the CSD.

You can locate CEECCSD in SCEESAMP. If you use the CICS Version 4 autoinstall facility, you do not need to define Language Environment modules manually in the CSD.

To run a Enterprise PL/I CICS application, you need to define the Enterprise PL/I member event handler CEEEV011 in the CICS CSD definition table:

```
DEFINE PROGRAM(CEEEV011) GROUP(CEE) LANGUAGE(ASSEMBLER)
DEFINE PROGRAM(IBMPAM24) GROUP(CEE) LANGUAGE(ASSEMBLER)
```

To debug a PL/I transaction with Debug Tool, you need to define the Debug Tool APIs in the CICS CSD definition table:

```
DEFINE PROGRAM(IBMPDAPI) GROUP(CEE) LANGUAGE(ASSEMBLER)
```

© Copyright IBM Corp. 1999, 2017

### Macro-level interface

The CICS macro-level interface is not supported.

## Compiler options for programs that run under CICS

You must use the SYSTEM(CICS) or SYSTEM(MVS) option when you compile your CICS programs that are PL/I MAINs.

If a CICS program is to be reentrant (and most should be) and if it uses CONTROLLED variables or FILEs, you must also compile the program with the NOWRITABLE option.

## Linking CICS applications and runtime considerations

You are generally no longer required to take special action when you link an Enterprise PL/I object module under CICS with the exception that for a routine that is to be FETCHed, you must code the linkage editor ENTRY statement so that it nominates the actual entry point.

PDSEs are supported by CICS Transaction Server 1.3 or later. See the *CICS Transaction Server for OS/390 Release Guide*, GC34-5701, where there are several references to PDSEs, and a list of prerequisite APAR fixes.

### Error-handling

Language Environment prohibits the use of some EXEC CICS commands in any PL/I ON-unit or in any code called from a PL/I ON-unit.

The following EXEC CICS commands are not allowed:

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

All other EXEC CICS commands are allowed within an ON-unit. However, they must be coded by using the NOHANDLE option, the RESP option, or the RESP2 option.

### FETCHing a PL/I MAIN procedure

CICS does not support PL/I FETCHing a PL/I MAIN procedure.

### Runtime output

When a program is compiled with DISPLAY(STD), all runtime output is transmitted to a CICS transient data queue CESE.

When a program is compiled with DISPLAY(WTO), the DISPLAY output is routed to the CICS JESLOG. All other runtime output is transmitted to a CICS transient data queue CESE.

Language Environment ignores the MSGFILE option under CICS. Figure 2 shows format of the output data queue.

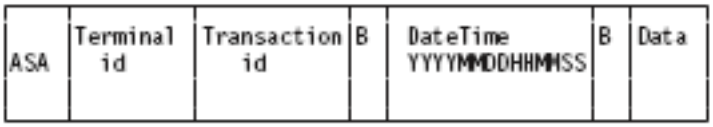



*Figure 2. CESE Output Data Queue*

In addition, the PL/I transient queues CPLI and CPLD are no longer used. As a result, you do not need to specify entries for the CPLI and CPLD in the CICS Destination Control Table (DCT).

## Abend codes used by PL/I under CICS

The APL*x* abend codes that were issued under OS PL/I Version 2 are no longer issued. Instead, Language Environment-defined abend codes are issued.

For more information about Language Environment abend codes, see *z/OS Language Environment Run-Time Messages*.

# Migrating to the integrated CICS preprocessor

This topic highlights some key points that you must take into consideration when you develop and compile CICS programs and when you use the CICS preprocessor.

When you are developing programs for execution under CICS, all the EXEC CICS commands must be translated in one of two ways:

- By the command language translator provided by CICS in a job step prior to the PL/I compilation
- By the PL/I CICS preprocessor as part of the PL/I compilation (this requires CICS TS 2.2 or later)

To use the CICS preprocessor, you must also specify the PP(CICS) compile-time option.

If your CICS program is a MAIN procedure, you must also compile it with the SYSTEM(CICS) option. NOEXECOPS is implied with this option, and all parameters passed to the MAIN procedure must be POINTERs.

If your CICS program includes any files or uses any macros that contain EXEC CICS statements, you must also run the MACRO preprocessor before your code is translated (in either of the ways described above). If you are using the CICS preprocessor, you can specify this with one PP option as illustrated in the following example:

```
pp (macro(...)  cics(...)  )
```

Finally, in order to use the CICS preprocessor, you must have the CICS SDFHLOAD data set as part of the STEPLIB DD for the PL/I compiler.

For more information about the integrated PL/I CICS preprocessor, see the *Enterprise PL/I for z/OS Programming Guide*.

# Chapter 22. IMS considerations for PL/I applications

This chapter explains the considerations for running Enterprise PL/I programs that use IMS under Language Environment.

## Interfaces to IMS

Language Environment supports the PLITDLI, ASMTDLI, and EXEC DLI interfaces from a PL/I routine. It also supports CEETDLI interface from a Enterprise PL/I routine running under IMS/ESA Version 4.

Under Language Environment, CEETDLI is the recommended interface. CEETDLI supports calls that use an Application Interface Block (AIB) or a Program Communication Block (PCB). For more information about AIB and a complete description of the CEETDLI interface, see *IMS/ESA Version 4 Application Programming Guide*.

## SYSTEM(IMS) compile-time option

You should use the SYSTEM(IMS) option when compiling all PL/I MAIN programs invoked from IMS.

When you recompile your main procedure with Enterprise PL/I, the object module assumes that the parameters are passed as BYVALUE. Language Environment converts the parameters to the BYVALUE style for you, if necessary, so the parameters are always passed correctly.

If the BYADDR attribute is specified or implied for the parameters to an IMS MAIN routine, when you compile your main procedure with Enterprise PL/I, you will receive an error message, and the compiler will apply the BYVALUE attribute instead.

For more information about the SYSTEM(IMS) compile-time option, see the *Enterprise PL/I for z/OS Programming Guide*.

## PLICALLA support in IMS

The PL/I PLICALLA entry point is supported under Language Environment.

See "PLICALLA considerations" on page 39 for details.

## PSB language options supported

Language Environment supports PL/I applications with the following PSBGEN LANG options in the supported releases of IMS:

**IMS/ESA Version 4**

Table 12 on page 172 shows support for PSB LANG options in IMS/ESA Version 4 Release 1, and later releases.

© Copyright IBM Corp. 1999, 2017

*Table 12. PSB LANG options for IMS/ESA Version 4 Release 1, and later*

| SYSTEM option | Entry point | LANG= |
|---|---|---|
| IMS | CEESTART | PLI or other values except PASCAL |
| IMS | PLICALLA | PLI |
| MVS | PLICALLA | PLI |
| MVS | CEESTART | PLI |
| Other | - - | Illegal |

## Storage usage considerations

With IMS/ESA Version 3 Release 1 or later, the parameters passed to the IMS interfaces are no longer restricted to the area below the 16M line.

The parameters will be above the 16M line if you observe the following rules:

- If the parameters passed to IMS are in CONTROLLED or BASED storage, specify the ANYWHERE suboption of the HEAP runtime option.
- If the parameters passed to IMS are in AUTOMATIC storage, specify the ANYWHERE suboption of the STACK runtime option.
- If the parameters passed to IMS are in STATIC storage, link the load module with the AMODE(31) attribute.

