IBM z/OS Debugger
15.0.7

User's Guide

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 541.

Eighth Edition (May 2023)

This edition applies to IBM® z/OS® Debugger, 15.0.7 (Program Number 5724-T07 with the PTF for PH50796), which supports the following compilers:

- Open Enterprise SDK for Go 1.17 (Program Number 5655-GOZ)
- z/OS XL C/C++ Version 2 (Program Number 5650-ZOS)
- C/C++ feature of z/OS Version 1 (Program Number 5694-A01)
- C/C++ feature of OS/390® (Program Number 5647-A01)
- C/C++ for MVS/ESA Version 3 (Program Number 5655-121)
- AD/Cycle C/370 Version 1 Release 2 (Program Number 5688-216)
- Enterprise COBOL for z/OS 6.1, 6.2, 6.3, and 6.4 (Program Number 5655-EC6)
- Enterprise COBOL for z/OS Version 5 (Program Number 5655-W32)
- Enterprise COBOL for z/OS Version 4 (Program Number 5655-S71)
- Enterprise COBOL for z/OS and OS/390 Version 3 (Program Number 5655-G53)
- COBOL for OS/390 & VM Version 2 (Program Number 5648-A25)
- COBOL for MVS™ & VM Version 1 Release 2 (Program Number 5688-197)
- COBOL/370 Version 1 Release 1 (Program Number 5688-197)
- VS COBOL II Version 1 Release 3 and Version 1 Release 4 (Program Numbers 5668-958, 5688-023) - with limitations
- OS/VS COBOL, Version 1 Release 2.4 (5740-CB1) - with limitations
- High Level Assembler for MVS & VM & VSE Version 1 Release 4, Version 1 Release 5, Version 1 Release 6 (Program Number 5696-234)
- Enterprise PL/I for z/OS 6.1 (Program Number 5655-PL6)
- Enterprise PL/I for z/OS Version 5 Release 1, Release 2, and Release 3 (Program Number 5655-PL5)
- Enterprise PL/I for z/OS Version 4 (Program Number 5655-W67)
- Enterprise PL/I for z/OS and OS/390 Version 3 (Program Number 5655-H31)
- VisualAge® PL/I for OS/390 Version 2 Release 2 (Program Number 5655-B22)
- PL/I for MVS & VM Version 1 Release 1 (Program Number 5688-235)
- OS PL/I Version 2 Release 1, Version 2 Release 2, Version 2 Release 3 (Program Numbers 5668-909, 5668-910) - with limitations

This edition also applies to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

You can find out more about IBM z/OS Debugger by visiting the following IBM Web sites:

- IBM Debug for z/OS: https://www.ibm.com/products/debug-for-zos
- IBM Developer for z/OS: https://www.ibm.com/products/developer-for-zos
- IBM Z and Cloud Modernization Stack: https://www.ibm.com/docs/z-modernization-stack

© Copyright International Business Machines Corporation 1992, 2023.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document..... xvii
Who might use this document..... xvii
Accessing z/OS licensed documents on the Internet..... xvii
How this document is organized..... xviii
Terms used in IBM z/OS Debugger documentation..... xx
How to read syntax diagrams..... xxi
Symbols..... xxi
Syntax items..... xxii
Syntax examples..... xxii
How to provide your comments..... xxiii

Summary of changes..... xxv

Overview of IBM z/OS Debugger..... xxxiii

Part 1. Getting started with z/OS Debugger..... 1

Chapter 1. z/OS Debugger: overview..... 3
z/OS Debugger interfaces..... 5
Batch mode..... 5
Full-screen mode..... 5
Full-screen mode using the Terminal Interface Manager..... 5
Remote debug mode..... 6
IBM z/OS Debugger Utilities..... 7
IBM z/OS Debugger Utilities: Job Card..... 7
IBM z/OS Debugger Utilities: Program Preparation..... 7
IBM z/OS Debugger Utilities: z/OS Debugger Setup File..... 7
IBM z/OS Debugger Utilities: IMS TM Debugging..... 8
IBM z/OS Debugger Utilities: Load Module Analyzer..... 8
IBM z/OS Debugger Utilities: z/OS Debugger User Exit Data Set..... 8
IBM z/OS Debugger Utilities: Other IBM Application Delivery Foundation for z/OS tools..... 8
IBM z/OS Debugger Utilities: JCL for Batch Debugging..... 8
IBM z/OS Debugger Utilities: IMS BTS Debugging..... 8
IBM z/OS Debugger Utilities: JCL to Setup File Conversion..... 8
IBM z/OS Debugger Utilities: Delay Debug Profile..... 8
IBM z/OS Debugger Utilities: IMS Transaction and User ID Cross Reference Table..... 9
IBM z/OS Debugger Utilities: Non-CICS Debug Session Start and Stop Message Viewer..... 9
IBM z/OS Debugger Utilities: z/OS Debugger Code Coverage..... 9
IBM z/OS Debugger Utilities: z/OS Debugger Deferred Breakpoints..... 9
IBM z/OS Debugger Utilities: IBM z/OS Debugger JCL Wizard..... 9
Starting IBM z/OS Debugger Utilities..... 9

Chapter 2. Debugging a program in full-screen mode: introduction..... 11
Compiling or assembling your program with the proper compiler options..... 11
Starting z/OS Debugger..... 12
The z/OS Debugger full screen interface..... 12
Stepping through a program..... 14
Running your program to a specific line..... 14
Setting a breakpoint..... 14
Displaying the value of a variable..... 15

Displaying memory through the Memory window....16
Changing the value of a variable....17
Skipping a breakpoint....17
Clearing a breakpoint....17
Recording and replaying statements....18
Stopping z/OS Debugger....19

Part 2. Preparing your program for debugging....21

Chapter 3. Preparing to remote debug in standard mode....23

Chapter 4. Planning your debug session....25
Choosing compiler options for debugging....25
Choosing TEST or NOTEST compiler suboptions for COBOL programs....27
Choosing TEST or NOTEST compiler suboptions for PL/I programs....34
Choosing TEST or DEBUG compiler suboptions for C programs....39
Choosing TEST or DEBUG compiler suboptions for C++ programs....44
Understanding how hooks work and why you need them....48
Understanding what symbol tables do and why saving them elsewhere can make your application smaller....49
Choosing a debugging mode....49
Debugging in browse mode....51
Choosing a method or methods for starting z/OS Debugger....53
Choosing how to debug old COBOL programs....56
Creating deferred breakpoints for COBOL and PL/I programs....57

Chapter 5. Updating your processes so you can debug programs with z/OS Debugger....59
Update your compilation, assembly, and linking process....59
Compiling your program without using IBM z/OS Debugger Utilities....59
Compiling your program by using IBM z/OS Debugger Utilities....61
Compiling a Enterprise PL/I program on an HFS or zFS file system....62
Compiling your C program with c89 or c++....62
Compiling a C program on an HFS or zFS file system....63
Compiling a C++ program on an HFS or zFS file system....63
Update your library and promotion process....64
Make the modifications necessary to implement your preferred method of starting z/OS Debugger....64

Chapter 6. Preparing a LangX COBOL program....67
Compiling your OS/VS COBOL program....67
Compiling your VS COBOL II program....68
Compiling your Enterprise COBOL program....68
Creating the EQALANGX file for LangX COBOL programs....68
Link-editing your program....69

Chapter 7. Preparing an assembler program....71
Before you assemble your program....71
Assembling your program....71
Creating the EQALANGX file for an assembler program....71
Assembling your program and creating EQALANGX....72
Link-editing your program....73
Restrictions for link-editing your assembler program....74

Chapter 8. Preparing a Db2 program....75
Processing SQL statements....75
Linking Db2 programs for debugging....76
Binding Db2 programs for debugging....77

Chapter 9. Preparing a Db2 stored procedures program 79

Chapter 10. Preparing a CICS program 81
Link-editing EQADCCXT into your program 81
Creating and storing a DTCN profile 82
Displaying a list of active DTCN profiles and managing DTCN profiles 85
Description of fields on the DTCN Primary Menu screen 86
Description of fields on the DTCN Menu 2 screen 90
Description of fields on the DTCN Advanced Options screen 91
Creating and storing debugging profiles with CADP 92
Starting z/OS Debugger for non-Language Environment programs under CICS 92
Passing runtime parameters to z/OS Debugger for non-Language Environment programs under CICS 92

Chapter 11. Preparing an IMS program 95
Starting z/OS Debugger under IMS by using CEEUOPT or CEEROPT 95
Managing runtime options for IMSplex users by using IBM z/OS Debugger Utilities 95
Setting up the DFSBXITA user exit routine 96

Chapter 12. Specifying the TEST runtime options through the Language Environment user exit 97
Editing the source code of CEEBXITA 98
Modifying the naming pattern 98
Modifying the message display level 99
Modifying the call back routine registration 99
Activate the cross reference function and modifying the cross reference table data set name 100
Comparing the two methods of linking CEEBXITA 100
Linking the CEEBXITA user exit into your application program 100
Linking the CEEBXITA user exit into a private copy of a Language Environment runtime module 101
Creating and managing the TEST runtime options data set 101
Creating and managing the TEST runtime options data set by using Terminal Interface Manager (TIM) 102
Creating and managing the TEST runtime options data set by using IBM z/OS Debugger Utilities 103

Part 3. Starting z/OS Debugger 105

Chapter 13. Writing the TEST runtime option string 107
Special considerations while using the TEST run-time option 107
Simple TEST option 107
Defining TEST suboptions in your program 108
Suboptions and NOTEST 108
Implicit breakpoints 108
Primary commands file and USE file 108
Running in batch mode 108
Starting z/OS Debugger at different points 108
Session log 109
Precedence of Language Environment runtime options 109
Example: TEST runtime options 110
Specifying additional run-time options with VS COBOL II and PL/I programs 113
Specifying the STORAGE run-time option 113
Specifying the TRAP(ON) run-time option 113
Specifying TEST run-time option with #pragma runopts in C and C++ 114

Chapter 14. Starting z/OS Debugger from the IBM z/OS Debugger Utilities 115
Creating the setup file 115
Editing an existing setup file 115

Copying information into a setup file from an existing JCL....116
Entering file allocation statements, runtime options, and program parameters....116
Saving your setup file....117
Starting your program....117

Chapter 15. Starting z/OS Debugger from a program....119
Starting z/OS Debugger with CEETEST....119
Additional notes about starting z/OS Debugger with CEETEST....121
Example: using CEETEST to start z/OS Debugger from C/C++....122
Example: using CEETEST to start z/OS Debugger from COBOL....123
Example: using CEETEST to start z/OS Debugger from PL/I....124
Starting z/OS Debugger with PLITEST....125
Starting z/OS Debugger with the __ctest() function....126

Chapter 16. Starting z/OS Debugger in batch mode....129
Example: JCL that runs z/OS Debugger in batch mode....129
Modifying the example to debug in full-screen mode....130

Chapter 17. Starting z/OS Debugger for batch or TSO programs....131
Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal....131
Starting z/OS Debugger for programs that start in Language Environment....133
Example: Allocating z/OS Debugger load library data set....134
Example: Allocating z/OS Debugger files....134
Starting z/OS Debugger for programs that start outside of Language Environment....134
Passing parameters to EQANMDBG....135
Example: Modifying JCL that invokes an assembler Db2 program running in a batch TSO environment....137

Chapter 18. Starting z/OS Debugger under CICS....139
Comparison of methods for starting z/OS Debugger under CICS....139
Starting z/OS Debugger under CICS by using DTCN....140
Ending a CICS debugging session that was started by DTCN....141
Example: How z/OS Debugger chooses a CICS program for debugging....141
Starting z/OS Debugger for CICS programs by using CADP....141
Starting z/OS Debugger under CICS by using CEEUOPT....141
Starting z/OS Debugger under CICS by using compiler directives....142

Chapter 19. Starting a debug session....143

Chapter 20. Starting z/OS Debugger in other environments....145
Starting z/OS Debugger from Db2 stored procedures....145

Part 4. Debugging your programs in full-screen mode....147

Chapter 21. Using full-screen mode: overview....149
z/OS Debugger session panel....149
Session panel header....150
Source window....152
Monitor window....152
Log window....153
Memory window....154
Command pop-up window....155
List pop-up window....155
Creating a preferences file....156
Displaying the source....156
Changing which file appears in the Source window....157

Entering commands on the session panel ... 158
Order in which z/OS Debugger accepts commands from the session panel ... 160
Using the session panel command line ... 160
Issuing system commands ... 160
Entering prefix commands on specific lines or statements ... 161
Entering multiple commands in the Memory window ... 162
Using commands that are sensitive to the cursor position ... 162
Using Program Function (PF) keys to enter commands ... 162
Initial PF key settings ... 163
Retrieving previous commands ... 163
Composing commands from lines in the Log and Source windows ... 164
Opening the Command pop-up window to enter long z/OS Debugger commands ... 164
Navigating through z/OS Debugger windows ... 164
Moving the cursor between windows ... 165
Switching between the Memory window and Log window ... 165
Scrolling through the physical windows ... 165
Enlarging a physical window ... 166
Scrolling to a particular line number ... 166
Finding a string in a window ... 167
Displaying the line at which execution halted ... 169
Navigating through the Memory window ... 169
Creating a commands file ... 171
Recording your debug session in a log file ... 172
Creating the log file ... 172
Recording how many times each source line runs ... 173
Recording the breakpoints encountered ... 173
Setting breakpoints to halt your program at a line ... 174
Setting breakpoints in a load module that is not loaded or in a program that is not active ... 174
Controlling how z/OS Debugger handles warnings about invalid data in comparisons ... 174
Stepping through or running your program ... 175
Recording and replaying statements ... 176
Saving and restoring settings, breakpoints, and monitor specifications ... 178
Saving and restoring automatically ... 180
Disabling the automatic saving and restoring of breakpoints, monitors, and settings ... 181
Restoring manually ... 181
Performance considerations in multi-enclave environments ... 182
Displaying and monitoring the value of a variable ... 182
One-time display of the value of variables ... 183
Adding variables to the Monitor window ... 184
Displaying the Working-Storage Section of a COBOL program in the Monitor window ... 184
Displaying the data type of a variable in the Monitor window ... 185
Replacing a variable in the Monitor window with another variable ... 185
Adding variables to the Monitor window automatically ... 186
How z/OS Debugger handles characters that cannot be displayed in their declared data type ... 189
Modifying characters that cannot be displayed in their declared data type ... 189
Formatting values in the Monitor window ... 189
Displaying values in hexadecimal format ... 190
Monitoring the value of variables in hexadecimal format ... 190
Modifying variables or storage by using a command ... 190
Modifying variables or storage by typing over an existing value ... 191
Opening and closing the Monitor window ... 192
Displaying and modifying memory through the Memory window ... 192
Modifying memory through the hexadecimal data area ... 192
Managing file allocations ... 192
Displaying error numbers for messages in the Log window ... 193
Displaying a list of compile units known to z/OS Debugger ... 194
Requesting an attention interrupt during interactive sessions ... 194
Ending a full-screen debug session ... 195

Chapter 22. Debugging a COBOL program in full-screen mode....197
Example: sample COBOL program for debugging....197
Halting when certain routines are called in COBOL....200
Identifying the statement where your COBOL program has stopped....200
Modifying the value of a COBOL variable....200
Halting on a COBOL line only if a condition is true....201
Debugging COBOL when only a few parts are compiled with TEST....202
Capturing COBOL I/O to the system console....202
Displaying raw storage in COBOL....203
Getting a COBOL routine traceback....203
Tracing the run-time path for COBOL code compiled with TEST....203
Generating a COBOL run-time paragraph trace....204
Finding unexpected storage overwrite errors in COBOL....205
Halting before calling an invalid program in COBOL....205

Chapter 23. Debugging a LangX COBOL program in full-screen mode....207
Example: sample LangX COBOL program for debugging....207
Defining a compilation unit as LangX COBOL and loading debug information....209
Defining a compilation unit in a different load module as LangX COBOL....209
Halting when certain LangX COBOL programs are called....210
Identifying the statement where your LangX COBOL program has stopped....210
Displaying and modifying the value of LangX COBOL variables or storage....210
Halting on a line in LangX COBOL only if a condition is true....210
Debugging LangX COBOL when debug information is only available for a few parts....211
Getting a LangX COBOL program traceback....211
Finding unexpected storage overwrite errors in LangX COBOL....211

Chapter 24. Debugging a PL/I program in full-screen mode....213
Example: sample PL/I program for debugging....213
Halting when certain PL/I functions are called....216
Identifying the statement where your PL/I program has stopped....216
Modifying the value of a PL/I variable....216
Halting on a PL/I line only if a condition is true....217
Debugging PL/I when only a few parts are compiled with TEST....217
Displaying raw storage in PL/I....218
Getting a PL/I function traceback....218
Tracing the run-time path for PL/I code compiled with TEST....218
Finding unexpected storage overwrite errors in PL/I....219
Halting before calling an undefined program in PL/I....220

Chapter 25. Debugging a C program in full-screen mode....221
Example: sample C program for debugging....221
Halting when certain functions are called in C....224
Modifying the value of a C variable....224
Halting on a line in C only if a condition is true....225
Debugging C when only a few parts are compiled with TEST....225
Capturing C output to stdout....226
Capturing C input to stdin....226
Calling a C function from z/OS Debugger....226
Displaying raw storage in C....227
Debugging a C DLL....227
Getting a function traceback in C....227
Tracing the run-time path for C code compiled with TEST....227
Finding unexpected storage overwrite errors in C....228
Finding uninitialized storage errors in C....229
Halting before calling a NULL C function....229

Chapter 26. Debugging a C++ program in full-screen mode....231
Example: sample C++ program for debugging....231
Halting when certain functions are called in C++....234
Modifying the value of a C++ variable....235
Halting on a line in C++ only if a condition is true....236
Viewing and modifying data members of the this pointer in C++....236
Debugging C++ when only a few parts are compiled with TEST....236
Capturing C++ output to stdout....237
Capturing C++ input to stdin....237
Calling a C++ function from z/OS Debugger....238
Displaying raw storage in C++....238
Debugging a C++ DLL....238
Getting a function traceback in C++....238
Tracing the run-time path for C++ code compiled with TEST....239
Finding unexpected storage overwrite errors in C++....240
Finding uninitialized storage errors in C++....240
Halting before calling a NULL C++ function....241

Chapter 27. Debugging an assembler program in full-screen mode....243
Example: sample assembler program for debugging....243
Defining a compilation unit as assembler and loading debug data....245
Deferred LDDs....246
Re-appearance of an assembler CU....246
Multiple compilation units in a single assembly....246
Loading debug data from multiple CSECTs in a single assembly using one LDD command....247
Loading debug data from multiple CSECTs in a single assembly using separate LDD commands....247
Debugging multiple CSECTs in a single assembly after the debug data is loaded....247
Halting when certain assembler routines are called....248
Identifying the statement where your assembler program has stopped....248
Displaying and modifying the value of assembler variables or storage....248
Converting a hexadecimal address to a symbolic address....249
Halting on a line in assembler only if a condition is true....249
Getting an assembler routine traceback....249
Finding unexpected storage overwrite errors in assembler....250

Chapter 28. Customizing your full-screen session....251
Defining PF keys....251
Defining a symbol for commands or other strings....251
Customizing the layout of physical windows on the session panel....252
Opening and closing physical windows....252
Resizing physical windows....253
Zooming a window to occupy the whole screen....253
Customizing session panel colors....253
Customizing profile settings....254
Saving customized settings in a preferences file....256
Saving and restoring customizations between z/OS Debugger sessions....257

Part 5. Debugging your programs by using z/OS Debugger commands....259

Chapter 29. Entering z/OS Debugger commands....261
Using uppercase, lowercase, and DBCS in z/OS Debugger commands....261
DBCS....261
Character case and DBCS in C and C++....261
Character case in COBOL and PL/I....262
Abbreviating z/OS Debugger keywords....262

Entering multiline commands in full-screen ... 263
Entering multiline commands in a commands file ... 263
Entering multiline commands without continuation ... 263
Using blanks in z/OS Debugger commands ... 264
Entering comments in z/OS Debugger commands ... 264
Using constants in z/OS Debugger commands ... 264
Getting online help for z/OS Debugger command syntax ... 265

Chapter 30. Debugging COBOL programs ... 267
z/OS Debugger commands that resemble COBOL statements ... 267
COBOL command format ... 267
COBOL compiler options in effect for z/OS Debugger commands ... 268
COBOL reserved keywords ... 268
Using COBOL variables with z/OS Debugger ... 268
Accessing COBOL variables ... 269
Assigning values to COBOL variables ... 269
Example: assigning values to COBOL variables ... 269
Displaying values of COBOL variables ... 270
Using DBCS characters in COBOL ... 271
%PATHCODE values for COBOL ... 271
Declaring session variables in COBOL ... 272
z/OS Debugger evaluation of COBOL expressions ... 273
Displaying the results of COBOL expression evaluation ... 273
Using constants in COBOL expressions ... 274
Using z/OS Debugger functions with COBOL ... 274
Using %HEX with COBOL ... 274
Using the %STORAGE function with COBOL ... 275
Qualifying variables and changing the point of view in COBOL ... 275
Qualifying variables in COBOL ... 275
Changing the point of view in COBOL ... 276
Considerations when debugging a COBOL class ... 277
Debugging VS COBOL II programs ... 277
Finding the listing of a VS COBOL II program ... 278

Chapter 31. Debugging a LangX COBOL program ... 279
Loading a LangX COBOL program's debug information ... 279
z/OS Debugger session panel while debugging a LangX COBOL program ... 279
Restrictions for debugging a LangX COBOL program ... 280
%PATHCODE values for LangX COBOL programs ... 281
Restrictions for debugging non-Language Environment programs ... 281

Chapter 32. Debugging PL/I programs ... 283
z/OS Debugger subset of PL/I commands ... 283
PL/I language statements ... 283
%PATHCODE values for PL/I ... 284
PL/I conditions and condition handling ... 285
Entering commands in PL/I DBCS freeform format ... 286
Initializing z/OS Debugger for PL/I programs when TEST(ERROR, ...) run-time option is in effect ... 286
z/OS Debugger enhancements to LIST STORAGE PL/I command ... 286
PL/I support for z/OS Debugger session variables ... 286
Accessing PL/I program variables ... 287
Accessing PL/I structures ... 287
z/OS Debugger evaluation of PL/I expressions ... 289
Supported PL/I built-in functions ... 289
Using SET WARNING PL/I command with built-in functions ... 290
Unsupported PL/I language elements ... 291
Debugging OS PL/I programs ... 291
Restrictions while debugging Enterprise PL/I programs ... 291

Chapter 33. Debugging C and C++ programs 293
z/OS Debugger commands that resemble C and C++ commands 293
Using C and C++ variables with z/OS Debugger 294
Accessing C and C++ program variables 294
Displaying values of C and C++ variables or expressions 294
Assigning values to C and C++ variables 295
%PATHCODE values for C and C++ 295
Declaring session variables with C and C++ 296
C and C++ expressions 296
Calling C and C++ functions from z/OS Debugger 298
C reserved keywords 299
C operators and operands 299
Language Environment conditions and their C and C++ equivalents 300
z/OS Debugger evaluation of C and C++ expressions 300
Intercepting files when debugging C and C++ programs 301
Scope of objects in C and C++ 303
Storage classes in C and C++ 304
Blocks and block identifiers for C 304
Blocks and block identifiers for C++ 305
Example: referencing variables and setting breakpoints in C and C++ blocks 305
Scope and visibility of objects in C and C++ programs 306
Blocks and block identifiers in C and C++ programs 306
Displaying environmental information for C and C++ programs 307
Qualifying variables and changing the point of view in C and C++ 307
Qualifying variables in C and C++ 307
Changing the point of view in C and C++ 308
Example: using qualification in C 308
Stepping through C++ programs 309
Setting breakpoints in C++ 310
Setting breakpoints in C++ using AT ENTRY/EXIT 310
Setting breakpoints in C++ using AT CALL 311
Examining C++ objects 311
Example: displaying attributes of C++ objects 311
Monitoring storage in C++ 312
Example: monitoring and modifying registers and storage in C 312

Chapter 34. Debugging an assembler program 315
The SET ASSEMBLER and SET DISASSEMBLY commands 315
Loading an assembler program's debug information 315
z/OS Debugger session panel while debugging an assembler program 316
%PATHCODE values for assembler programs 317
Using the STANDARD and NOMACGEN view 319
Debugging non-reentrant assembler 320
Manipulating breakpoints in non-reentrant assembler load modules 320
Manipulating local variables in non-reentrant assembler load modules 320
Restrictions for debugging an assembler program 320
Restrictions for debugging a Language Environment assembler MAIN program 322
Restrictions on setting breakpoints in the prologue of Language Environment assembler programs 322
Restrictions for debugging non-Language Environment programs 322
Restrictions for debugging assembler code that uses instructions as data 322
Restrictions for debugging self-modifying assembler code 323
Restrictions for debugging assembler programs that consist of multiple sections 324
Restrictions for debugging assembler programs when SET DEFAULT VIEW NOMACGEN is in use 325

Chapter 35. Debugging a disassembled program 327

The SET ASSEMBLER and SET DISASSEMBLY commands....327
Capabilities of the disassembly view....327
Starting the disassembly view....328
The disassembly view....328
Performing single-step operations in the disassembly view....329
Setting breakpoints in the disassembly view....329
Restrictions for debugging self-modifying code....329
Displaying and modifying registers in the disassembly view....330
Displaying and modifying storage in the disassembly view....330
Changing the program displayed in the disassembly view....330
Restrictions for the disassembly view....330

Part 6. Debugging in different environments....331

Chapter 36. Debugging Db2 programs....333
Debugging Db2 programs in batch mode....333
Debugging Db2 programs in full-screen mode....333

Chapter 37. Debugging Db2 stored procedures....335
Resolving some common problems while debugging Db2 stored procedures....335

Chapter 38. Debugging IMS programs....337
Using IMS Transaction Isolation to create a private message-processing region and select transactions to debug....337
Using IMS pseudo wait-for-input (PWFI) with IMS Transaction Isolation....339
Debugging IMS batch programs interactively by running BTS in TSO foreground....340
Debugging non-Language Environment IMS BTS programs....340
Debugging IMS batch programs in batch mode....341
Debugging non-Language Environment IMS MPPs....341
Verifying configuration and starting a region for non-Language Environment IMS MPPs....341
Choosing an interface and gathering information for non-Language Environment IMS MPPs..342
Running the EQASET transaction for non-Language Environment IMS MPPs....342
Debugging Language Environment IMS MPPs without issuing /SIGN ON....344
Syntax of the EQASET transaction for Language Environment MPPs....344
Creating setup file for your IMS program by using IBM z/OS Debugger Utilities....344
Using IMS message region templates to dynamically swap transaction class and debug in a private message region....345
Placing breakpoints in IMS applications to avoid the appearance of z/OS Debugger becoming unresponsive....347

Chapter 39. Debugging CICS programs....349
Displaying the contents of channels and containers....349
Controlling pattern-match breakpoints with the DISABLE and ENABLE commands....351
Preventing z/OS Debugger from stopping at EXEC CICS RETURN....352
Early detection of CICS storage violations....353
Saving settings while debugging a pseudo-conversational CICS program....353
Saving and restoring breakpoints and monitor specifications for CICS programs....353
Restrictions when debugging under CICS....354
Accessing CICS resources during a debugging session....355
Accessing CICS storage before or after a debugging session....355

Chapter 40. Debugging ISPF applications....357

Chapter 41. Debugging programs in a production environment....359
Fine-tuning your programs for z/OS Debugger....359
Removing hooks....359
Removing statement and symbol tables....360

Debugging without hooks, statement tables, and symbol tables....360
Debugging optimized COBOL programs....362

Chapter 42. Debugging UNIX System Services programs....365
Debugging MVS POSIX programs....365

Chapter 43. Debugging non-Language Environment programs....367
Debugging exclusively non-Language Environment programs....367
Debugging MVS batch or TSO non-Language Environment initial programs....367
Debugging CICS non-Language Environment assembler or non-Language Environment COBOL initial programs....367

Part 7. Debugging complex applications....369

Chapter 44. Debugging multilanguage applications....371
z/OS Debugger evaluation of HLL expressions....371
z/OS Debugger interpretation of HLL variables and constants....371
HLL variables....372
HLL constants....372
z/OS Debugger commands that resemble HLL commands....372
Qualifying variables and changing the point of view....373
Qualifying variables....373
Changing the point of view....374
Handling conditions and exceptions in z/OS Debugger....375
Handling conditions in z/OS Debugger....375
Handling exceptions within expressions (C and C++ and PL/I only)....376
Debugging multilanguage applications....376
Debugging an application fully supported by Language Environment....377
Using session variables across different programming languages....377
Creating a commands file that can be used across different programming languages....379
Coexistence with other debuggers....379
Coexistence with unsupported HLL modules....379

Chapter 45. Debugging multithreading programs....381
Restrictions when debugging multithreading applications....381

Chapter 46. Debugging across multiple processes and enclaves....383
Starting z/OS Debugger within an enclave....383
Viewing z/OS Debugger windows across multiple enclaves....383
Ending a z/OS Debugger session within multiple enclaves....383
Using z/OS Debugger commands within multiple enclaves....384

Chapter 47. Debugging a multiple-enclave interlanguage communication (ILC) application....389