## Coordinated condition handling under IMS

Language Environment and IMS condition handling are coordinated, which means that if a program interrupt or abend occurs when your application is running in an IMS environment, the Language Environment condition manager is informed whether the problem occurred in your application or in IMS.

If the problem occurs in IMS, Language Environment, as well as any invoked HLL-specific condition handler, percolates the condition back to IMS.

With Language Environment runtime option TRAP(ON), Language Environment continues to support coordinated condition handling for the PLITDLI and ASMTDLI interface invoked from a PL/I routine.

With IMS/ESA Version 3 with PTF UN4928 or with IMS/ESA Version 4, Language Environment also supports the coordinated condition handling for CEETDLI, CTDLI from a C routine, CBLTDLI from a COBOL program, AIBTDLI from a PL/I program, and ASMTDLI from a non-PL/I program.

Note that if a program interrupt or abend occurs in your application outside of IMS, or if a software condition of severity 2 or greater is raised outside of IMS, the Language Environment condition manager takes normal condition handling actions described in the *z/OS Language Environment Programming Guide*. In this case, in order to give IMS a chance to do database rollback, you must do one of the following actions:

- Resolve the error completely so that your application can continue.
- Issue a rollback call to IMS, and then terminate the application.

- Make sure that the application terminates abnormally by using the ABTERMENC(ABEND) runtime option to transform all abnormal terminations into system abends in order to cause IMS rollbacks.
- Make sure that the application terminates abnormally by providing a modified assembler user exit (CEEBXITA) that transforms all abnormal terminations into system abends in order to cause IMS rollbacks.

  The assembler user exit you provide should check the return code and reason code or the CEEAUE_ABTERM bit, and request an abend by setting the CEEAUE_ABND flag to ON, if appropriate. See the *z/OS Language Environment Programming Guide* for details.

## Performance enhancement with Library Retention (LRR)

If you use LRR, you can get an improvement in performance.

See "Improving CPU utilization" on page 131 for details.

# Chapter 23. DB2 considerations for PL/I applications

This chapter explains the source language considerations for programs that run with DB2.

## General DB2 considerations

If you write a user-defined function in PL/I, DB2 passes some string-locator descriptors to the PL/I procedure.

In order for such a program to run correctly under Enterprise PL/I, you must compile the program with the CMPAT(V1) or CMPAT(V2) option.

**Note:** When you use CMPAT(V1) , the size of a BLOB, CLOB, or DBCLOB must be less than 32 K.

## Migrating to the integrated SQL preprocessor

The integrated PL/I SQL preprocessor approach eliminates the need for a separate precompilation step with the DB2 precompiler in PL/I programs containing SQL statements.

**Note:** You must have DB2 for z/OS Version 9 Release 1 or later to use the SQL preprocessor.

### Programming and compilation considerations

When you use the PL/I SQL Preprocessor the PL/I compiler handles your source program containing embedded SQL statements at compile time, without your having to use a separate precompile step. Although the use of a separate precompile step continues to be supported, use of the PL/I SQL Preprocessor is recommended. Interactive debugging with Debug Tool is enhanced when you use the PL/I SQL Preprocessor because you see only the SQL statements while debugging (and not the generated PL/I source).

In addition, using the PL/I SQL Preprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. When you process SQL statements with the PL/I SQL Preprocessor, you can now do the following:

- Use fully-qualified names for structured host variables.
- Include SQL statements at any level of a nested PL/I program, instead of in only the top-level source file.
- Use nested SQL INCLUDE statements.

The PL/I compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the PL/I SQL Preprocessor generates. The listing of the EXEC SQL statement is displayed in a readable format that is similar to the original source.

To use the PL/I SQL Preprocessor, follow these steps:

1. Specify the following option when you compile your program:

```
PP(SQL('options'))
```

© Copyright IBM Corp. 1999, 2017

This compiler option indicates that you want the compiler to invoke the PL/I SQL preprocessor. Specify a list of SQL processing options in the parenthesis after the SQL keyword. The options can be separated by a comma or by a space but **must** be enclosed in single or double quotation marks.

For example, `PP(SQL('DATE(USA),TIME(USA)')` tells the preprocessor to use the USA format for both DATE and TIME data types.

In addition, for LOB support, you must specify the following option:

```
LIMITS( FIXEDBIN(31,63)  FIXEDDEC(15,31) )
```

2. Include DD statements for the following data sets in the JCL for your compile step:
   - DB2 load library (*prefix*.SDSNLOAD)

     The PL/I SQL preprocessor calls DB2 modules to do the SQL statement processing. Therefore, you need to include the name of the DB2 load library data set in the STEPLIB concatenation for the compile step.
   - Library for SQL INCLUDE statements

     If your program contains `SQL INCLUDE` *`member-name`* statements that specify secondary input to the source program, you need to include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compile step.
   - DBRM library

     The compilation of the PL/I program generates a DB2 database request module (DBRM) and the DBRMLIB DD statement is required to designate the data set to which the DBRM is written.

   For example, you might have the following lines in your JCL:

```
//STEPLIB  DD  DSN=DSNA10.SDSNLOAD,DISP=SHR
//SYSLIB   DD  DSN=PAYROLL.MONTHLY.INCLUDE,DISP=SHR
//DBRMLIB  DD  DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
```

For more information about the integrated PL/I SQL preprocessor, see the *Enterprise PL/I for z/OS Programming Guide*.

## FOR BIT DATA assignment notes

The old DB2 Precompiler services did not know about or handle CCSID values for host variables. Because of this lack of knowledge, you could update FOR BIT DATA columns with CHARACTER data. The new DB2 V9.1 or later DB2 Precompiler services does know about CCSID values and assigns them to host variables by using the default CCSID value.

This causes problems if you have code that updates FOR BIT DATA columns with CHARACTER data. The integrated PL/I SQL preprocessor has created a new option, CCSID0 / NOCCSID0 to handle these cases. The CCSID0 option, the default, causes a CCSID of 0 to be assigned to host variables allowing the assignment of CHARACTER variables to FOR BIT DATA database columns.

## Prerequisite DB2 APARs

This topic lists the PTFs for the APAR need to be installed when you migrate to Enterprise PL/I V4R1 and later releases.

- PM18574 - for tolerance of DB2 coprocessor XREF option in DB2 V9 and V8 systems.

# Part 6. Appendixes

© Copyright IBM Corp. 1999, 2017

# Appendix A. Conversion and migration aids

This section describes the conversion and migration tools available for your assistance during the actual conversion and migration activities.

These tools are available:

- OS PL/I Routine Replacement Tool
- OS PL/I V1R5.1 main load module ZAP
- OS PL/I Shared library replacement tool
- OS PL/I object module relinking tool - APARs PN69803
- EDGE Portfolio Analyzer
- Vendor products

## OS PL/I Routine Replacement Tool

Language Environment does not support OS PL/I Version 1 Release 3.0 - 5.0 load modules. For these load modules, you can do one of the following:

- Relink the object modules directly with Language Environment.
- Replace the library routines in the load module with the Language Environment stubs.

Language Environment provides two samples, located in SCEESAMP, that replace the library routines in your OS PL/I Version 1 Release 3.0 - 5.1 and Version 2 load modules with corresponding Language Environment stubs. These samples contain a list of linkage editor REPLACE control statements that replace each library routine in your load module with the corresponding stub in Language Environment and are described as follows:

- IBMWRLK is for MVS non-CICS and VM.

  **For MVS non-CICS**, use it to replace OS PL/I V1R3.0 - V1R5.1 and V2 load modules, both multitasking and nonmultitasking. It contains a CHANGE statement to rename the OS PL/I HLL user exit IBMBINT to CEEBINT.

- IBMWRLKC is for CICS.

  Use it to replace OS PL/I V1R3.0 - V1R5.1 and V2 load modules. It contains a CHANGE statement to rename the OS PL/I HLL user exit IBMBINT to CEEBINT and PLIMAIN to CEEMAIN. It also contains INCLUDE statements to ensure the load module works under CICS.

  The CICS macro language is not supported.

The MVS JCL example below shows the replacement of runtime library routines from a user load module while retaining the user object module. In the example, `MYPDS.LOAD` is the data set name of a load module library that contains the load module with the name `MYLMOD`.

```
//RELINK    EXEC PGM=IEWL,PARM='LIST,MAP,XREF,SIZE(3072K,4K)',REGION=5M
//SYSPRINT  DD   SYSOUT=A
//SYSLIB    DD   DSN=CEE.V1R4M0.SCEELKED,DISP=SHR
//SAMPLIB   DD   DSN=CEE.V1R4M0.SCEESAMP,DISP=SHR
//SYSUT1    DD   UNIT=SYSDA,SPACE=(1024,(200,200))
//SYSLMOD   DD   DSN=MYPDS.LOAD,DISP=OLD
```

© Copyright IBM Corp. 1999, 2017

```
//SYSLIN    DD   *
  INCLUDE   SAMPLIB(IBMWRLK)
  INCLUDE   SYSLMOD(MYLMOD)
  NAME      MYLMOD(R)
```

If you replace a load module under CICS, the CICS SDFHLOAD data set must be specified in the SYSLIB.

## OS PL/I Version 1 Release 5.1 main load module ZAP

Language Environment supports OS PL/I Version 1 Release 5.1 main load module with the following restriction:

- If the main load module is for MVS non-Shared Library, non-CICS and nonmultitasking, or VM, it must first be ZAPped with one of the Language Environment-provided samples located in Language Environment SCEESAMP. Detailed instructions for using the ZAP are provided in IBMRZAPM and IBMRZAPV. The following describes each sample:
  - IBMRZAPM for MVS non-Shared Library, non-CICS, nonmultitasking

    The ZAPped main load module, including one that contains the OS PL/I fast initialization and termination feature, continues to run under OS PL/I Version 1 Release 5.1 and Version 2. When the ZAPped main load module contains the OS PL/I fast initialization and termination feature, it always dynamically loads the OS PL/I runtime initialization routine IBMBPIIA once. IBMBPIIA is not deleted until the task terminates. This one-time loading of IBMBPIIA might affect the performance of your application. If you put IBMBPIIA in LPA, the performance effect can be minimized.

    The ZAPped main load module is supported by Language Environment unless the load module contains the OS PL/I fast initialization and termination feature. Language Environment always dynamically loads the initialization and termination routines. If you put the Language Environment library routines and CEEBLIIA in LPA(E) as recommended in *z/OS Language Environment Installation and Customization under OS/390* and *z/OS Language Environment Customization*, the performance effect can be minimized.
  - IBMRZAPV for VM

    The ZAPped main load module is **not** supported under OS PL/I Version 1 Release 5.1 or Version 2. It is supported only under Language Environment.

If you do not ZAP your main load module, read "OS PL/I Routine Replacement Tool" on page 179 to understand what else you can do. You can also recompile your application with Enterprise PL/I or OS PL/I Version 2. See Chapter 7, "Object module and load module considerations," on page 49 to understand how Language Environment supports OS PL/I object and load modules.

The sample ZAP is available in the IBM Support Center for customers who do not have Language Environment but want to prepare to migrate to Language Environment.

## OS PL/I Shared Library replacement tool

In order to support OS PL/I Version 1 Release 5.1 and Version 2 load modules that use the Shared Library, the library module in that Shared Library must be replaced with Language Environment stubs.

Language Environment provides the following two sample JCL, located in SCEESAMP, to replace the Shared Library:

- IBMRLSLA for OS PL/I Version 1 Release 5.1 MVS CICS or multitasking and OS PL/I Version 2 Shared Library
- IBMRLSLB for OS PL/I Version 1 Release 5.1 MVS non-CICS nonmultitasking Shared Library

You must understand how Language Environment supports OS PL/I Shared Library before you use the JCL.

## OS PL/I Object Module Relinking Tool - APAR PN69803

OS PL/I Version 2 Release 3 provides APAR PN69803 to help you migrate your PL/I-COBOL ILC applications and PLISRTx applications.

### ILC applications

Language Environment does not support the OS PL/I-COBOL ILC applications. You must relink any OS PL/I object module in a PL/I-COBOL ILC application.

See "Differences in Interlanguage Communication support" on page 45 for ILC support under Language Environment.

If you relink your OS PL/I object module in the PL/I-COBOL ILC application with PN69803, however, the resultant load module is supported by Language Environment. PN69803 provides you the flexibility to prepare the PL/I-COBOL ILC relinking while you are using OS PL/I Version 2 Release 3. When you complete the relinking, you can switch to Language Environment whenever you are ready.

Before you relink your PL/I-COBOL ILC applications with PN69803, you must first apply the following PL/I-COBOL ILC APARs to PL/I and COBOL:

OS PL/I V2R3 common library: PN36844
VS COBOL II V1R3.0 library: PN13459
VS COBOL II V1R3.1 library: PN04721
VS COBOL II V1R3.2 library: PN09732

**Note:** VS COBOL II V1R4.0 has the above COBOL APARs in its base code.

If you have not applied the above APARs, PN69803 will not work. The above APARs are not required if your applications do not contain PL/I-COBOL ILC.

Even though your PL/I-COBOL ILC applications are relinked with PN69803, you might still be required to link them with Language Environment if they contain a function described in this book or in *COBOL for OS/390 & VM Migration Guide* that requires relinking. For example, you will still have to relink your application if it contains any COBOL NORES or the load module contains an OS PL/I object module that is not supported by Language Environment. In the latter case, you must recompile your OS PL/I object module with Enterprise PL/I or OS PL/I Version 2.

### PLISRTx applications

While OS PL/I applications that use PLISRTx are supported by Language Environment for OS/390 & VM Release 1.4 and later, it is recommended that you relink your applications that use PLISRTx.

See "Differences in PLISRTx support" on page 42 for the reasons. You can relink either with Language Environment or with PN69803 on OS PL/I Version 2 Release 3. Either method gives your load module the benefits of exploiting the Language Environment DFSORT interface support.

## EDGE Portfolio Analyzer

The Edge Portfolio Analyzer helps you to take an inventory of your existing OS PL/I and PL/I for MVS & VM load modules. The Edge Portfolio Analyzer can perform these tasks:

- Determine which version and release of the OS PL/I compiler or the PL/I for MVS & VM compiler created the load module.
- Determine which compiler options were specified when the load module was compiled.
- Determine which load modules call for the current system date.
- Determine which CSECTs need to be replaced.