Chapter 48. Debugging programs called by Java native methods....391

Chapter 49. Solving problems in complex applications....393
Debugging programs loaded from library lookaside (LLA)....393
Debugging user programs that use system prefixed names....393
Displaying system prefixes....394
Debugging programs with names similar to system components....394
Debugging programs containing data-only modules....394
Optimizing the debugging of large applications....394
Using explicit debug mode to load debug data for only specific modules....395
Excluding specific load modules and compile units....396
Displaying current NAMES settings....396
Using the EQAOPTS NAMES command to include or exclude the initial load module....396

Using delay debug mode to delay starting of a debug session 397
Debugging subtasks created by the ATTACH assembler macro..... 398
Debugging tasks running under a generic user ID by using Terminal Interface Manager 399

Appendix A. Data sets used by z/OS Debugger..... 401

Appendix B. How does z/OS Debugger locate source, listing, or separate debug files?..... 407
Remote debugging in standard mode..... 407
Non-remote debugging and remote debugging in Debug Tool compatibility mode..... 408
How does z/OS Debugger locate source and listing files?..... 410
How does z/OS Debugger locate COBOL source during code coverage..... 410
How does z/OS Debugger locate COBOL and PL/I separate debug files..... 411
How does z/OS Debugger locate EQALANGX files..... 411
How does z/OS Debugger locate the C/C++ source file and the .dbg file?..... 412
How does z/OS Debugger locate the C/C++ .mdbg file?..... 412

Appendix C. Examples: Preparing programs and modifying setup files with IBM z/OS Debugger Utilities..... 415
Creating personal data sets..... 415
Starting IBM z/OS Debugger Utilities..... 416
Compiling or assembling your program by using IBM z/OS Debugger Utilities..... 416
Modifying and using a setup file..... 419
Run the program in foreground..... 419
Run the program in batch..... 419

Appendix D. IBM z/OS Debugger JCL Wizard..... 421
Invoking the IBM z/OS Debugger JCL Wizard..... 422
Viewing help in the panel..... 423
Commands and parameters..... 425
Debugging a Language Environment program using Terminal Interface Manager..... 426
Debugging a Language Environment program using a remote debugger without Debug Manager..... 430
Debugging a Language Environment program using a remote debugger with Debug Manager..... 434
Debugging a non-Language Environment program using Terminal Interface Manager..... 437
Debugging a Language Environment Db2 program using a remote debugger with Debug Manager..... 442
Debugging a non-Language Environment Db2 program using a remote debugger with Debug Manager..... 444
Starting z/OS Debugger Code Coverage..... 448
Without a debug session..... 448
With a debug session using Terminal Interface Manager..... 450
Debugging a Language Environment VS COBOL II program compiled with the NOTEST option by using the Terminal Interface Manager..... 452
Debugging a Language Environment COBOL program that calls non-Language Environment subprograms..... 455
Removing JCL statements..... 459

Appendix E. z/OS Debugger Code Coverage..... 461
Overview of z/OS Debugger Code Coverage..... 461
Introduction to z/OS Debugger Code Coverage..... 461
Collecting code coverage observations with z/OS Debugger..... 462
Code coverage selection and extraction process..... 462
Code coverage reporting process..... 463
Code coverage Viewer..... 464
Code coverage by using z/OS Debugger..... 465
Setup..... 465
Generating code coverage extracted observations..... 468

IBM z/OS Debugger Utilities Option E..... 471
Annotated listing format..... 479
Batch facilities..... 483
Batch examples..... 484
Generating code coverage for CICS transactions..... 485
Generating code coverage in IMS Transaction Isolation..... 485
XML tags for code coverage..... 486
XML tags definition for the Observation file..... 486
XML tag hierarchy for the Observation file..... 489
XML Tags used in the Options file..... 489
XML tags used in the Selection file..... 489

Appendix F. Notes on debugging in batch mode..... 493

Appendix G. Using IMS message region templates to dynamically swap transaction class and debug in a private message region..... 495

Appendix H. Displaying and modifying CICS storage with DTST..... 497
Starting DTST..... 497
Examples of starting DTST..... 497
Modifying storage through the DTST storage window..... 499
Navigating through the DTST storage window..... 499
DTST storage window..... 500
Navigation keys for help screens..... 501
Syntax of the DTST transaction..... 501
Examples..... 503

Appendix I. z/OS Debugger Load Module Analyzer..... 505
Choosing a method to start Load Module Analyzer..... 505
Starting the Load Module Analyzer by using JCL..... 505
Starting the Load Module Analyzer by using IBM z/OS Debugger Utilities..... 505
Description of the JCL statements to use with Load Module Analyzer..... 505
Description of DD names used by Load Module Analyzer..... 506
Description of parameters used by Load Module Analyzer..... 507
Description of EQASYSPF file format..... 508
Description of EQAPGMNM file format..... 509
Description of program output created by Load Module Analyzer..... 510
Description of output contents created by Load Module Analyzer..... 510
Example: Output created by Load Module Analyzer for an OS/VS COBOL load module..... 510
Example: Compiler options output created by Load Module Analyzer..... 511

Appendix J. Running NEWCOPY on programs by using DTNP transaction..... 513

Appendix K. Using the IBM Debug Tool plug-ins..... 515
Migrating to the z/OS Debugger Profiles view..... 517
Instrument JCL for Debug Tool Debugging plug-in..... 518
Debug Tool Code Coverage plug-in..... 520
Load Module Analyzer plug-in..... 522
Locating the trace file of the DTCN Profile, the DTSP Profile, Instrument JCL for Debugging, Code Coverage, and Load Module Analyzer view..... 524
Example: .debugtool.dtcn.trace file..... 524
Examples: .debugtool.dtsp.trace files..... 525
Examples: .debugtool.bjfd.trace files..... 525

Appendix L. Debugging a program processed by the Automatic Binary Optimizer for z/OS..... 527

Appendix M. Limitations of 64-bit support in Debug Tool compatibility mode.....529

Appendix N. Debugging programs compiled with IBM Open Enterprise SDK for Go..... 531

Appendix O. Support resources and problem solving information..... 535
Accessing the IBM Support portal..... 535
Getting fixes..... 535
Subscribing to support updates..... 535
Contacting IBM Support..... 536
Determine the business impact of your problem..... 536
Gather diagnostic information..... 536
Submit the problem to IBM Support..... 537

Appendix P. Accessibility..... 539
Using assistive technologies 539
Keyboard navigation of the user interface..... 539
Accessibility of this document..... 539

Notices.....541
Copyright license..... 541
Privacy policy considerations..... 542
Programming interface information..... 542
Trademarks and service marks..... 542
Glossary.....543
IBM z/OS Debugger publications.....553

Index..... 555

About this document

z/OS Debugger combines the richness of the z/OS environment with the power of Language Environment® to provide a debugger for programmers to isolate and fix their program bugs and test their applications. z/OS Debugger gives you the capability of testing programs in batch, using a nonprogrammable terminal in full-screen mode, or using a workstation interface to remotely debug your programs.

Who might use this document

This document is intended for programmers using z/OS Debugger to debug high-level languages (HLLs) with Language Environment and assembler programs either with or without Language Environment. Throughout this document, the HLLs are referred to as C, C++, COBOL, and PL/I.

z/OS Debugger runs on the z/OS operating system and supports the following subsystems:

- CICS®
- Db2®
- IMS
- JES batch
- TSO
- UNIX System Services in remote debug mode or full-screen mode using the Terminal Interface Manager only

To use this document and debug a program written in one of the supported languages, you need to know how to write, compile, and run such a program.

Accessing z/OS licensed documents on the Internet

z/OS licensed documentation is available on the Internet in PDF format at the IBM Resource Link® Web site at:

```
http://www.ibm.com/servers/resourcelink
```

Licensed documents are available only to customers with a z/OS license. Access to these documents requires an IBM Resource Link user ID and password, and a key code. With your z/OS order you received a Memo to Licensees, (GI10-8928), that includes this key code.

To obtain your IBM Resource Link user ID and password, log on to:

```
http://www.ibm.com/servers/resourcelink
```

To register for access to the z/OS licensed documents:

1. Sign in to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.

Note: You cannot access the z/OS licensed documents unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

Printed licensed documents are not available from IBM.

You can use the PDF format on either **z/OS Licensed Product Library CD-ROM** or IBM Resource Link to print licensed documents.

© Copyright IBM Corp. 1992, 2023

How this document is organized

Note: Chapters 2, 14, 21 to 28, and Appendices C, E to J are not applicable to IBM Developer for z/OS (non-Enterprise Edition), IBM Z and Cloud Modernization Stack (Wazi Code). In addition, Chapters 3, 6, 31 and Appendix L are not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

This document is divided into areas of similar information for easy retrieval of appropriate information. The following list describes how the information is grouped:

- Part 1 groups together introductory information about z/OS Debugger. The following list describes each chapter:
  - Chapter 1 introduces z/OS Debugger and describes some of its features.
  - Chapter 2 describes a simple scenario of how to use z/OS Debugger in full-screen mode, introducing you to some basic commands that you might use frequently.
- Part 2 groups together information about how to prepare programs for debugging. The following list describes each chapter:
  - Chapter 3 describes how to compile your program to prepare to remote debug in standard mode.
  - Chapter 4 describes how to choose compiler options, debugging mode, and runtime options so that you can prepare programs for debugging. It also describes your options for debugging COBOL programs compiled with compilers that are now out-of-service.
  - Chapter 5 describes how to implement the choices you made in chapter 4.
  - Chapter 6 describes how to prepare a LangX COBOL program.
  - Chapter 7 describes how to prepare an assembler program.
  - Chapter 8 describes how to prepare a Db2 program.
  - Chapter 9 describes how to prepare a Db2 stored procedures program.
  - Chapter 10 describes how to prepare a CICS program.
  - Chapter 11 describes how to prepare an IMS program.
  - Chapter 12 describes how to include a call to the TEST runtime option into a program.
- Part 3 groups together information that describes the different methods you can use to start z/OS Debugger. The following list describes each chapter:
  - Chapter 13 describes how to write the TEST runtime option to indicate how and when you want to start z/OS Debugger.
  - Chapter 14 describes how to start z/OS Debugger from IBM z/OS Debugger Utilities.
  - Chapter 15 describes how to start z/OS Debugger from a program.
  - Chapter 16 describes how to start z/OS Debugger in batch mode.
  - Chapter 17 describes how to start z/OS Debugger for your batch or TSO programs.
  - Chapter 18 describes how to start z/OS Debugger from CICS programs.
  - Chapter 19 describes how to start z/OS Debugger in full-screen mode.
  - Chapter 20 describes how to start z/OS Debugger in full-screen mode using the Terminal Interface Manager. This chapter also describes some tips to starting z/OS Debugger from a stored procedure.
- Part 4 groups together information about how to debug a program in full-screen mode and provides an example of how to debug a C, COBOL, and PL/I program in full-screen mode. The following list describes each chapter:
  - Chapter 21 provides overview information about full-screen mode.
  - Chapter 22 provides a sample COBOL program to describe how to debug it in full-screen mode.
  - Chapter 23 provides a sample OS/VS COBOL program as representative of non-Language Environment COBOL programs to describe how to debug it in full-screen mode.
  - Chapter 24 provides a sample PL/I program to describe how to debug it in full-screen mode.

  - Chapter 25 provides a sample C program to describe how to debug it in full-screen mode.
  - Chapter 26 provides a sample C++ program to describe how to debug it in full-screen mode.
  - Chapter 27 provides a sample assembler program to describe how to debug it in full-screen mode.
  - Chapter 28 describes how to modify the appearance of a full-screen mode debugging session and save those changes, as well as other settings, into files.
- Part 5 groups together information about how to enter and use z/OS Debugger commands.
  - Chapter 29 provides information about entering mixed case commands, using DBCS characters, abbreviating commands, entering multiline commands, and entering comments.
  - Chapter 30 describes how to use z/OS Debugger commands to debug COBOL programs.
  - Chapter 31 describes how to use z/OS Debugger commands to debug LangX COBOL programs.
  - Chapter 32 describes how to use z/OS Debugger commands to debug PL/I programs.
  - Chapter 33 describes how to use z/OS Debugger commands to debug C or C++ programs.
  - Chapter 34 describes how to use z/OS Debugger commands to debug assembler programs.
  - Chapter 35 describes how to use z/OS Debugger commands to debug disassembly programs.
- Part 6 groups together information about debugging Db2, Db2 stored procedures, IMS, CICS, ISPF, UNIX System Services, and production-level programs.
  - Chapter 36 describes how to debug a Db2 program.
  - Chapter 37 describes how to debug a Db2 stored procedure.
  - Chapter 38 describes how to debug an IMS program.
  - Chapter 39 describes how to debug a CICS program.
  - Chapter 40 describes how to debug an ISPF program.
  - Chapter 41 describes how to debug a production-level program.
  - Chapter 42 describes how to debug a program running in the UNIX System Services shell.
  - Chapter 43 describes how to debug programs that do not start or run in Language Environment.
- Part 7 groups together information about how to debug programs written in multiple language or running in multiple processes.
  - Chapter 44 describes how to debug a program written in multiple languages.
  - Chapter 45 describes the restrictions when you debug a multithreaded program.
  - Chapter 46 describes how to debug a program that runs across multiple processes and enclaves.
  - Chapter 47 describes how to debug a multiple-enclave interlanguage communication (ILC) application.
  - Chapter 48 describes how to debug programs that are called by Java™ native methods.
  - Chapter 49 describes how to solve various problems when debugging complex applications.
- Part 8 groups together appendixes. The following list describes each appendix:
  - Appendix A describes the data sets that z/OS Debugger uses to retrieve and store information.
  - Appendix B describes the process z/OS Debugger uses to locate source, listing, or side files.
  - Appendix C provides an example that guides you through the process of preparing a sample program and modifying existing setup files by using IBM z/OS Debugger Utilities.
  - Appendix D describes the IBM z/OS Debugger JCL Wizard.
  - Appendix E describes how to use z/OS Debugger Code Coverage.
  - Appendix F describes notes on debugging in batch mode.
  - Appendix G describes using IMS message region templates to dynamically swap transaction class and debug in a private message region.
  - Appendix H describes how to use the DTST transaction to display and modify CICS storage.

- Appendix I describes how to use Load Module Analyzer, a stand-alone program that is shipped with z/OS Debugger.
- Appendix J describes how you can use the DTNP transaction, supplied by z/OS Debugger, to load a new copy of a program into an active CICS region.
- Appendix K describes how to install the IBM Debug Tool DTCN Profile Manager, DTSP Profile Manager, Instrument JCL for Debugging, z/OS Debugger Code Coverage, and Load Module Analyzer plug-ins.
- Appendix L describes how to debug a load module or program object processed by the Automatic Binary Optimizer for z/OS.
- Appendix M describes limitations of 64-bit support in Debug Tool compatibility mode.
- Appendix N describes how to debug programs compiled with IBM Open Enterprise SDK for Go.
- Appendix O describes the resources that are available to help you solve any problems you might encounter with z/OS Debugger.
- Appendix P describes the features and tools available to people with physical disabilities that help them use z/OS Debugger and z/OS Debugger documents.