**Note:** The Edge Portfolio Analyzer is no longer sold by IBM, but you can still purchase the product from Edge directly. For more information, you can visit their Web site at www.edge-information.com

## Vendor products

A number of non-IBM conversion tools are available to help you upgrade your source programs to Enterprise PL/I programs and move to Language Environment. IBM has compiled a list of vendor products enabled to work with Language Environment and Enterprise PL/I in the *Language Environment Enabled Vendor Tools and Application Packages* document. You can get this information on the web at `http://www.ibm.com/s390/le` and then go to the Library link.

# Appendix B. Compiler elements comparison

Enterprise PL/I has renamed its parts so that, if you want to, you can install it in the same SMP/E zone as OS PL/I or PL/I for MVS & VM.

To help you identify the elements of each product, the following table lists the name differences:

*Table 13. PL/I element names*

| OS PL/I | PL/I for MVS & VM | Enterprise PL/I |
|---|---|---|
| IEL0AA | IEL1AA | IBMZPLI |
| IKJEN00n | IEL1IKJn | |
| IEL0nn | IEL1nn | IBMZnn |
| PLInnnnn | IEL1Mnnn | IBMZMnnn |
| PLIXnnn | IEL1nnn | IBMZnnn |
| PLIHELP | IEL1PLIH | -- |

© Copyright IBM Corp. 1999, 2017

# Appendix C. Compiler limit comparison

This section lists the compiler implementation limits for OS PL/I, PL/I for MVS & VM, VisualAge PL/I, and Enterprise PL/I.

*Table 14. Language element limits*

| Language element | Description | OS PL/I | PL/I for MVS & VM | VisualAge PL/I | Enterprise PL/I |
|---|---|---|---|---|---|
| Arrays | Maximum number of dimensions for an array | 15 | 15 | 15 | 15 |
| | Minimum lower bound | -2147483648 | -2147483648 | -2147483648 | -2147483648 |
| | Maximum upper bound | +2147483647 | +2147483647 | +2147483647 | +2147483647 |
| Structures | Maximum number of levels in a structure | 15 | 15 | 15 | 15 |
| | Maximum level-number in a structure | 255 | 255 | 255 | 255 |
| Arithmetic Precisions | Maximum precision for FIXED DEC | 15 | 15 | 31 | 31 |
| | Maximum precision for FIXED BINARY | 31 | 31 | 63 | 63 |
| | Maximum precision for FLOAT DEC | 33 | 33 | 33 | 33 |
| | Maximum precision for FLOAT BINARY | 109 | 109 | 109 | 109 |
| | Maximum scale factor for FIXED data | 127 | 127 | 127 | 127 |
| | Minimum scale factor for FIXED data | -128 | -128 | -128 | -128 |
| String and AREA Variables or Constants | Maximum length of CHARACTER | 32767 | 32767 | 32767 | 32767 |
| | Maximum length of BIT | 32767 | 32767 | 32767 | 32767 |
| | Maximum length of GRAPHIC | 16383 | 16383 | 16383 | 16383 |
| | Maximum length of WIDECHAR | n/a | n/a | 16383 | 32767 |
| | Maximum size of AREA | 2147483647 | 2147483647 | 2147483647 | 2147483647 |
| Built-In Functions | Maximum number of arguments to the IAND, IOR, MAX, and MIN functions | 64 | 64 | 64 | 64 |

© Copyright IBM Corp. 1999, 2017

*Table 14. Language element limits (continued)*

| Language element | Description | OS PL/I | PL/I for MVS & VM | VisualAge PL/I | Enterprise PL/I |
|---|---|---|---|---|---|
| Program Size | Maximum length of an identifier | 31 | 31 | 100 | 100 |
| | Maximum number of procedures in a program | 255 | 255 | 255 | 255 |
| | Maximum number of DEFAULT statements in a block | 31 | 31 | 31 | 31 |
| | Maximum nesting of %INCLUDE statements | 8 | 8 | 2046 | 2046 |
| | Maximum number of lines in any source file | 65,535 | 65,535 | 1048575 | 1048575 |
| | Maximum number of statements | 32,767 | 32,767 | 16777215 | 16777215 |
| | Maximum number of LIKE-attributes in a block | 63 | 63 | 63 | 63 |
| | Maximum number of output expressions in a data-list | 60 | 60 | 60 | 60 |
| | Maximum number of repetitive DO-specifications in a data-list | 25 | 25 | 50 | 50 |

*Table 14. Language element limits (continued)*

| Language element | Description | OS PL/I | PL/I for MVS & VM | VisualAge PL/I | Enterprise PL/I |
|---|---|---|---|---|---|
| Program Size | Maximum size of a data aggregate containing no unaligned bits | 2147483648 | 2147483648 | 2147483647 | 2147483647 |
| | Maximum size of a data aggregate containing some unaligned bits | 268435455 | 268435455 | 268435455 | 268435455 |
| | Maximum number of arguments in a CALL or function reference | 64 | 64 | 255 | 255 |
| | Maximum number of parameters for a procedure | 64 | 63 | 4095 | 4095 |
| | Maximum nesting of factored attributes | 15 | 15 | 15 | 15 |
| | Maximum nesting of BEGIN and PROCEDURE statements | 42 | 42 | 30 | 30 |
| | Maximum nesting of DO-groups | 38 | 38 | 49 | 49 |
| | Maximum nesting of IF statements | 80 | 80 | 49 | 49 |
| | Maximum nesting of SELECT-statements | 50 | 50 | 49 | 49 |
| | Maximum length of %NOTE message | 256 | 256 | 32767 | 32767 |

*Table 14. Language element limits (continued)*

| Language element | Description | OS PL/I | PL/I for MVS & VM | VisualAge PL/I | Enterprise PL/I |
|---|---|---|---|---|---|
| Miscellaneous | Maximum number of picture characters in a character picture | 511 | 511 | 511 | 511 |
| | Maximum number of bytes in a numeric picture | 256 | 256 | 253 | 253 |
| | Maximum number of numeric picture characters in a numeric picture | 15 | 15 | 31 | 31 |
| | Maximum length for a KEYTO character string | 120 | 120 | 120 | 120 |
| | Maximum length for a KEYTO graphic or widechar string | 60 | 60 | 60 | 60 |
| | Maximum KEY length | 8 | 8 | 32763 | 32763 |
| | Maximum line size for LINESIZE | 32,000 | 32,000 | 32,000 | 32,759 for F-format or U-format, and 32,751 for V-format |
| | Minimum line size for LINESIZE | 10 | 10 | 1 | 1 |
| | Maximum page size for PAGESIZE | 32,000 | 32,000 | 32,767 | 32,767 |
| Miscellaneous | Minimum page size for PAGESIZE | 1 | 1 | 1 | 1 |
| | Maximum size of DISPLAY character string | 126 | 126 | 126 | 126 |
| | Maximum DISPLAY reply message. | 72 | 72 | 72 | 72 |