The last several topics list notices, bibliography, and glossary of terms.

Terms used in this document

Because of differing terminology among the various programming languages supported by z/OS Debugger, as well as differing terminology between platforms, a group of common terms is established. The following table lists these terms and their equivalency in each language.

z/OS Debugger term	C and C++ equivalent	COBOL or LangX COBOL equivalent	PL/I equivalent	assembler
Compile unit	C and C++ source file	Program	• Program • PL/I source file for Enterprise PL/I • A package statement or the name of the main procedure for Enterprise PL/I[1]	CSECT
Block	Function or compound statement	Program, nested program, method, or PERFORM group of statements	Block	CSECT
Label	Label	Paragraph name or section name	Label	Label

Note:

1. The PL/I program must be compiled with and run in one of the following environments:
   - Compiled with Enterprise PL/I for z/OS, Version 3.6 or later
   - Compiled with Enterprise PL/I for z/OS, Version 3.5, with the PTFs for APARs PK35230 and PK35489 applied

z/OS Debugger provides facilities that apply only to programs compiled with specific levels of compilers. Because of this, this document uses the following terms:

assembler
Refers to assembler programs with debug information assembled by using the High Level Assembler (HLASM).

COBOL
Refers to the all COBOL compilers supported by z/OS Debugger except the COBOL compilers described in the term *LangX COBOL*.

Disassembly or disassembled
Refers to high-level language programs compiled without debug information or assembler programs without debug information. The debugging support z/OS Debugger provides for these programs is through the disassembly view.

Enterprise PL/I
Refers to the Enterprise PL/I for z/OS and OS/390 and the VisualAge PL/I for OS/390 compilers.

LangX COBOL
Refers to any of the following COBOL programs that are supported through use of the EQALANGX debug file:

- Programs compiled using the IBM OS/VS COBOL compiler.
- Programs compiled using the VS COBOL II compiler with the NOTEST compiler option.
- Programs compiled using the Enterprise COBOL for z/OS V3 and V4 compiler with the NOTEST compiler option.

When you read through the information in this document, remember that OS/VS COBOL programs are non-Language Environment programs, even though you might have used Language Environment libraries to link and run your program.

VS COBOL II programs are non-Language Environment programs when you link them with the non-Language Environment library. VS COBOL II programs are Language Environment programs when you link them with the Language Environment library.

Enterprise COBOL programs are always Language Environment programs. Note that COBOL DLL's cannot be debugged as LangX COBOL programs.

Read the information regarding non-Language Environment programs for instructions on how to start z/OS Debugger and debug non-Language Environment COBOL programs, unless information specific to LangX COBOL is provided.

PL/I
Refers to all levels of PL/I compilers. Exceptions will be noted in the text that describe which specific PL/I compiler is being referenced.

How to read syntax diagrams

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol
Definition

►►—
Indicates the beginning of the syntax diagram.

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

Indicates that the syntax is continued from the previous line.

Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

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

Item type
Definition

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

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

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

Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples

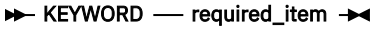
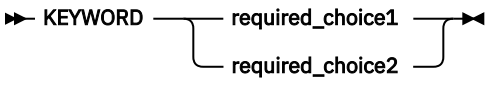
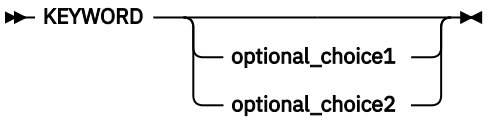
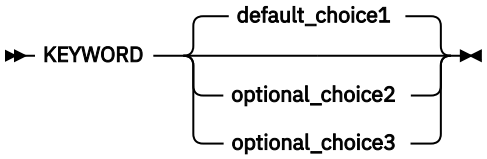
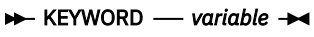
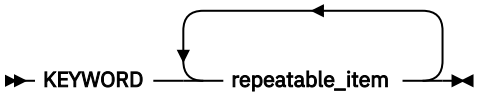
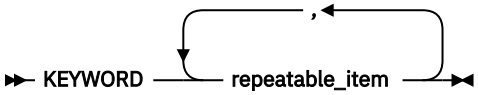
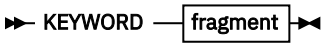
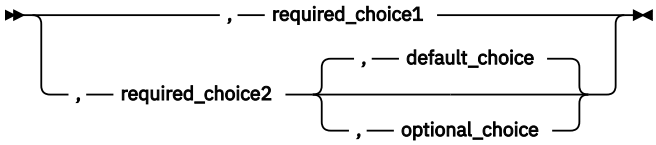
Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item. Optional items appear below the main path of the horizontal line.	

Table 1. Syntax examples (continued)

Item	Syntax example
Optional choice. An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	
Default. Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	
Variable. Variables appear in lowercase italics. They represent names or values.	
Repeatable item. An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. A character within the arrow means you must separate repeated items with that character. An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	 
Fragment. The ┤ fragment ├ symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	 **fragment** 

How to provide 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 z/OS Debugger documentation, you can leave a comment in IBM Documentation:

- IBM Developer for z/OS and IBM Developer for z/OS Enterprise Edition: https://www.ibm.com/docs/en/developer-for-zos
- IBM Debug for z/OS: https://www.ibm.com/docs/debug-for-zos
- IBM Z and Cloud Modernization Stack: https://www.ibm.com/docs/z-modernization-stack

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.

Summary of changes

15.0.7

z/OS Debugger Profiles view

- On the **IBM z/OS Debugger Preferences** page, you can now choose whether to automatically synchronize debug profiles in the view with those in the remote system when you establish an RSE connection. For more information, see the "Setting debug preferences" topic in IBM Documentation.

15.0.6

Debug Profile Service

- On the **IBM z/OS Debugger Preferences** page, you can specify to ignore the SSL certificate errors when the Debug Profile Service that you want to connect to does not have a valid SSL certificate. For more information, see the "Setting debug preferences" topic in IBM Documentation.

15.0.5

z16, Compilers and CICS

- Support is added for the new IBM z16 hardware.
- Support is added for CICS Transaction Server for z/OS 6.1.
- Support is added for the following compiler versions in both 31-bit mode and 64-bit mode:
  - Enterprise COBOL for z/OS 6.4
  - Enterprise PL/I for z/OS 6.1

User-defined functions

- User-defined functions are supported for programs compiled with Enterprise COBOL for z/OS 6.4.

64-bit support

- Debug Tool compatibility mode now supports delay debug.
- The following EQAOPTS commands are now supported:
  - DLAYDBG
  - DLAYDBGDSN
  - DLAYDBGTRC

  For the remaining limitations, see Appendix M, “Limitations of 64-bit support in Debug Tool compatibility mode,” on page 529.

Interoperability

- Interoperability is now supported between 31-bit and 64-bit COBOL programs. Use delay debug mode to improve efficiency.
- You can now debug 31-bit COBOL programs called from 64-bit Java programs. Delay debug mode is required when you debug this type of application.

  For more information, see Using delay debug mode to delay starting of a debug session.

Automatic Binary Optimizer for z/OS

- Support is added for Automatic Binary Optimizer for z/OS 2.2.
- In Debug Tool compatibility mode, you can now debug programs compiled with Enterprise COBOL for z/OS Version 5 or later that have been optimized by Automatic Binary Optimizer for z/OS 2.2. The optimized programs can be debugged in the same manner as those that are not optimized.

© Copyright IBM Corp. 1992, 2023

Code Coverage

- You can now filter code coverage results for z/OS batch applications using existing JCL and z/OS UNIX applications by specifying a filter list file in the launch configuration. For more information, see "Filtering code coverage results during collection" in IBM Documentation.
- The module exclude list is deprecated and replaced by the filter list file.

Debug profile

- As a system administrator of Debug Profile Service, you can now use z/OS Debugger Profile Management to identify and delete CICS (DTCN) profiles that might interfere with z/OS system resources. For more information, see "Managing profiles with z/OS Debugger Profile Management" in *IBM z/OS Debugger Customization Guide*.

Host configuration

- When not run as a started task, Debug Manager now queries your security product for explicit permission to start. For more information, see "Starting Debug Manager as a user job" in *IBM z/OS Debugger Customization Guide*.

15.0.4

IBM Z® Open Debug

- You can now connect to a debug session and start debugging when you list debug sessions that are parked on a z/OS machine.
- With the Wazi for VS Code IDE, you can now use a single action to activate a debug profile, launch and debug an application.

Code Coverage

- You can now merge and export code coverage results on z/OS from command line into a single file of various formats with the `ccexport.sh` command. For more information, see "Merging and exporting code coverage results from z/OS" in IBM Documentation.
- Code Coverage Service can now be started as part of Remote Debug Service. For more information, see "Generating code coverage in headless mode using Remote Debug Service" in IBM Documentation.
- When started via the headless code coverage collector, Code Coverage Service now supports secured connections, and requires authentication.
- In the **Code Coverage Results** view, you can now add a secured Code Coverage Service result location (https). You can add and clear untrusted certificates in the CCS keystore file. For more information, see "Working with result locations" in IBM Documentation.
- The code coverage output location specified in the `o,output` parameter is ignored in the startup key unless you specify `-a,allowoutputlocation=TRUE` in the command line when you start the code coverage collector and use headless code coverage. For more information, see "Specifying code coverage options in the startup key" and "Starting and stopping the headless code coverage daemon" in IBM Documentation.
- The Code Coverage API documentation is updated from version 10.1.2 to 11.0.0.

IBM Open Enterprise SDK for Go

- In Debug Tool compatibility mode, you can now debug Go programs compiled with IBM Open Enterprise SDK for Go 1.17. For more information, see Appendix N, “Debugging programs compiled with IBM Open Enterprise SDK for Go,” on page 531.

Decimal point

- In Debug Tool compatibility mode, if the `DECIMAL-POINT IS COMMA` clause is specified in a COBOL program compiled with Enterprise COBOL for z/OS Version 6 Release 3 (UI78163) or later, the debugger displays decimals as commas in the **Variables** and **Monitors** views in Eclipse IDEs, and expressions accept commas, in addition to periods, as decimal points.

Host configuration

- You can now configure the `eqahcc.enc` file to start Code Coverage Service as part of Remote Debug Service. For more information, see the "Customizing with the sample job EQARMTSU" topic in *IBM z/OS Debugger Customization Guide*.
- With Debug Manager, you can leverage Dynamic Virtual IP Addressing (DVIPA) available in IBM Explorer for z/OS to concurrently run identical setups on different systems in your sysplex, and have TCP/IP, optionally with the help of WLM, distribute the client connections among these systems. Ensure that each Debug Manager has a unique external port per system and the port is explicitly defined in TCP/IP definitions. For more information, see the "Distributed Dynamic VIPA" section in *IBM z/OS Debugger Customization Guide*.
- MVS data set *`userid`*`.EQATIOUT` is no longer needed when you install and configure the IMS transaction isolation extension for Eclipse IDE users.

15.0.3

z/OS 2.5

- Support is added for z/OS 2.5.

IBM Open Enterprise SDK for Go

- In Debug Tool compatibility mode, you can now debug Go programs compiled with IBM Open Enterprise SDK for Go 1.16. For more information, see Appendix N, “Debugging programs compiled with IBM Open Enterprise SDK for Go,” on page 531.

64-bit support

- Debug Tool compatibility mode now supports playback for 64-bit COBOL programs. For the remaining limitations, see Appendix M, “Limitations of 64-bit support in Debug Tool compatibility mode,” on page 529.

Source Level Code Coverage for COBOL

- When you start a code coverage session with the Eclipse IDE or headless code coverage, you can now choose to use the source listing. Source level code coverage offers direct mapping between code coverage entries and the program source, to exclude the need to post process the code coverage data. Source level code coverage improves integration with tools like ZUnit and SonarQube as part of an automated pipeline. For more information, see “How does z/OS Debugger locate COBOL source during code coverage” on page 410.

Code Coverage Service API

- Code Coverage Service (CCS) RESTful API is now available to enable custom extensions. For more information, see "Code Coverage Service RESTful API Documentation" in IBM Documentation.

z/OS Debugger Profiles view

- Remote IMS Application with Isolation launch configurations have been replaced by the IMS Isolation debug profiles. All existing IMS launch configurations are automatically migrated to the **z/OS Debugger Profiles view**. You can create and activate IMS Isolation profiles in the view to debug and run code coverage for IMS transactions in private regions.

  To use this function, ensure that the system programmer installed and configured the IMS transaction isolation extension for the ADFz Common Components server. If you want to configure the region name for the private region, ask the system programmer to update z/OS Debugger to 15.0.3 or later, with the PTF for APAR PH41774 applied.

  Note: IMS Isolation profiles are only available in IBM Developer for z/OS Enterprise Edition.
- You can now view the Remote System Explorer z/OS connection status in the view. A new option **Refresh z/OS Connections** is provided in the view toolbar to establish all z/OS connections and synchronize the profiles.
- You can now duplicate the content from an existing debug profile to create a new one efficiently.

- Generic profiles might trigger z/OS Debugger unexpectedly and consume unnecessary resources. When you activate a generic profile, warnings are now displayed. You can choose to hide the warnings.
- In the Debug Profile Editor, you can now save a debug profile without activating it, and leave it for future use.

For more information, see "Managing debug profiles with the z/OS Debugger Profiles view" in IBM Documentation.

Debug Profile Service

- You now only need to expose one port to use Debug Profile Service. A new configuration switch is added to `eqaprof.env` to select whether to use secure HTTP protocol. For more information, see "Customizing with the sample job EQAPRFSU" in *IBM z/OS Debugger Customization Guide*.

z/OS Debugger commands

- The following commands are now supported in Debug Tool compatibility mode for remote debugging:
  - STEP
  - GO
  - RUNTO
  - JUMPTO
  - COMMENT

Dark theme

- Dark theme is now supported for remote debugger in the Eclipse IDE.

Message EQA9924U

- Message ICH408I is now issued in the console with EQA9924U to provide you with more information to address the issue.

15.0.2

IBM Z Open Debug

- You can now debug High Level Assembler (HLASM) z/OS programs with IBM Z Open Debug.
- For Wazi for Dev Spaces, the log files are now in `/projects/.debug/logs`.

Code Coverage

- In the **Code Coverage Results** view, you can now export code coverage results in Cobertura format. For more information, see "Exporting code coverage results in Cobertura format" in IBM Documentation.
- You can now specify parameters in the startup key to generate code coverage results in Cobertura and SonarQube formats. In addition, short parameters values `-e,exportertype=SQ|PDF|COB` are added for you to use both in the startup key and in the headless code coverage daemon. For more information, see "Specifying code coverage options in the startup key" and "Starting and stopping the headless code coverage daemon" in IBM Documentation.
- When you view code coverage results in an editor, you can now see a code coverage summary of the included files for PL/I source files with `%INCLUDE` statements. For more information, see "Viewing code coverage results in an editor" in IBM Documentation.

z/OS Batch Applications launches

- In the **Remote Systems** or **z/OS Projects** view, or when you are editing the JCL source in the editor, after you choose **Debug As** or **Code Coverage As** from the menu, the following options are available:
  - **z/OS Batch Application**: Launch a debug or code coverage session without a debug profile.