# Appendix D. Batch processing sample

The code samples show how to implement a 'batch compiler' with Enterprise PL/I.

```
batch: proc options(main);

   dcl eof     bit(1);
   dcl rc      fixed bin(15);
   dcl system  builtin;
   dcl source  char(80);
   dcl sysutz  file output record sequential
               env( fb,recsize(80) );

   dcl compin  file input record sequential;

   dcl plixopt ext static char(40) var
               init('errcount(0),heap(2m,1m,any,free)');

   open file(compin);

   rc = 0;
   eof = '0'b;
   data_read = '0'b;
   on endfile(compin) eof = '1'b;

   data_read = '0'b;
   open file(sysutz);
   read file(compin) into(source);
   do while( eof = '0'b );
     if substr(source,1,8) = '*PROCESS' then
       if data_read then
         do;
           close file(sysutz);

           rc = max( rc, system('ibmzpli @dd:options') );

           data_read = '0'b;
           open file(sysutz);
         end;
       else;
     else
       data_read = '1'b;
     write file(sysutz) from(source);
     read file(compin) into(source);
   end;

   close file(sysutz);

   rc = max( rc, system('ibmzpli @dd:options') );
   call pliretc(rc );
 end;
```

This program when compiled and linked could be used as a "batch compiler" if the following JCL were used when the program is run.

```
//SYSPRINT  DD SYSOUT=*
//OPTIONS   DD  *
 dd(*,sysutz) name
 limits(extname(7)) norent cmpat(v2)
//COMPIN    DD  *
*PROCESS X(F);
  x: proc;
    dcl a ext char(80);
```

© Copyright IBM Corp. 1999, 2017

```
  end;
*PROCESS NORENT;
  y: proc;
    dcl b ext char(40);
  end;
//SYSLIN   DD DSN=...,DISP=(MOD)
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUTZ  DD DSN=&&SOURCE,DISP=(NEW),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
```

The first line in the OPTIONS DD specifies the DD(*,SYSUTZ) and NAME and is necessary to make the program work as a batch compiler. The second line is used merely as an example.

# Appendix E. Debugging tool comparison

Debug Tool is a program analyzer that runs within Language Environment and supports a number of high level languages, including Enterprise PL/I.

For Enterprise PL/I, Debug Tool is orderable as a feature of the compiler.

## Differences between debugging tools

IBM Debug Tool is the interactive debugger that supports PL/I and Language Environment. Debug Tool functions are equivalent to PLITEST functions. Some names of PLITEST commands, however, have changed in Debug Tool and are no longer accepted. These are listed in Table 15.

You must have Language Environment for OS/390 & VM Release 4 or later installed on your system before you can use Debug Tool with your OS PL/I applications.

*Table 15. PLITEST Commands and Their Debug Tool Equivalents*

| PLITEST command | Equivalent Debug Tool command |
|---|---|
| CLEAR ON | CLEAR AT OCCURENCE |
| LIST %FPRS | LIST SHORT FLOATING |
| LIST %LPRS | LIST LONG FLOATING |
| LIST %GPRS | LIST REGISTERS |
| LIST SNAP | LIST CALLS |
| MOVECURS | CURSER |
| ON | AT OCCURENCE |
| QUERY AT | LIST AT |
| QUERY ATTRIBUTES | DESCRIBE ATTRIBUTES |
| QUERY BEARINGS | QUERY LOCATION |
| QUERY ENVIRONMENT | DESCRIBE ENVIRONMENT |
| QUERY MONITOR | LIST MONITOR |
| QUERY NAMES 'pattern' | LIST NAMES 'pattern' |
| QUERY NAMES PROCEDURE | LIST PROCEDURE |
| QUERY PROGRAM | DESCRIBE PROGRAM |
| QUERY STATEMENT NUMBERS | LIST STATEMENT NUMBERS |
| SEARCH | FIND |
| SET GRAPHIC | SET DBCS |
| SET LANGUAGE | SET NATIONAL LANGUAGE |
| SET LAST n | SET HISTORY n |
| SET FILE | SET LOG |
| SIGNAL (ON cond) PROGRAM | TRIGGER (ON cond) |
| SIGNAL (ON cond) TEST | TRIGGER AT OCCURENCE (ON cond) |
| SIGNAL (AT cond) TEST | TRIGGER AT (AT cond) |

© Copyright IBM Corp. 1999, 2017

*Table 15. PLITEST Commands and Their Debug Tool Equivalents (continued)*

| PLITEST command | Equivalent Debug Tool command |
|---|---|
| VTRACE | STEP |
| WINDOWS | LAYOUT |

# Notices

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

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

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

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.

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

IBM Corporation
J74/G4
555 Bailey Avenue
P.O. Box 49023

© Copyright IBM Corp. 1999, 2017

San Jose, CA 95161-9023
U.S.A.

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

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

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

## Programming interface information

This book is intended to help the customer migrate from previous releases of PL/I to Enterprise PL/I and z/OS Language Environment. This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Enterprise PL/I.

## 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” at www.ibm.com/legal/copytrade.shtml.

Windows is a trademark of Microsoft Corporation in the United States and/or other countries.

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

# Bibliography

## PL/I publications

### Enterprise PL/I for z/OS

*Programming Guide*, GI11-9145
*Language Reference*, SC14-7285
*Messages and Codes*, GC14-7286
*Compiler and Run-Time Migration Guide*, GC14-7284

### PL/I for MVS & VM

*Installation and Customization under MVS*, SC26-3119
*Language Reference*, SC26-3114
*Compile-Time Messages and Codes*, SC26-3229
*Diagnosis Guide*, SC26-3149
*Migration Guide*, SC26-3118
*Programming Guide*, SC26-3113
*Reference Summary*, SX26-3821

### PL/I for AIX

*Programming Guide*, SC14-7319
*Language Reference*, SC14-7320
*Messages and Codes*, GC14-7321
*Installation Guide*, GC14-7322

## Related publications

### DB2 UDB for z/OS

*Administration Guide*, SC19-2968

*Application Programming and SQL Guide*, SC19-2969

*Command Reference*, SC19-2972

*Messages*, GC19-2979

*Codes*, GC19-2971

*SQL Reference*, SC19-2983

See also the Information Center: publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2z10.doc/src/alltoc/db2z_10_prodhome.htm

### DFSORT

*Application Programming Guide*, SC33-4035

*Installation and Customization*, SC33-4034

### IMS/ESA

*Application Programming: Database Manager*, SC26-8015

*Application Programming: Database Manager Summary*, SC26-8037

*Application Programming: Design Guide*, SC26-8016

*Application Programming: Transaction Manager*, SC26-8017

*Application Programming: Transaction Manager Summary*, SC26-8038

*Application Programming: EXEC DL/I Commands for CICS and IMS™*, SC26-8018

© Copyright IBM Corp. 1999, 2017

*Application Programming: EXEC DL/I Commands for CICS and IMS Summary*, SC26-8036

### TXSeries for Multiplatforms

*Encina Administration Guide Volume 2: Server Administration*, SC09-4474

*Encina SFS Programming Guide*, SC09-4483

See also the Information Center: publib.boulder.ibm.com/infocenter/txformp/v7r1/index.jsp

### z/Architecture

*Principles of Operation*, SA22-7832

### z/OS Language Environment

*Concepts Guide*, SA22-7567

*Debugging Guide*, GA22-7560

*Run-Time Messages*, SA22-7566

*Customization*, SA22-7564

*Programming Guide*, SA22-7561

*Programming Reference*, SA22-7562

*Run-Time Application Migration Guide*, GA22-7565

*Writing Interlanguage Communication Applications*, SA22-7563

### z/OS MVS

*JCL Reference*, SA22-7597

*JCL User's Guide*, SA22-7598

*System Commands*, SA22-7627

### z/OS TSO/E

*Command Reference*, SA22-7782

*User's Guide*, SA22-7794

### z/OS UNIX System Services

*z/OS UNIX System Services Command Reference*, SA22-7802

*z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803

*z/OS UNIX System Services User's Guide*, SA22-7801

### Unicode and character representation

*z/OS Support for Unicode: Using Conversion Services*, SC33-7050

# Index

## Special characters


(NO)GOSTMT 68
(NO)GOSTMT compiler option 68


## A


abend codes
  CICS considerations 57, 169
ABTERMENC run-time option 127
ABTERMENC runtime option 36
accessibility
  assistive technologies xvi
  of Enterprise PL/I for z/OS xv
ADDBUFF ENVIRONMENT option 64
ALL31 run-time option 36, 127, 131
ALL31 runtime option 36
AMODE(24)
  link-edit considerations 125
  mixing with 31-bit data 26
  support 70
AMODE(31)
  ALL31 considerations 36
APARs
  Prerequisite LE 130
AREA
  with INITIAL 119
array expressions restrictions 65
ASCII ENVIRONMENT option 64
ASMTDLI IMS interface 57, 171
assembler support
  calling PL/I 128
  IMS considerations 58, 171
  invocation of PL/I 46, 165
  invoking the compiler from 69
  main parameter list 47
  need to link CEESG011 128
  PLIMAIN entry point 46
  PLISTART entry point 46
assistive technologies xvi


## B


BACKWARDS file attribute 64
batch compilation
  example 189
  restrictions 68
BUFFERS ENVIRONMENT option 64
BUFOFF ENVIRONMENT option 64
built-in functions
  DATE/TIME 43
  math 130
  over 100 new 8
  restricted 66
  with conversion from graphic to character type 121
  with scaled FIXED BIN 121


## C


CEEBXITA user exit 166
CEESTART
  and PLICALLA 39
  using 46
CEEUOPT
  and PLICALLB 39
  and runtime options 36
CICS considerations
  abend codes used by PL/I 57, 169
  CSD file, updating 55, 167
  discussion of 55, 167
  dropping support for OS PL/I 5
  error handling 56
  integrated pr eprocessor 8
  integrated preprocessor 169
  invoking existing CICS applications 32
  linking Enterprise PL/I applications 168
  macro-level interface 56, 168
  runtime output 56, 168
  STACK runtime option, using 56
  SYSTEM compiler option 168
CMPAT compiler option 74, 135
  DB2 considerations 175
compatibility
  compiler options
    BIFPREC(15) 73
    CMPAT(V*) 73
    DFT(LINKAGE(SYSTEM)) 73, 77
    DFT(NOBIN1ARG) 77
    DFT(OVERLAP) 73, 77
    EXTRN(FULL) 73, 75
    LIMITS(EXTNAME(7)) 73, 75
    NOREDUCE 73, 77
    NORENT 73, 76
    NORESEXP 73, 78
    NOWRITABLE 78
    RULES(LAXCTL) 73, 78
  considerations
    PLICALLA entry point 39
    PLICALLB entry point 39
  run-time options needed
    ABTERMENC(RETCODE) 127
    DEPTHCONDLMT(0) 127
    ERRCOUNT(0) 127
    TRAP(ON) 127
    XUFLOW(ON) 127
compile unit definition 44
compiler advantages
  FETCH 8
  integrated preprocessors 8
  multithreading, support of 8
  new built-in functions 8
compiler messages
  2603 90
  EXIT option 100
  IBM1044 87
  IBM1053 87, 121
  IBM1063 113
compiler messages *(continued)*
  IBM1065 87
  IBM1089 107
  IBM1091 88
  IBM1099 88
  IBM1181 89
  IBM1196 119
  IBM1206 90
  IBM1208 90
  IBM1215 91
  IBM1216 91
  IBM1220 92, 107
  IBM1927 92
  IBM1936 70
  IBM1948 93
  IBM2063 83, 93
  IBM2402 93
  IBM2409 94
  IBM2410 94
  IBM2412 94
  IBM2421 95
  IBM2610 95
  IBM2611 95
  IBM2617 96
  IBM2621 96
  IBM2622 96
  IBM2626 97
  IBM2628 97
  IBM2801 98
  IBM2804 98
  IBM2810 98
  IBM2811 99
  IBM2812 99
  IBM5002 70
compiler options
  BACKREG 73
  BIFPREC 74
  CMPAT 74, 135
  DEFAULT
    LINKAGE 77, 135
    NOBIN1ARG 77
    NONASGN 80
    NONCONNECTED 80
    NOOVERLAP 80
    OVERLAP 77, 135
    REORDER 80, 81
  EXTRN 75, 135
  GONUMBER 86
  LIMITS
    EXTNAME 75, 135
  NOREDUCE 77, 81
  NORENT 76, 82, 135, 161
  NOWRITABLE 76, 78, 161
  OPTIMIZE 80, 81
  PREFIX 86
  REDUCE 81
  RENT 76, 161
  restricted 67
  RULES
    LAXCTL 78
    LAXSTRZ 85


© Copyright IBM Corp. 1999, 2017

compiler options *(continued)*
RULES *(continued)*
NOLAXCTL 82
NOLAXDCL 83
NOLAXIF 83
NOLAXMARGINS 85
NOMULTICLOSE 85
SYSTEM 76
TEST 86
unsupported 68
WRITABLE 76, 161
compiler options restricted
INCLUDE 67
LANGLVL 68
LIST 68
STMT 68
SYSTEM 68
compiler options unsupported 68
CONTROL 68
COUNT 68
DECK 68
ESD 68
FLOW 68
HGPR 68
IMPRECISE 68
LMESSAGE 68
SEQUENCE 68
SIZE 68
SMESSAGE 68
compiler restrictions
array expressions 65
built-in functions 66
DBCS 67
DEFAULT statement 65
extents of automatic variables 66
GRAPHIC and POSITION 67
iSUB defining 66
LABEL arrays 66
MACRO preprocessor 67
multitasking facility 42
OPTIONS(REENTRANT) 66
pseudovariables 66
RECORD I/O 64
STREAM I/O 64
structure expressions 65
VM 9
compiler support dropped
CHARSET(48) 63
CHECK 63
EGCS 63
Fortran 63
invalid code 63
multitasking 63
VM 9
condition handling
differences 37
IBMBXITA and IBMBEER differences 38
IMS considerations 58, 172
severity differences 39
timing differences 37
U4039 differences 38
unhandled condition differences 38
conditions
ERROR 37
FIXEDOVERFLOW 105, 117
OVERFLOW 115
conditions *(continued)*
UNDERFLOW 36
ZERODIVIDE 115
considerations
before migrating 35
assembler 165
condition handling 37
DATE/TIME built-in functions 43
debugging tools 191
ILC differences 45
performance retuning 131
PLIDUMP 44
preinitialized program 41
runtime message 43
runtime options 35
storage report 45
storage use retuning 131
user return code 43
using sort program 42
installation
Enterprise PL/I 23
OS/390 requirements 7
product configuration 183
product configuration, SCEELKED 7
product configuration, SCEERUN 7
link-edit
CHANGE card 127
math routines 54
NCAL linkage editor option 53
PLICALLA and PLICALLB 125
symbol table 53
subsystem
CICS 55, 167
DB2 59, 175
IMS 57, 171
Considerations
Before Migrating
Run-time messages 129
Runtime messages 128, 129
CONTROL compiler option 68
COUNT compiler option 68
COUNT run-time option 127
COUNT runtime option 35
CPU utilization, improving 131
CSD file, updating 55, 167
CSECTs
IDR information 18, 182
symbol table 53