- **z/OS Batch Application with a debug profile**: Launch a debug or code coverage session with a debug profile.
- **z/OS Batch Application ...**: Create a launch configuration to launch a debug or code coverage session.

For more information, see "Launching a debug session for z/OS batch applications using existing JCL" in IBM Documentation.

IBM z/OS Debugger JCL Wizard

- The **Program/Procedure Selection List** panel is updated to include procedures, in addition to programs. Selecting a procedure will provide a panel to enter the procedure step override for the DD statements generated. The After (A) and Before (B) line commands are no longer required.
- You can now select **SVC screening** to enable SVC screening for batch non-Language Environment programs.
- You can now select **Intercept on** to show COBOL DISPLAY statements on the IBM z/OS Debugger log or Debug Console in the Eclipse IDE.
- Error messages are improved.
- The END command (**PF3**) in the **Program/Procedure Selection List** panel is modified to cancel the request. Previously, selecting **PF3** would not exit the panel.
- When the process is completed, the cursor is now placed on the command line.
- The **z/OS Debugger LDD Generation for Non-LE Programs** panel is now populated with the initial program name and subprograms selected in the **Request AT ENTRY Sub-Program Breakpoints** panel. Program names provided in this panel can be modified.
- After you select **Code Coverage** from the parameters selection panel, the EQAXOPT lines are generated to specify `CCPROGSELECTDSN`, `CCOUTPUTDSN` and `CCOUTPUTDSNALLOC`, if the `CODE_COVERAGE_SETUP` value is configured to YES in the EQAJCL REXX procedure.
- Previously the wizard would verify the program source members identified in the **z/OS Debugger LDD Generation for Non-LE Programs** panel with each library identified by the **z/OS Debugger Debug Libraries** panel to verify that the members are present. This function is removed because z/OS Debugger now flags any such members in the IBM z/OS Debugger log or Debug Console in the Eclipse IDE when the LDD command is entered.

For more information, see IBM z/OS Debugger JCL Wizard

AT LABEL * command

- For Enterprise COBOL for z/OS Version 5 and later, `AT LABEL *` now highlights the labels similar to statement breakpoints.
- You can now use PF6 or `AT LINE` to remove or add a single global label hook if `AT LABEL *` was issued. To disable this functionality, use `DISABLE AT LABEL *`.
- If you issue `AT LABEL *` again or use `ENABLE AT LABEL *`, the global label hooks are reset. The hooks that you removed are added back.

For more information, see "AT LABEL command" in *IBM z/OS Debugger Reference and Messages*.

15.0.1

64-bit support

- Debug Tool compatibility mode now supports the following features:
  - Code coverage
  - Source entry breakpoints
  - CEETEST

  For the remaining limitations, see Appendix M, “Limitations of 64-bit support in Debug Tool compatibility mode,” on page 529.

Code Coverage

- With Concurrent Debug and Code Coverage, you can run code coverage collection in parallel with the active debug session in the Eclipse IDE. The code coverage data is collected during the debug run, and code coverage annotations are displayed and updated in the debug editor. Concurrent Debug and Code Coverage requires z/OS Debugger 15.0.1 or later. For more information, see the "Generating code coverage in a remote debug session" topic in IBM Documentation.
- Headless code coverage report can now be exported with a Cobertura exporter. For more information, see the "Starting and stopping the headless code coverage daemon" topic in IBM Documentation.
- Headless code coverage collector now supports filtering of module, compiler units, and files. For more information, see the "Filtering code coverage results" topic in IBM Documentation.

Debug Profile Editor

- In the Debug Profile Editor of the Eclipse IDE, new key bindings are available to show the error tooltip and the overall error summary. For more information, see the "Debug profile key bindings" topic in IBM Documentation.

Debug Profile Service

- As an alternative of a keystore file, you can now use a RACF managed key ring to enable secure communication with Debug Profile Service. For more information, see the "Enabling secure communication with a RACF managed key ring" section in *IBM z/OS Debugger Customization Guide*.
- A new optional HOST attribute is added to the CICS region configuration. For more information, see the instructions in the `/etc/debug/dtcn.ports` sample configuration file.
- The Debug Profile Service API now provides more detailed diagnostic messages when authentication fails.

IBM Z Open Debug

- Log files can now be found in the user's home directory.

CICS trace entries

- A new parameter, DNT, is added to the CICS startup parameter INITPARM to support disabling generation of z/OS Debugger trace entries. For more information, see the "Adding support for debugging under CICS" topic in *IBM z/OS Debugger Customization Guide*.

15.0.0

64-bit support

- Debug Tool compatibility mode now supports the following compiler features:
  - The 64-bit COBOL feature of z/OS for COBOL V6.3 and later
  - The 64-bit C/C++ feature of z/OS

  For the limitations, see Appendix M, “Limitations of 64-bit support in Debug Tool compatibility mode,” on page 529.

  The PTFs for z/OS Language Environment APARs PH26071 and PH28997 are required for this support.

IBM Z Open Debug

- IBM Z Open Debug is now also available with the Wazi for Dev Spaces IDE, in addition to the Wazi for VS Code IDE. Both IDEs are offered in IBM Wazi Developer for Red Hat CodeReady Workspaces and IBM Developer for z/OS Enterprise Edition. For a comparison of features provided in different products and IDEs, see Overview of IBM z/OS Debugger.
- You can now specify `TEST(,,,RDS:*)` for the TEST runtime option to start a debug session using Remote Debug Service for Wazi for VS Code or Wazi for Dev Spaces.

Code Coverage

- Headless code coverage for z/OS is now included with IBM Debug for z/OS. Use the headless code coverage collector to generate code coverage results of tests that are run as part of your DEVOPS pipeline. For more information, see the "Generating code coverage in headless mode using a daemon" section in IBM Documentation.
- Single letter parameters are now supported in the headless code coverage collector command line and in EQA_STARTUP_KEY when you use JCL. For more information, see topics "Starting and stopping the headless code coverage daemon" and "Specifying code coverage options in the startup key" in IBM Knowledge Center.
- Support is added for PL/I programs compiled with `LISTVIEW(SOURCE)` to generate code coverage results for main program and all `%INCLUDE` files. For more information, see the "Supported compilers and options for code coverage in Debug Tool compatibility mode" topic in IBM Knowledge Center.
- The **Code Coverage Results** view of the Eclipse IDE now supports CCS result locations. You can add a CCS result location which collects and retrieves code coverage data by using RESTful API, and interact with the results under the CCS result location in the same way as locally stored results. CCS result locations require Headless Code Coverage 15.0.0 or later. For more information, see the "Viewing code coverage results in the Code Coverage Results view" topic in IBM Knowledge Center.
- You can now also use Remote Debug Service to collect code coverage results similar to the headless code coverage collector for IBM Wazi Developer for Red Hat CodeReady Workspaces or IBM Developer for z/OS Enterprise Edition. For more information, see the "Generating code coverage in headless mode using Remote Debug Service" topic in IBM Knowledge Center.

Debug Profile Editor

- In the Debug Profile Editor of the Eclipse IDE, you can now use the quick outline to navigate to a field. For more information, see the "Quick outline for the Debug Profile Editor" topic in IBM Knowledge Center.

z/OS XL C/C++

- Support is added for `DEBUG(NOFILE)`. For more information, see Choosing DEBUG compiler suboptions for C programs and Choosing DEBUG compiler suboptions for C++ programs.

Debug Tool Plug-ins

- The following Debug Tool plug-ins of the Eclipse IDE are deprecated and will be removed in the next release:
  - DTCN Profile Manager plug-in
  - DTSP Profile Manager plug-in
  - Instrument JCL for Debug Tool Debugging plug-in
  - Debug Tool Code Coverage plug-in
  - Load Module Analyzer plug-in

  You can use the **z/OS Debugger Profiles** view to create and manage debug profiles, z/OS batch applications launches to dynamically instrument and submit JCL, and the **Code Coverage Results** view to work with compiled code coverage results. For more information, see the following topics in IBM Documentation: Managing debug profiles with the z/OS Debugger Profiles view, Launching a debug session for z/OS batch applications using existing JCL, and Viewing code coverage results in the Code Coverage Results view.

Load Module Analyzer

- The Load Module Analyzer is deprecated and will be removed in a future version.

Host configuration

- Remote Debug Service can now be configured to collect headless code coverage. For more information, see the "Adding support for Remote Debug Service" section in *IBM z/OS Debugger Customization Guide*.
- The record size for the DTCN VSAM file is increased to 3000 bytes. To use the DTCN VSAM repository with z/OS Debugger 15.0, create a new file using the SEQASAMP(EQAWCRVS) sample JCL. You can also convert your existing VSAM file to the new record size and format using the EQADPCNV utility. For more information, see the "Migrating a debug profiles VSAM file from an earlier release" topic in *IBM z/OS Debugger Customization Guide*.
- The IMS Transaction Isolation Facility is enhanced to utilize type 2 IMS commands for retrieving information on transactions, in cases where the type 1 commands that are normally used are disallowed. For more information, see the "Scenario F: Enabling the Transaction Isolation Facility" topic in *IBM z/OS Debugger Customization Guide*.

Overview of IBM z/OS Debugger

IBM z/OS Debugger is the next iteration of IBM debug technology on IBM Z and consolidates the IBM Integrated Debugger and IBM Debug Tool engines into one unified technology. IBM z/OS Debugger is progressing towards one remote debug mode based on Debug Tool compatibility mode. In support of this direction, Debug Tool compatibility mode, when available in the user interface, is selected by default for V14.1.2 or later.

IBM z/OS Debugger is a host component that supports various debug interfaces, like the Eclipse and Visual Studio Code IDEs. z/OS Debugger and the supported debug interfaces are provided with the following products:

IBM Developer for z/OS Enterprise Edition

This product is included in IBM Application Delivery Foundation for z/OS. IBM Developer for z/OS Enterprise Edition provides all the debug features.

IBM Developer for z/OS Enterprise Edition currently provides debug functions in the following IDEs:

- IBM Developer for z/OS Eclipse
- Wazi for Dev Spaces, through IBM Z Open Debug
- Wazi for VS Code, through IBM Z Open Debug

See Table 3 on page xxxv for the debug features supported in different IDEs.

IBM Developer for z/OS

IBM Developer for z/OS is a subset of IBM Developer for z/OS Enterprise Edition. IBM Developer for z/OS, previously known as IBM Developer for z Systems or IBM Rational® Developer for z Systems®, is an Eclipse-based integrated development environment for creating and maintaining z/OS applications efficiently.

IBM Developer for z/OS includes all enhancements in IBM Developer for z/OS Enterprise Edition except for the debug features noted in Table 2 on page xxxiv.

IBM Debug for z/OS

IBM Debug for z/OS is a subset of IBM Developer for z/OS Enterprise Edition. IBM Debug for z/OS focuses on debugging solutions for z/OS application developers. See Table 2 on page xxxiv for the debug features supported.

IBM Debug for z/OS does not provide advanced developer features that are available in IBM Developer for z/OS Enterprise Edition.

For information about how to install the IBM Debug for z/OS Eclipse IDE, see Installing the IBM Debug for z/OS Eclipse IDE.

IBM Z and Cloud Modernization Stack

IBM Z and Cloud Modernization Stack brings together component capabilities from IBM Z into an integrated platform that is optimized for Red Hat OpenShift Container Platform. With this solution, you can analyze the impact of application changes on z/OS, create and deploy APIs for z/OS applications, work on z/OS applications with cloud native tools, and standardize ID automation for z/OS. Starting from 2.0, Wazi Code is delivered in IBM Z and Cloud Modernization Stack. Wazi Code 1.x is still available in IBM Wazi Developer for Red Hat CodeReady Workspaces.

The debug functions are available in the IDEs provided with Wazi Code:

- Wazi for Dev Spaces, through IBM Z Open Debug
- Wazi for VS Code, through IBM Z Open Debug
- Wazi for Eclipse

See Table 2 on page xxxiv and Table 3 on page xxxv for the debug features supported in the product and different IDEs.

© Copyright IBM Corp. 1992, 2023

Table 2 on page xxxiv maps out the features that differ in products. Not all the available features are listed. To find the features available in different remote IDEs, see Table 3 on page xxxv.

Table 2. Debug feature comparison

	IBM Debug for z/OS	IBM Developer for z/OS	IBM Developer for z/OS Enterprise Edition	IBM Z and Cloud Modernization Stack (Wazi Code)
Main features				
3270 interface, including z/OS Debugger Utilities	√		√	
Eclipse IDE, see Table 3 on page xxxv for feature details.[1]	√	√	√	√
IBM Z Open Debug provided with the Wazi for Dev Spaces IDE, see Table 3 on page xxxv for feature details.[1]			√	√
IBM Z Open Debug provided with the Wazi for VS Code IDE, see Table 3 on page xxxv for feature details.[1]			√	√
Code Coverage features				
Compiled Language Code Coverage[2]	√	√ [3]	√	
Headless Code Coverage	√	√	√	
Java Code Coverage		√	√	
ZUnit Code Coverage[4]		√	√	
z/OS Debugger Code Coverage (3270 and remote interfaces) [5]	√		√	
3270 features				
z/OS Debugger full screen, batch or line mode	√		√	
IMS Isolation support	√		√	

Table 2. Debug feature comparison (continued)				
	IBM Debug for z/OS	**IBM Developer for z/OS**	**IBM Developer for z/OS Enterprise Edition**	**IBM Z and Cloud Modernization Stack (Wazi Code)**
Compiler support features				
Assembler support: Create EQALANGX files	√	√	√	
Assembler support: Debugging [6]	√	√	√[7]	√[7]
LANGX COBOL support [8]	√	√	√	
Support for Automatic Binary Optimizer (ABO)	√	√	√	
Load Module Analyzer[9]	√		√	

Notes:

1. The following features are supported only in remote debug mode:
   - Support for 64-bit COBOL feature of z/OS for COBOL V6.3 and later
   - Support for 64-bit Enterprise PL/I for z/OS Version 5
   - Support for 64-bit C/C++ feature of z/OS
   - Support for IBM Open Enterprise SDK for Go 1.16.
2. Code coverage does not support Go programs.
3. IBM Developer for z/OS includes z/OS Debugger remote debug and compiled code coverage Eclipse interface, but does not include z/OS Debugger Code Coverage.
4. ZUnit Code Coverage is only supported in Debug Tool compatibility mode.
5. z/OS Debugger Code Coverage can only be enabled in the 3270 interface.
6. Debugging assembler requires that you have EQALANGX files that have been created via ADFz Common Components or a product that ships the ADFz Common Components.
7. This feature is only available with the Eclipse IDE.
8. LANGX COBOL refers to any of the following programs:
   - A program compiled with the IBM OS/VS COBOL compiler.
   - A program compiled with the IBM VS COBOL II compiler with the NOTEST compiler option.
   - A program compiled with the IBM Enterprise COBOL for z/OS Version 3 or Version 4 compiler with the NOTEST compiler option.
9. Load Module Analyzer is deprecated and will be removed in a future version.

Table 3. Remote IDE debug feature comparison		
Feature	**Eclipse-based debug interface**	**IBM Z Open Debug [1,10]**
Debug Tool compatibility mode[2]	√	√
Standard mode[3,10]	√[4]	

Table 3. Remote IDE debug feature comparison (continued)		
Feature	**Eclipse-based debug interface**	**IBM Z Open Debug [1,10]**
Integration with Language Editors[10]	• COBOL Editor[5] • PL/I Editor[5] • Remote C/C++ Editor[4,5] • System z LPEX Editor[4,5]	• Z Open Editor
Visual Debug	√[5,10]	
Debugging ZUnit tests	√[6,10]	
Debug profile management	√[4,10]	√
IMS Isolation UI	√[7]	
Integration with CICS Explorer views	√ [4,5]	
Integration with property groups	√[5,10]	
Team Debug support	√[4,5]	
Integrated launch[10]	• z/OS UNIX Application launch configuration • z/OS Batch Application using existing JCL • z/OS Batch Application using a property group[5]	
Debug Tool Plug-ins	√[4, 8]	
Modules	√	
Memory	√	
Program navigation		
Step over/Next	√	√
Step into/Step in	√	√
Step return/Step out	√	√
Jump to location	√[10]	
Run to location/Run to cursor	√[10]	√
Resume/Continue	√	√
Terminate	√	√
Animated step	√	
Playback	√[10]	
Breakpoints		
Line/statement breakpoints	√	√
Entry breakpoints	√	
Source entry breakpoints	√[10]	
Event breakpoint	√[10]	

Table 3. Remote IDE debug feature comparison (continued)		
Feature	**Eclipse-based debug interface**	**IBM Z Open Debug [1,10]**
Address breakpoint	√[10]	
Watch breakpoint	√[10]	
Variables & Registers		
Variables	√	√
Registers	√	√[9]
Modifying variable and register values	√	√
Setting variable filter	√	
Changing variable representation	√	
Dereferencing variables	√	
Displaying in memory view	√	
Monitors		
Displaying monitor	√	√
Modifying monitor value	√	
Changing variable representation	√	
Dereferencing variables	√	
Debug Console		
Evaluating variables and expressions		√
z/OS Debugger commands	√[10]	

Notes:

1. IBM Z Open Debug is provided with Wazi for Dev Spaces and Wazi for VS Code.
2. Debug Tool compatibility mode does not support 64-bit Enterprise PL/I for z/OS Version 5.
3. Standard mode does not support 64-bit COBOL feature of z/OS for COBOL V6.3 and later. Source view for COBOL V6.2 and later is supported only in standard mode.
4. This feature is not available in Wazi for Eclipse.
5. This feature is not available in IBM Debug for z/OS.
6. Debugging ZUnit tests is only supported in Debug Tool compatibility mode.
7. This feature is only available in IBM Developer for z/OS Enterprise Edition.
8. IBM Developer for z/OS includes Debug Tool plug-ins, but does not include Load Module Analyzer and z/OS Debugger Code Coverage 3270 interfaces.
9. Registers are available in the **Variables** view.
10. Programs compiled with IBM Open Enterprise SDK for Go are not supported.

Part 1. Getting started with z/OS Debugger

© Copyright IBM Corp. 1992, 2023

Chapter 1. z/OS Debugger: overview

z/OS Debugger helps you test programs and examine, monitor, and control the execution of programs that are written in assembler, C, C++, COBOL, PL/I, or Go on a z/OS system. Your applications can include other languages; z/OS Debugger provides a disassembly view where you can debug, at the machine code level, those portions of your application. However, in the disassembly view, your debugging capabilities are limited. Table 4 on page 3 and Table 5 on page 4 map out the combinations of compilers and subsystems that z/OS Debugger supports.

You can use z/OS Debugger to debug your programs in batch mode, interactively in full-screen mode, or in remote debug mode.

Table 4 on page 3 maps out the z/OS Debugger interfaces and compilers or assemblers each interface supports.

Table 4. z/OS Debugger interface type by compiler or assembler

Compiler or assembler	Batch mode	Full-screen mode	Remote debug mode [1]
Open Enterprise SDK for Go 1.16			X
Enterprise COBOL for z/OS V3 and V4 compiled with the NOTEST compiler option [2]	X	X	X
Enterprise COBOL for z/OS compiled with the TEST compiler option	X	X	X
Enterprise COBOL for z/OS and OS/390 compiled with the NOTEST compiler option [2]	X	X	
Enterprise COBOL for z/OS and OS/390 compiled with the TEST compiler option	X	X	X
COBOL for OS/390 & VM	X	X	X
COBOL for MVS & VM	X	X	X
AD/Cycle COBOL/370 Version 1 Release 1	X	X	
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (with limitations; for programs compiled with the NOTEST compiler option and linked with the Language Environment library.) [2]	X	X	
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (with limitations; for programs compiled with the NOTEST compiler option and linked with a non-Language Environment library.) [2]	X	X	
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (with limitations; for programs compiled with the TEST compiler option and linked with the Language Environment library.)	X	X	X
OS/VS COBOL, Version 1 Release 2.4 (with limitations) [2]	X	X	
Enterprise PL/I for z/OS compiled with the TEST compiler option	X	X	X
Enterprise PL/I for z/OS and OS/390 compiled with the TEST compiler option	X	X	X
PL/I for MVS & VM	X	X	
OS PL/I Version 2 Release 1, Version 2 Release 2, and Version 2 Release 3 (with limitations)	X	X	
C/C++ feature of z/OS	X	X	X

© Copyright IBM Corp. 1992, 2023

Table 4. z/OS Debugger interface type by compiler or assembler (continued)

Compiler or assembler	Batch mode	Full-screen mode	Remote debug mode [1]
C/C++ feature of OS/390 Version 2 Release 10 and later	X	X	X
C/C++ feature of OS/390 Version 1 Release 3 and earlier	X	X	
C/C++ for MVS/ESA Version 3 Release 2	X	X	
AD/Cycle C/370 Version 1 Release 2	X	X	
IBM High Level Assembler (HLASM), Version 1 Release 4, Version 1 Release 5, and Version 1 Release 6	X	X	X

Notes:

1. This column of the table is only applicable for Debug Tool compatibility mode. For standard mode, see Chapter 3, “Preparing to remote debug in standard mode,” on page 23.
2. See Chapter 6, “Preparing a LangX COBOL program,” on page 67 for information about how to prepare a program of this type.

Table 5 on page 4 maps out the z/OS Debugger interfaces and subsystems each interface supports.

Table 5. z/OS Debugger interface type by subsystem

Subsystem	Batch mode	Full-screen mode	Full-screen mode using the Terminal Interface Manager	Remote debug mode
TSO	X	X	X	X
JES batch	X		X	X
UNIX System Services			X	X
CICS		X[1]		X
Db2	X	X	X	X
Db2 stored procedures			X	X
IMS TM			X	X
IMS batch	X		X	X
IMS BTS		X	X	X
Airline Control System (ALCS)				X[2]

[1] You can use 3 different ways to debug CICS programs in full-screen mode: single terminal mode, screen control mode, and separate terminal mode.
[2] Only for C and C++ programs.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts

“z/OS Debugger interfaces” on page 5

Related tasks

Chapter 4, “Planning your debug session,” on page 25
Chapter 21, “Using full-screen mode: overview,” on page 149

Related references
IBM z/OS Debugger Reference and Messages

z/OS Debugger interfaces

The terms *full-screen mode*, *batch mode*, and *remote debug mode* identify the types of debugging interfaces that z/OS Debugger provides. Only remote debug mode supports debugging Go programs and 64-bit COBOL, PL/I, and C/C++ programs.

Batch mode

Notes:

- This mode is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).
- This mode does not support Go programs and 64-bit COBOL, PL/I, and C/C++ programs. Use remote debug mode instead to debug these programs.

You can use a z/OS Debugger commands file to predefine a series of z/OS Debugger commands to be performed on a running application. Neither terminal input, nor user interaction is available during batch mode debugging. When commands in the commands file are processed by the debugger, they can produce messages that are written to the z/OS Debugger log. Log messages are written to a log file for your review at a later time.

The term "batch mode" debugging refers to this debugging method, which is controlled by a predefined script. Note that batch mode debugging is not limited to debugging batch programs. Batch mode can be used with any type of application supported by z/OS Debugger, including online applications running under CICS, IMS/TM, or TSO.

Full-screen mode

Notes:

- This mode is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).
- This mode does not support Go programs and 64-bit COBOL, PL/I, and C/C++ programs. Use remote debug mode instead to debug these programs.

z/OS Debugger provides an interactive full-screen interface on a 3270 device, with debugging information displayed in three windows:

- A Source window in which to view your program source or listing
- A Log window, which records commands and other interactions between z/OS Debugger and your program
- A Monitor window in which to monitor changes in your program

You can debug all languages supported by z/OS Debugger in full-screen mode.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
IBM z/OS Debugger Customization Guide

Full-screen mode using the Terminal Interface Manager

Notes:

- This mode is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).
- This mode does not support Go programs and 64-bit COBOL, PL/I, and C/C++ programs. Use remote debug mode instead to debug these programs.

Full-screen mode using the Terminal Interface Manager provides the same interactive full-screen interface that full-screen mode provides and enables you to debug types of programs that you could not debug with full-screen mode. For example, you can debug a COBOL batch job running in MVS/JES, a Db2 Stored Procedure, an IMS transaction running on a IMS MPP region, or an application running in UNIX System Services.

The Terminal Interface Manager (TIM) is a component of z/OS Debugger that provides communication between the debugger, which controls an application program as it runs, and a terminal session where you interact with the debugger. To use the TIM you connect a 3270 terminal session to the TIM.

The debugger displays on that terminal session in full-screen mode and accepts your commands. You can connect to the TIM from a dedicated 3270 terminal session, for example, a terminal emulator session configured to connect to it. Optionally, you can access the TIM from VTAM® session manager software.

Contact your system administrator to determine how to access a terminal session using the TIM on your system.

Remote debug mode

In remote debug mode, the host application starts z/OS Debugger, which communicates through a TCP/IP connection to a remote debugger on your workstation. Only remote debug mode supports debugging 64-bit COBOL, PL/I, and C/C++ programs.

z/OS Debugger can work with the remote IDE to provide you with the ability to debug host programs, including batch programs, through a graphical user interface (GUI) on the workstation.

IBM z/OS Debugger supports two modes of remote debugging:

Standard mode
Uses the Program Information and Control Library (PICL) engine technology. With the PICL technology, part of the processing is performed in the client machine, which can reduce the overhead in the z/OS system. Standard mode supports 64-bit Enterprise PL/I for z/OS Version 5 and the 64-bit C/C++ feature of z/OS. Source view for COBOL V6.2 and later is supported only in standard mode.

Debug Tool compatibility mode
Uses the remote debug engine from the former Debug Tool for z/OS to perform all host debugging tasks. This mode supports all features that are available in Debug Tool for z/OS. Debug Tool compatibility mode supports the 64-bit COBOL feature of z/OS for COBOL V6.3 and later, the 64-bit C/C++ feature of z/OS, and IBM Open Enterprise SDK for Go 1.16. The following feature is provided in the Debug Tool compatibility mode only:

- Debugging and collecting code coverage for ZUnit tests

IBM z/OS Debugger is progressing towards one remote debug mode based on Debug Tool compatibility mode. In support of this direction, Debug Tool compatibility mode, when available in the user interface, is selected by default for V14.1.2 or later. Any existing launches, property groups, or updated preferences remain unchanged.

You can enter some z/OS Debugger commands through the remote debugger's Debug Console. For a list of z/OS Debugger commands that you can enter, see "z/OS Debugger commands supported in remote debug mode" in the *IBM z/OS Debugger Reference and Messages*.

Unless otherwise specified, the information about remote debug mode in *IBM z/OS Debugger User's Guide* applies only to Debug Tool compatibility mode.

For more information about remote debugging with IBM z/OS Debugger, see the IBM Developer for z/OS, IBM Z and Cloud Modernization Stack (Wazi Code) documentation in IBM Documentation.

IBM z/OS Debugger Utilities

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

IBM z/OS Debugger Utilities is a set of ISPF panels that give you access to tools that can help you manage your debugging sessions. This topic describes these tools.

IBM z/OS Debugger Utilities: Job Card

The tool (under option 0, called Job Card) helps you create a JOB card that is used by the tools in **Program Preparation** (option 1), **z/OS Debugger Setup File** (option 2), and **JCL for Batch Debugging** (option 8).

IBM z/OS Debugger Utilities: Program Preparation

The set of tools under the **Program Preparation** (option 1) can help you manage all the tasks required to compile or assemble, and link your programs. They can also help you convert older COBOL source code and copybooks to newer versions of COBOL by using COBOL and CICS Command Level Conversion Aid (CCCA). The **Program Preparation** option can be very useful if you do not have an established build process at your site. The following list describes the specific tasks that **Program Preparation** can help you do:

- Run the Db2 precompiler or the CICS translator.
- Set compiler options.
- Specify naming patterns for your data sets.
- Specify input data sets for copy processing.
- Convert, compile, and link-edit your programs in either TSO foreground or MVS batch.
- Convert, compile, and link-edit your high level language programs in either TSO foreground or MVS batch.
- Convert, assemble, and link-edit your assembler programs in either TSO foreground or MVS batch.
- Generate EQALANGX side files.
- Generate a listing from an EQALANGX or COBOL SYSDEBUG side file.
- Prepare the following COBOL programs for debugging:
  - Programs written for non-Language Environment COBOL.
  - Programs previously compiled with the CMPR2 compiler option.

  To prepare these programs, you convert the source to the newer COBOL standard and compile it with the newer compilers. After you debug your program, you can do one of the following:
  - Make changes to your non-Language Environment COBOL source and repeat the conversion and compilation every time you want to debug your program.
  - Make changes in the converted source and stop maintaining your non-Language Environment COBOL source.

IBM z/OS Debugger Utilities: z/OS Debugger Setup File

Setup files can save you time when you are debugging a program that needs to be restarted multiple times. Setup files store information needed to allocate the necessary files and run a single job-step with z/OS Debugger either in MVS batch or TSO foreground. You can create several setup files for each program; each setup file can store information about starting and running your program in different circumstances. To create and manage setup files, select **z/OS Debugger Setup File** (option 2).