## D

data sets
load module considerations 49
new, OS/390 7
DATE/TIME built-in functions 43
DB2 considerations 59, 175
DBCS restrictions 67
SQL Preprocessor 175
Debug Tool 191
comparing with PLITEST 191
product relationships 7
debugging tools, differences in 191
DECK compiler option 68
DEFAULT compiler option
LINKAGE 77, 135
DEFAULT compiler option *(continued)*
NOBIN1ARG 77
NONASGN 80
NONCONNECTED 80
NOOVERLAP 80
OVERLAP 77, 108, 135
REORDER 80, 81
DEFAULT statement restrictions 65
DEPTHCONDLMT run-time option 127
DEPTHCONDLMT runtime option 36
DFSORT, using 42
dump differences 130

## E

education
Enterprise PL/I 24
Language Environment 16
Enterprise PL/I for z/OS
accessibility xv
Enterprise PL/I for z/OS library xiii
Enterprise PL/I library xiii
ENVIRONMENT options not supported 64
ERRCOUNT run-time option 127
ERRCOUNT runtime option 36
ERROR condition 37
error handling, CICS considerations 56
ESD compiler option 68
EXCLUSIVE file attribute 64
EXEC DLI interface 57, 171
EXEC SQL statements 175
EXTERNAL
uninitialized STATIC 111
EXTRN compiler option 75, 135

## F

FETCH
compiler advantages 8
file attributes not supported
BACKWARDS 64
EXCLUSIVE 64
TRANSIENT 64
FIXED BIN
and FIXEDOVERFLOW 117
with precision <= 7 118
FIXEDOVERFLOW
and SIZE 105
restricted to FIXED DEC 117
FLOAT
assignments to FLOAT 109
FLOW compiler option 68
FLOW run-time option 127
FLOW runtime option 35

## G

GONUMBER compiler option 86
GRAPHIC and POSITION restrictions 67

# H

HEAP run-time option 127, 132
HEAP runtime option 35
HGPR compiler option 68

# I

IBMBEER user exit
  differences 38
  installation considerations 166
IBMBXITA user exit 166
  differences 38
IBMFXITA user exit 166
IBMRLSLx, replacing Shared Library 180
IBMWRLKx, replacing library routine 179
IEEE floating point
  support of 8
ILC (interlanguage communication)
  differences in 45
  enabled languages 45
  migration considerations 18
  PLIXOPT considerations 36
IMPRECISE compiler option 68
IMS considerations
  assembler language options support 58, 171
  condition handling 58, 172
  discussion of 57, 171
  interfaces 57, 171
  interfaces to 57, 171
  PLICALLA support 57, 171
  PSB language options 58, 171
  STEPLIB use and LE 20
  storage usage 58, 172
  SYSTEM compiler option 57, 171
INCLUDE compiler option 67
INDEXAREA ENVIRONMENT option 64
installation
  Language Environment 15
interlanguage communication (ILC)
  differences in 45
  enabled languages 45
introduction
  PL/I runtime environment 8
ISAINC run-time option 127
ISASIZE run-time option 127
ISASIZE runtime option 35
iSUB defining restrictions 66

# L

LABEL array restrictions 66
LANGLVL compiler option 68
Language Environment
  educating programmers 16
  Enterprise PL/I prerequisite level 15
  invoking existing CICS applications 32
  invoking existing non-CICS applications 31
  planning move to 15
  running existing applications 31
Language Environment library xiv
LANGUAGE run-time option 127
LANGUAGE runtime option 35
library routine replacement tool
  using IBMWRLKx 179
LIMITS compiler option
  EXTNAME 75, 135
    link-edit considerations 125
link-edit
  AMODE(24) considerations 125
  and CHANGE 127
  and PLICALLA 125
  and PLICALLB 125
  effects of LIMITS on 125
  effects of RENT/NORENT on 125
  existing applications with LE 32
  math routines, using 54
  NCAL option 53
  symbol table 53
  symbol tables
    CSECT 53
    discussion of 53
  using NCAL option 53
linking applications under CICS 168
LIST compiler option 68
LMESSAGE compiler option 68
LNKLST
  SCEERUN 3, 7
  use in migration 19
load module
  considerations for
    data sets 49
    OS PL/I Version 2 51
  general considerations 49
  identifying PL/I version 18, 182
  IDR information 18, 182
  Language Environment support
    OS PL/I version 1 prior to release 3.0 51
    OS PL/I version 1.3.0 - 1.4.0 51
    OS PL/I version 1.5.0 50
    OS PL/I version 1.5.1 49
    OS PL/I version 2 51
loops
  endless 107
LPALST
  use in migration 19
LRECL
  compiler SYSPRINT 70

# M

macro-level interface, CICS considerations 56, 168
main load module relinking aid
  using sample ZAP 180
main load module, user
  sample ZAP relinking aid 180
math built-ins
  differences 130
math routines, using OS PL/I 54
messages
  2603 90
  EXIT option 100
  IBM1044 87
  IBM1053 87
  IBM1065 87
  IBM1091 88
messages *(continued)*
  IBM1099 88
  IBM1181 89
  IBM1206 90
  IBM1208 90
  IBM1215 91
  IBM1216 91
  IBM1220 92
  IBM1927 92
  IBM1948 93
  IBM2063 83, 93
  IBM2402 93
  IBM2409 94
  IBM2410 94
  IBM2412 94
  IBM2421 95
  IBM2610 95
  IBM2611 95
  IBM2617 96
  IBM2621 96
  IBM2622 96
  IBM2626 97
  IBM2628 97
  IBM2801 98
  IBM2804 98
  IBM2810 98
  IBM2811 99
  IBM2812 99
migrating
  to new compiler 23
migration
  aid for replacing Shared Library 180
  compiler, basics 4
  cut to production 22
  general tasks 10
  ILC considerations 18
  library routine replacement tool 179
  LNKLST use 19
  object module relinking tool 181
  phasing in LE 18
  PL/I application conversion 26
  PL/I application priority 25
  regression testing 21
  relinking aid, using 181
  relinking PLISRTx modules 181
  runtime, basics 4
  sample ZAP for relinking main load module 180
  STEPLIB example 20
  STEPLIB use 19
  taking application inventory 16, 24
  tools and aids 179
MSGFILE run-time option 127, 129
MSGFILE runtime option 43
multitasking facility
  support of 9
multithreading
  support of 8

# N

NATLANG run-time option 127
NATLANG runtime option 35
NCAL linkage editor option 53
NCP ENVIRONMENT option 64
NOREDUCE compiler option 77, 81

NORENT compiler option 76, 82, 135, 161
link-edit considerations 125
notices 193
NOWRITABLE compiler option 76, 78, 161
NOWRITE ENVIRONMENT option 64

## O

object and load module
considerations 49, 51
object module
general considerations 49
ILC migration aid 181
Language Environment support
OS PL/I version 1 prior to release 3.0 51
OS PL/I version 1.3.0 - 1.4.0 51
OS PL/I version 1.5.0 50
OS PL/I version 1.5.1 49
OS PL/I version 2 51
OPTIMIZE compiler option 80, 81
OS PL/I
service 5
Version 2 load modules 51
OVERFLOW condition 115

## P

performance
compiler options
DFT(NONASGN) 80
DFT(NONCONNECTED) 80
DFT(NOOVERLAP) 80
DFT(REORDER) 80, 81
NORENT 82
OPTIMIZE(2) 80, 81
REDUCE 81
RULES(NOLAXCTL) 82
CPU utilization 131
FIXED BIN(15) as a loop control 113
FIXED DEC as a loop control 112
retuning for 131
storage utilization 132
TOTAL environment option 115
under CICS, improving 133
under IMS, improving 133
PLICALLA entry point
IMS considerations 57, 171
support for 39
PLICALLB entry point
support for 39
PLIDUMP
differences 44, 130
output produced by 44
PLIMAIN entry point 46
PLISRTx module relinking tool 181
PLISRTx, using 42
PLISTART
and PLICALLA 39
entry point 46
PLITDLI IMS interface 57, 171
PLITEST
comparing with Debug Tool 191
PLIXHD 71, 130
PLIXOPT
and PLICALLB 39
and runtime options 36
PREFIX compiler option 86
preinitialized program 41
preprocessors
CICS preprocessor 169
SQL preprocessor 175
product configuration
data sets
new 7
OS/390 7
discussion of 183
product relationships
Debug Tool 7
programs, preinitialized 41
PSB language options, IMS considerations 58, 171
pseudovariables restricted 66
PTFs
Prerequisite LE 130

## R

recompile
do I need to 3
RECORD I/O
restrictions 64
REDUCE compiler option 81
REENTRANT procedure option 66
REGIONAL ENVIRONMENT option 64
relinking OS PL/I-COBOL ILC
using the relinking tool 181
relinking user main load module
sample ZAP relinking aid 180
RENT compiler option 76, 161
link-edit considerations 125
replacing library routines
using IBMWRLKx 179
replacing OS PL/I Shared Library
sample replacement aid 180
REPORT run-time option 127
REPORT runtime option 35
retuning applications
CPU utilization 131
storage utilization, improving 132
under IMS, improving 133
return codes 128
RPTSTG run-time option 127
using for tuning storage 16, 131
RPTSTG runtime option 35
RULES compiler option
ANS 89
LAXCTL 78
LAXSTRZ 85
NOLAXCTL 82
NOLAXDCL 83
NOLAXIF 83
NOLAXMARGINS 85
NOMULTICLOSE 85
Run-time messages 129
run-time options
ABTERMENC 127
ALL31 127, 131
COUNT 127
DEPTHCONDLMT 127
ERRCOUNT 127
run-time options *(continued)*
FLOW 127
HEAP 127, 132
MSGFILE 127, 129
NATLANG 127
RPTSTG 127, 131
STACK 127, 132
STORAGE 127
TRAP 127
XUFLOW 127
runtime environment
for PL/I 8
runtime message differences 43
Runtime messages 128, 129
runtime options
ABTERMENC 36
ALL31 36
COUNT 35
DEPTHCONDLMT 36
differences 35
ERRCOUNT 36
FLOW 35
HEAP 35
ISASIZE 35
LANGUAGE 35
MSGFILE 43
NATLANG 35
REPORT 35
RPTSTG 9, 35
SPIE 35
STACK 35
STAE 35
TRAP 35
XUFLOW 36
runtime output, CICS considerations 56, 168

## S

SCEELKED
and non-IBM names 32
configuration 7
SCEERUN
configuration 7
in LNKLST 3
in STEPLIB or JOBLIB 53
SEQUENCE compiler option 68
service
CICS support 5
OS PL/I 5
Shared Library replacement aid
using IBMRLSLx 180
Shared Library support 43
Shared Library, OS PL/I
sample replacement aid 180
SIS ENVIRONMENT option 64
SIZE
and FIXEDOVERFLOW 105
SIZE compiler option 68
SKIP ENVIRONMENT option 64
SMESSAGE compiler option 68
SPIE run-time option 127
SPIE runtime option 35
SQL preprocessor
EXEC SQL statements 175
new, integrated 8

SQL Preprocessor
  restrictions lifted 175
  using 175
STACK run-time option 127, 132
STACK runtime option 35, 56
STAE run-time option 127
STAE runtime option 35
STATIC
  retaining unused INTERNAL 104
  uninitialized EXTERNAL 111
  writeable reentrant 8
STEPLIB
  migration example 20
  use in migration 19
STMT compiler option 68
storage
  DASD requirements 16
  Enterprise PL/I requirements 23
  usage
    IMS considerations 58, 172
    retuning for 131
  virtual requirements 16
storage report differences 45, 130
STORAGE run-time option 127
storage utilization, improving 132
STREAM I/O
  restrictions 64
  unprintable characters 110
structure expression restrictions 65
subsystem considerations
  CICS 55, 167
  DB2 59, 175
  IMS 57, 171
subsystem performance, improving 133
symbol tables
  considerations for 53
  CSECT 53
SYSLIN DD
  restrictions 70
SYSPRINT
  LRECL value 70
  sharing between old and new PL/I 136
  support for MSGFILE(SYSPRINT) 43, 129
SYSTEM compiler option 68, 76
  CICS considerations 168
  IMS considerations 57, 171

# T

TEST compiler option 86
TOTAL environment option 115
TOTAL ENVIRONMENT option 64
TP ENVIRONMENT option 64
TRANSIENT file attribute 64
TRAP run-time option 127
TRAP runtime option 35
TRKOFL ENVIRONMENT option 64
TSO 69

# U

U4039 ABEND 38
UNDERFLOW condition 36
UNICODE
  support of 8
UNLOCK statement 64
user exits
  CEEBINT 166
  CEEBXITA 166
  IBMBEER 166
  IBMBXITA 166
  IBMFXITA 166
  installation considerations 166
  user exits 166
user information xiii
user main load module
  sample ZAP relinking aid 180
user return code differences 43

# V

VM
  support of 9

# W

WRITABLE compiler option 76, 161

# X

XUFLOW run-time option 36, 127
XUFLOW runtime option 36

# Z

ZAP, main load module relinking aid 180
ZERODIVIDE condition 115

# Readers' Comments — We'd Like to Hear from You

**Enterprise PL/I for z/OS**
**Compiler and Run-Time Migration Guide**
**Version 5 Release 1**

**Publication No. GC27-8930-00**

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

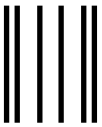
If you would like a response from IBM, please fill in the following information:

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Phone No. | Email address |

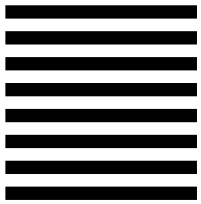
**Readers' Comments — We'd Like to Hear from You**
GC27-8930-00

Cut or Fold
Along Line

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



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



# BUSINESS REPLY MAIL

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

POSTAGE WILL BE PAID BY ADDRESSEE

IBM
H150/090
555 Bailey Avenue
San Jose, CA
USA 95141-1003

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



Cut or Fold
Along Line

Product Number: 5655-PL5

Printed in USA

GC27-8930-00