IBM z/OS Debugger Utilities: IMS TM Debugging

You can create private IMS message regions to debug test applications without interfering with other regions by using one of two features:

- You can use predefined IMS message region templates to start a private IMS message region, assign a specific transaction to the region, and run that transaction in the region
- You can use the IMS Transaction Isolation function to view a list of IMS transactions for Message Processing Regions in an IMS system, and select the ones that you want to debug. You can also use this function to clone a transaction's operating environment into a private message-processing region that is reserved for your use. Any transactions that you register to debug are routed to this private environment to isolate you from other users of that same transaction and environment.

For IMSplex users, you can modify the Language Environment runtime parameters table without relinking the applications. The tools that can help you complete these tasks are found under option 4, called IMS TM Debugging.

IBM z/OS Debugger Utilities: Load Module Analyzer

Load Module Analyzer analyzes MVS load modules or program objects to determine the language translator (compiler or assembler) used to generate the object for each CSECT. The tool that can help you complete this task can be found under option 5, called Load Module Analyzer.

IBM z/OS Debugger Utilities: z/OS Debugger User Exit Data Set

This function assists you in preparing a TEST runtime option data set that is used by the z/OS Debugger Language Environment user exit. The z/OS Debugger Language Environment user exits use this TEST runtime option string to start a debug session. The tool that can help you complete this task is found under option 6, called z/OS Debugger User Exit Data Set, in IBM z/OS Debugger Utilities.

IBM z/OS Debugger Utilities: Other IBM Application Delivery Foundation for z/OS tools

This function provides an interface to the IBM File Manager ISPF functions. You can find these tools under option 7, called Other IBM Application Delivery Foundation for z/OS tools, in IBM z/OS Debugger Utilities.

IBM z/OS Debugger Utilities: JCL for Batch Debugging

Modify the JCL for a batch job so that z/OS Debugger is started when the job is run. The tool that can help you complete this task is found under option 8, called JCL for Batch Debugging, in IBM z/OS Debugger Utilities.

IBM z/OS Debugger Utilities: IMS BTS Debugging

The IMS BTS Debugging option helps you run and debug IMS BTS programs by saving, into a set up file, the information needed to create the runtime environment for the program. IBM z/OS Debugger Utilities uses the information in the set up file to create the appropriate JCL statements, which you can then run in the foreground or submit as a batch job.

IBM z/OS Debugger Utilities: JCL to Setup File Conversion

The **JCL to Setup File Conversion** option is an alternative to the **z/OS Debugger Setup File** option above. With this option, you can select from a list of JCL steps rather than from a list of JCL cards to specify what to convert to a set up file format.

IBM z/OS Debugger Utilities: Delay Debug Profile

The Delay Debug Profile function assists you in preparing a data set that contains TEST runtime options, and pattern match arguments. The data set is used by the z/OS Debugger delay debug mode to find a

match of a program name or C function name (compile unit) (along with an optional load module name). When a match is found, z/OS Debugger uses the TEST runtime option string to start a debug session. The tool that helps you complete this task is found under Option B, called Delay Debug Profile, in IBM z/OS Debugger Utilities.

IBM z/OS Debugger Utilities: IMS Transaction and User ID Cross Reference Table

The IMS Transaction and User ID Cross Reference Table contains the cross reference information between an IMS Transaction and a User ID. z/OS Debugger uses the information to find the ID of the user who wants to debug the transaction and to construct the name of the user's debug profile data set. This function is used when an IMS transaction runs using a generic ID as is in the case with transactions started using the MQ or web gateway.

IBM z/OS Debugger Utilities: Non-CICS Debug Session Start and Stop Message Viewer

The Non-CICS Debug Session Start and Stop Message Viewer allows users to browse the start and stop messages of debug sessions. You can use it to track debug sessions and identify abnormal sessions that are started but not terminated.

IBM z/OS Debugger Utilities: z/OS Debugger Code Coverage

The z/OS Debugger Code Coverage allows users to view the code coverage observations generated from the z/OS Debugger session. It also provides functions to extract and merge the code observations and generate reports.

IBM z/OS Debugger Utilities: z/OS Debugger Deferred Breakpoints

The z/OS Debugger Deferred Breakpoints allows users to create and view a list of breakpoints prior to starting the debug session. It reduces the time spent in the debugging session and also the system resource usages.

IBM z/OS Debugger Utilities: IBM z/OS Debugger JCL Wizard

The IBM z/OS Debugger JCL Wizard, an ISPF edit macro named EQAJCL, can be used to modify a JCL or procedure member and create statements to invoke z/OS Debugger in various environments.

Starting IBM z/OS Debugger Utilities

IBM z/OS Debugger Utilities can be started in one of the following ways:

- If an option was installed to access the IBM z/OS Debugger Utilities primary options ISPF panel from an existing panel, then select that option by using instructions from the installer.
- If the z/OS Debugger data sets were installed into your normal logon procedure, enter the following command from the ISPF Command Shell panel (by default set as option 6):

```
EQASTART NATLANG(language_id)
```

- If z/OS Debugger was not installed in your ISPF environment, enter this command from the ISPF Command Shell panel (by default set as option 6):

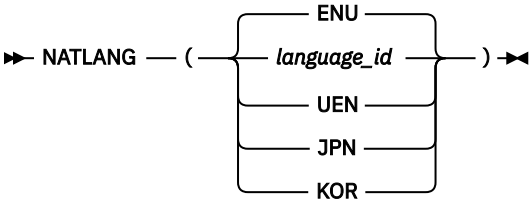
```
EX 'hlq.SEQAEXEC(EQASTART)' 'NATLANG(language_id)'
```

To determine which method to use on your system, contact your system administrator.

NATLANG(*language_id*) is optional. If you specify NATLANG(*language_id*), your settings are remembered by EQASTART and become the default on subsequent starts of EQASTART when you do not specify parameters.

NATLANG

The NATLANG parameter specifies that national language to be used to display program messages. The syntax of this parameter is:



language_id
One of the following IDs:

ENU
English

UEN
Uppercase English

JPN
Japanese

Feature needed: JPN is not a valid choice unless the JPN feature of z/OS Debugger has been installed.

KOR
Korean

Feature needed: KOR is not a valid choice unless the KOR feature of z/OS Debugger has been installed.

Chapter 2. Debugging a program in full-screen mode: introduction

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

Full-screen mode is the interface that z/OS Debugger provides to help you debug programs on a 3270 terminal. This topic describes the following tasks which make up a basic debugging session:

1. "Compiling or assembling your program with the proper compiler options" on page 11
2. "Starting z/OS Debugger" on page 12
3. After you start z/OS Debugger, you will see the full-screen mode interface. "The z/OS Debugger full screen interface" on page 12 describes the parts of the interface. Then you can do any of the following tasks:
   - "Stepping through a program" on page 14
   - "Running your program to a specific line" on page 14
   - "Setting a breakpoint" on page 14
   - "Skipping a breakpoint" on page 17
   - "Clearing a breakpoint" on page 17
   - "Displaying the value of a variable" on page 15
   - "Displaying memory through the Memory window" on page 16
   - "Changing the value of a variable" on page 17
   - "Recording and replaying statements" on page 18
4. "Stopping z/OS Debugger" on page 19

Each topic directs you to other topics that provide more information.

Compiling or assembling your program with the proper compiler options

Each programming language has a comprehensive set of compiler options. It is important to use the correct compiler options to prepare your program for debugging. The following list describes the simplest set of compiler options to use for each programming language:

Compiler options that you can use with C programs
The TEST and DEBUG compiler options provide suboptions to refine debugging capabilities. Which compiler option and suboptions to choose depends on the version of the C compiler that you are using.

Compiler options that you can use with C++ programs
The TEST and DEBUG compiler options provide suboptions to refine debugging capabilities. Which compiler option and suboptions to choose depends on the version of the C++ compiler that you are using.

Compiler options that you can use with COBOL programs
The TEST compiler option provides suboptions to refine debugging capabilities. Some suboptions are used only with a specific version of COBOL. This chapter assumes the use of suboptions available to all versions of COBOL.

Compiler options that you can use with LangX COBOL programs
When you compile your OS/VS COBOL program, the following options are required: NOTEST, SOURCE, DMAP, PMAP, VERB, XREF, NOLST, NOBATCH, NOSYMDMP, NOCOUNT.

© Copyright IBM Corp. 1992, 2023

When you compile your VS COBOL II program, the following options are required: NOOPTIMIZE, NOTEST, SOURCE, MAP, XREF, and LIST (or OFFSET).

When you compile your Enterprise COBOL for z/OS V3 and V4 program, the following options are required: NOOPTIMIZE, NOTEST, SOURCE, MAP, XREF, and LIST.

Compiler options that you can use with PL/I programs

The TEST compiler option provides suboptions to refine debugging capabilities. Some suboptions are used only with a specific version of PL/I. This chapter assumes the use of suboptions available to all versions of PL/I, except for PL/I for MVS or OS PL/I compilers, which must also specify the SOURCE suboption.

Assembler options that you can use with assembler programs

When you assemble your program, you must specify the ADATA option. Specifying this option generates a SYSADATA file, which the EQALANGX postprocessor needs to create a debug file.

See Chapter 4, “Planning your debug session,” on page 25 for instructions on how to choose the correct combination of compiler options and suboptions to use for your situation.

Starting z/OS Debugger

There are several methods to start z/OS Debugger in full-screen mode. Each method is designed to help you start z/OS Debugger for programs that are compiled with an assortment of compiler options and that run in a variety of runtime environments. Part 3, “Starting z/OS Debugger,” on page 105 describes each of these methods.

In this topic, we describe the simplest and most direct method to start z/OS Debugger for a program that runs in Language Environment in TSO. At a TSO READY prompt, enter the following command:

```
CALL 'USERID1.MYLIB(MYPROGRAM)' '/TEST'
```

Place the slash (/) before or after the TEST runtime option, depending on the programming language you are debugging.

The following topics can give you more information about other methods of starting z/OS Debugger:

- Chapter 14, “Starting z/OS Debugger from the IBM z/OS Debugger Utilities,” on page 115
- Chapter 13, “Writing the TEST runtime option string,” on page 107
- “Starting z/OS Debugger with CEETEST” on page 119
- “Starting z/OS Debugger with PLITEST” on page 125
- “Starting z/OS Debugger with the __ctest() function” on page 126
- “Starting z/OS Debugger for programs that start in Language Environment” on page 133
- Chapter 16, “Starting z/OS Debugger in batch mode,” on page 129
- “Starting z/OS Debugger for programs that start outside of Language Environment” on page 134
- “Starting z/OS Debugger under CICS by using DTCN” on page 140
- “Starting z/OS Debugger for CICS programs by using CADP” on page 141
- “Starting z/OS Debugger under CICS by using CEEUOPT” on page 141
- “Starting z/OS Debugger under CICS by using compiler directives” on page 142
- “Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 131
- “Starting z/OS Debugger from Db2 stored procedures” on page 145

The z/OS Debugger full screen interface

After you start z/OS Debugger, the z/OS Debugger screen appears:

```
COBOL    LOCATION: EMPLOOK initialization
Command ===>                                                  Scroll ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 0 OF 0
****************************** TOP OF MONITOR *******************************
****************************** BOTTOM OF MONITOR ****************************



SOURCE: EMPLOOK --1----+----2----+----3----+----4----+----5----+ LINE: 1 OF 349
       1        ************************************************************  .
       2        *                                                          *  .
       3        *                                                          *  .
       4        ************************************************************  .
       5                                                                      .
       6        ************************************************************  .
       7         IDENTIFICATION DIVISION.                                     .
       8        ************************************************************  .
       9         PROGRAM-ID.    "EMPLOOK".                                    .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 5
******************************** TOP OF LOG *********************************
IBM z/OS Debugger 15.0.n
08/04/2020 03:55:40 AM
5724-T07: Copyright IBM Corp. 1992, 2020
PF  1:?           2:STEP       3:QUIT       4:LIST       5:FIND      6:AT/CLEAR
PF  7:UP          8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG 12:RETRIEVE
```

The default screen is divided into four sections: the session panel header and three physical windows. The sessional panel header is the top two lines of the screen, which display the header fields and a command line. The header fields describe the programming language and the location in the program. The command line is where you enter z/OS Debugger commands.

A physical window is the space on the screen dedicated to the display of a specific type of debugging information. The debugging information is organized into the following types, called logical windows:

Monitor window
Variables and their values, which you can display by entering the `SET AUTOMONITOR ON` and `MONITOR` commands.

Source window
The source or listing file, which z/OS Debugger finds or you can specify where to find it.

Log window
The record of your interactions with z/OS Debugger and the results of those interactions.

Memory window
A section of memory, which you can display by entering the `MEMORY` command.

The default screen displays three physical windows, with one assigned the Monitor window, the second assigned the Source window, and the third assigned the Log window. You can swap the Memory window with the Log window.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Entering commands on the session panel” on page 158
“Navigating through z/OS Debugger windows” on page 164
“Customizing the layout of physical windows on the session panel” on page 252

Related references
“z/OS Debugger session panel” on page 149
MEMORY command in *IBM z/OS Debugger Reference and Messages*
MONITOR command in *IBM z/OS Debugger Reference and Messages*
SET AUTOMONITOR command in *IBM z/OS Debugger Reference and Messages*
WINDOW SWAP command in *IBM z/OS Debugger Reference and Messages*

Stepping through a program

Stepping through a program means that you run a program one line at a time. After each line is run, you can observe changes in program flow and storage. These changes are displayed in the Monitor window, Source window, and Log window. Use the STEP command to step through a program.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Stepping through or running your program" on page 175

Running your program to a specific line

You can run from one point in a program to another point by using one of the following methods:

- Set a breakpoint and use the GO command. This command runs your program from the point where it stopped to the breakpoint that you set. Any breakpoints that are encountered cause your program to stop. The RUN command is synonymous with the GO command.
- Use the GOTO command. This command resumes your program at the point that you specify in the command. The code in between is skipped.
- Use the JUMPTO command. This command moves the point at which your program resumes running to the statement you specify in the command; however, the program does not resume. The code in between is skipped.
- Use the RUNTO command. This command runs your program to the point that you specify in the RUNTO command. The RUNTO command is helpful when you haven't set a breakpoint at the point you specify in the RUNTO command.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

IBM z/OS Debugger Reference and Messages

Setting a breakpoint

In z/OS Debugger, breakpoints can indicate a stopping point in your program and a stopping point in time. Breakpoints can also contain activities, such as instructions to run, calculations to perform, and changes to make.

A basic breakpoint indicates a stopping point in your program. For example, to stop on line 100 of your program, enter the following command on the command line:

```
AT 100
```

In the Log window, the message `AT 100 ;` appears. If line 100 is not a valid place to set a breakpoint, the Log window displays a message similar to `Statement 100 is not valid`. The breakpoint is also indicated in the Source window by a reversing of the colors in the prefix area.

Breakpoints do more than just indicate a place to stop. Breakpoints can also contain instructions. For example, the following breakpoint instructs z/OS Debugger to display the contents of the variable *myvar* when z/OS Debugger reaches line 100:

```
AT 100 LIST myvar;
```

A breakpoint can contain instructions that alter the flow of the program. For example, the following breakpoint instructs z/OS Debugger to go to label `newPlace` when it reaches line 100:

```
AT 100 GOTO newPlace ;
```

A breakpoint can contain a condition, which means that z/OS Debugger stops at the breakpoint only if the condition is met. For example, to stop at line 100 only when the value of *myvar* is greater than 10, enter the following command:

```
AT 100 WHEN myvar > 10;
```

A breakpoint can contain complex instructions. In the following example, when z/OS Debugger reaches line 100, it alters the contents of the variable *myvar* if the value of the variable *mybool* is true:

```
AT 100 if (mybool == TRUE) myvar = 10 ;
```

The syntax of the complex instruction depends on the program language that you are debugging. The previous example assumes that you are debugging a C program. If you are debugging a COBOL program, the same example is written as follows:

```
AT 100 if mybool = TRUE THEN myvar = 10 ; END-IF ;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

Displaying the value of a variable

After you are familiar with setting breakpoints and running through your program, you can begin displaying the value of a variable. The value of a variable can be displayed in one of the following ways:

- One-time display (in the Log window) is useful for quickly checking the value of a variable.

  For one-time display, enter the following command on the command line, where *x* is the name of the variable:

  ```
  LIST (x)
  ```

  The Log window shows a message in the following format:

  ```
  LIST ( x ) ;
  x = 10
  ```

  Alternatively, you can enter the L prefix command in the prefix area of the Source window. In the following line from the Source window, type in L2 in the prefix area, then press Enter to display the value of *var2*:

  ```
  200            var1 = var2 + var3;
  ```

  z/OS Debugger creates the command `LIST (var2)`, runs it, then displays the following message in the Log window:

  ```
  LIST ( VAR2 ) ;
  VAR2 = 50
  ```

  You can use the L prefix command only with programs assembled or compiled with the following assemblers or compilers:

  - Enterprise PL/I for z/OS, Version 3.6 or 3.7 with the PTF for APAR PK70606, or later
  - Enterprise COBOL (compiled with the TEST compiler option)
  - Assembler
  - Disassembly
- Continuous display (in the Monitor window) is useful for observing the value of a variable over time.

For continuous display, enter the following command on the command line, where *x* is the name of the variable:

```
MONITOR LIST ( x )
```

In the Monitor window, a line appears with the name of the variable and the current value of the variable next to it. If the value of the variable is undefined, the variable is not initialized, or the variable does not exist, a message appears underneath the variable name declaring the variable unusable.

Alternatively, you can enter the M prefix command in the prefix area of the Source window. In the following line from the Source window, type in M3 in the prefix area, then press Enter to add *var3* to the Monitor window:

```
200           var1 = var2 + var3;
```

z/OS Debugger creates the command `MONITOR LIST (var3)`, runs it, then adds *var3* to the Monitor window.

You can use the M prefix command only with programs assembled or compiled with the following assemblers or compilers:

- Enterprise PL/I for z/OS, Version 3.6 or 3.7 with the PTF for APAR PK70606, or later
- Enterprise COBOL (compiled with the TEST compiler option)
- Assembler
- Disassembly

- A combination of one-time and continuous display, where the value of variables coded in the current line are displayed, is useful for observing the value of variables when the variables are used.

  For a combination of one-time and continuous display, enter the following command on the command line:

  ```
  SET AUTOMONITOR ON ;
  ```

  After a line of code is run, the Monitor window displays the name and value of each variable on the line of code. The `SET AUTOMONITOR` command can be used only with specific programming languages, as described in *IBM z/OS Debugger Reference and Messages*.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Displaying values of C and C++ variables or expressions” on page 294
“Displaying values of COBOL variables” on page 270
“Displaying and monitoring the value of a variable” on page 182

Related references

“Monitor window” on page 152
Description of the MONITOR COMMAND in *IBM z/OS Debugger Reference and Messages*
Description of the SET AUTOMONITOR COMMAND in *IBM z/OS Debugger Reference and Messages*

Displaying memory through the Memory window

Sometimes it is helpful to look at memory directly in a format similar to a dump. You can use the Memory window to view memory in this format.

The Memory window is not displayed in the default screen. To display the Memory window, use the `WINDOW SWAP MEMORY LOG` command. z/OS Debugger displays the Memory window in the location of the Log window.

After you display the Memory window, you can navigate through it using the `SCROLL DOWN` and `SCROLL UP` commands. You can modify the contents of memory by typing the new values in the hexadecimal data area.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 28, “Customizing your full-screen session,” on page 251
“Displaying the Memory window” on page 169
“Displaying and modifying memory through the Memory window” on page 192
“Scrolling through the physical windows” on page 165
Related references
“z/OS Debugger session panel” on page 149
WINDOW SWAP command in *IBM z/OS Debugger Reference and Messages*

Changing the value of a variable

After you see the value of a variable, you might want to change the value. If, for example, the assigned value isn't what you expect, you can change it to the desired value. You can then continue to study the flow of your program, postponing the analysis of why the variable wasn't set correctly.

Changing the value of a variable depends on the programming language that you are debugging. In z/OS Debugger, the rules and methods for the assignment of values to variables are the same as programming language rules and methods. For example, to assign a value to a C variable, use the C assignment rules and methods:

```
var = 1 ;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Assigning values to C and C++ variables” on page 295
“Assigning values to COBOL variables” on page 269

Skipping a breakpoint

Use the DISABLE command to temporarily disable a breakpoint. Use the ENABLE command to re-enable the breakpoint.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the DISABLE command in *IBM z/OS Debugger Reference and Messages*
Description of the ENABLE command in *IBM z/OS Debugger Reference and Messages*

Clearing a breakpoint

When you no longer require a breakpoint, you can clear it. Clearing it removes any of the instructions associated with that breakpoint. For example, to clear a breakpoint on line 100 of your program, enter the following command on the command line:

```
CLEAR AT 100
```

The Log window displays a line that says `CLEAR AT 100 ;` and the prefix area reverts to its original colors. These changes indicate that the breakpoint at line 100 is gone.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the CLEAR command in *IBM z/OS Debugger Reference and Messages*

Recording and replaying statements

You can record and subsequently replay statements that you run. When you replay statements, you can replay them in a forward direction or a backward direction. Table 6 on page 18 describes the sequence in which statements are replayed when you replay them in a forward direction or a backward direction.

Table 6. The sequence in which statements are replayed.

PLAYBACK FORWARD sequence	PLAYBACK BACKWARD sequence	COBOL Statements
1	9	`DISPLAY "CALC Begins."`
2	8	`MOVE 1 TO BUFFER-PTR.`
3	7	`PERFORM ACCEPT-INPUT 2 TIMES.`
8	2	`DISPLAY "CALC Ends."`
9	1	`GOBACK.`
		`ACCEPT-INPUT.`
4, 6	4, 6	`ACCEPT INPUT-RECORD FROM A-INPUT-FILE`
5, 7	3, 5	`MOVE RECORD-HEADER TO REPROR-HEADER.`

To begin recording, enter the following command:

```
PLAYBACK ENABLE
```

Statements that you run after you enter the `PLAYBACK ENABLE` command are recorded.

To replay the statements that you record:

1. Enter the `PLAYBACK START` command.
2. To move backward one statement, enter the STEP command.
3. Repeat step 2 as many times as you can to replay another statement.
4. To move forward (from the current statement to the next statement), enter the `PLAYBACK FORWARD` command.
5. Enter the STEP command to replay another statement.
6. Repeat step 5 as many times as you want to replay another statement.
7. To move backward, enter the `PLAYBACK BACKWARD` command.

`PLAYBACK BACKWARD` and `PLAYBACK FORWARD` change the direction commands like STEP move in.

When you have finished replaying statements, enter the `PLAYBACK STOP` command. z/OS Debugger returns you to the point at which you entered the `PLAYBACK START` command. You can resume normal debugging. z/OS Debugger continues to record your statements. To replay a new set of statements, begin at step 1.

When you finish recording and replaying statements, enter the following command:

```
PLAYBACK DISABLE
```

z/OS Debugger no longer records any statements and discards information that you recorded. The `PLAYBACK START`, `PLAYBACK FORWARD`, `PLAYBACK BACKWARD`, and `PLAYBACK STOP` commands are no longer available.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

Description of the PLAYBACK commands in *IBM z/OS Debugger Reference and Messages*

Stopping z/OS Debugger

To stop your debug session, do the following steps:

1. Enter the `QUIT` command.
2. In response to the message to confirm your request to stop your debug session, press "Y" and then press Enter.

Your z/OS Debugger screen closes.

Refer to *IBM z/OS Debugger Reference and Messages* for more information about the `QQUIT`, `QUIT ABEND` and `QUIT DEBUG` commands which can stop your debug session.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

Description of the `QUIT` command in *IBM z/OS Debugger Reference and Messages*

Description of the `QQUIT` command in *IBM z/OS Debugger Reference and Messages*

Part 2. Preparing your program for debugging

© Copyright IBM Corp. 1992, 2023

Chapter 3. Preparing to remote debug in standard mode

About this task

Note: This chapter is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

To prepare to remote debug in standard mode, you must compile your program with certain compiler options.

You can specify compiler options in the following ways:

- Creating and managing property groups in the Property Group Manager (See "Resource management with property groups.")
- Specifying the options in the **COBOL and PL/I Compile Step Options** window, which generates JCL (See "COBOL and PL/I step options.")
- Editing the JCL directly.

Procedure

To add compiler options to the JCL directly, complete the following steps:

1. Open the JCL file in the Remote Systems view of the z/OS Projects perspective.
2. Specify the following options:

Table 7. Compiler options for debugging

Compiler	Required options	Recommended options
z/OS C and C++	`DEBUG` or `DEBUG(FORMAT(DWARF))`	`DEBUG(NOHOOK)` for better generated code
Enterprise COBOL for z/OS V6.2 and later	`TEST` or `TEST(SEPARATE)`[1]	
Enterprise COBOL for z/OS V5 and V6.1	`TEST`	
Enterprise COBOL for z/OS V3.4 and V4	`SOURCE,LIST,XREF,MAP,NONUM`	`NOTEST` for better performance
Enterprise PL/I for z/OS V4 and V5 (31-bit)	`TEST(NOHOOK)`	`TEST(NOHOOK,SEPARATE)` for better performance
Enterprise PL/I for z/OS V5 (64-bit)	`TEST`	
High Level Assembler V1.6	`ADATA`	

© Copyright IBM Corp. 1992, 2023

Related information

“Remote debugging in standard mode” on page 407

[1] With TEST the debug data is placed in a NOLOAD segment in the program object. With TEST(SEPARATE) the debug data is placed in a separate debug file.

Chapter 4. Planning your debug session

Before you begin debugging, create a plan that can help you make the following choices:

- The compiler or assembler options and suboptions you need to use when you compile or assemble programs.
- The debugging mode (batch, full-screen, full-screen mode using the Terminal Interface Manager, or remote debug mode) that you will use to interact with z/OS Debugger.
- The method or methods you can use to start z/OS Debugger.
- If you have older COBOL programs, as listed in the *COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM: User's Guide*, how you want to debug them.

To help you create your plan, do the following tasks:

1. Use Table 8 on page 26 to record the compiler options and suboptions that you will use for your programs. The table contains compiler options that can provide the most debugging capability with the smallest program size for a general set of compilers. See "Choosing compiler options for debugging" on page 25 for the following information:
   - The prerequisites required for a compiler option and suboption.
   - Additional tasks that you might need to do to make a compiler option and suboption work at your site.
   - Information about how a compiler option and suboption might affect program size and z/OS Debugger functionality.
   - If you are using other Application Delivery Foundation for z/OS tools, information on how to choose compiler options so that you create output that can be used by the other Application Delivery Foundation for z/OS tools.
2. Use Table 5 on page 4 to record the debugging mode you will use. See "Choosing a debugging mode" on page 49 to learn about prerequisites and tasks you must do to make the debugging mode work.
3. Use Table 14 on page 54 to record the methods you will use to specify TEST runtime options. See "Choosing a method or methods for starting z/OS Debugger" on page 53 to help you determine which method will work best for your programs.
4. If you have older COBOL programs (as listed in the *COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM: User's Guide*) that you want to debug, you must decide between the following options:
   - Leave them in their old source and possibly have to debug them as LangX COBOL programs.
   - Convert them to the 1985 COBOL Standard level.

   See "Choosing how to debug old COBOL programs" on page 56 for more information.

After you have completed these tasks, use the information you collected to follow the instructions in Chapter 5, "Updating your processes so you can debug programs with z/OS Debugger," on page 59.

Choosing compiler options for debugging

Compiler options affect the size of your load module and the amount of z/OS Debugger functionality available to you. z/OS Debugger uses information such as hooks and symbol tables to gain control of a program, run the program statement-by-statement or line-by-line, and display information about your program.

To learn more about how hooks and symbol tables help z/OS Debugger debug your program, read the following topics:

- "Understanding how hooks work and why you need them" on page 48

© Copyright IBM Corp. 1992, 2023

- "Understanding what symbol tables do and why saving them elsewhere can make your application smaller" on page 49

To learn more about how the compiler options affect z/OS Debugger functionality, read the following topics:

- "Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27
- "Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34
- "Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
- "Choosing DEBUG compiler suboptions for C programs" on page 40
- "Choosing TEST or NOTEST compiler suboptions for C programs" on page 41
- "Choosing DEBUG compiler suboptions for C++ programs" on page 45
- "Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44
- "Choosing TEST or NOTEST compiler options for C++ programs" on page 46

Table 8. Record the compiler options you need to use in this table.

Compiler or assembler	Compiler options you will use
Open Enterprise SDK for Go	No compiler option is needed. DWARF data is always produced for Go and cannot be turned off.
Enterprise COBOL for z/OS Version 6 Release 2 with APAR PH04485[1] installed or later compiled with the TEST compiler option	`TEST(EJPD,SEPARATE(DSNAME),SOURCE)` ***or***
Enterprise COBOL for z/OS Version 5 and Version 6[2] compiled with the TEST compiler option	`TEST(EJPD,SOURCE)` ***or***
Enterprise COBOL for z/OS Version 4 compiled with the TEST compiler option	`TEST(NOHOOK,SEPARATE,EJPD)` ***or***
Enterprise COBOL for z/OS Version 3 or Version 4 compiled with the NOTEST compiler option [3]	`NOTEST,NOOPTIMIZE,SOURCE,MAP,XREF,LIST`(or `OFFSET`) ***or***
Enterprise COBOL for z/OS and OS/390, Version 3	`TEST(NONE,SYM,SEPARATE)` ***or***
COBOL for OS/390 & VM	`TEST(NONE,SYM,SEPARATE)` ***or***
COBOL for MVS & VM	`TEST(ALL,SYM)` ***or***
AD/Cycle COBOL/370 Version 1 Release 1	`TEST(ALL,SYM)` ***or***
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (for programs compiled with the NOTEST compiler option and linked with the Language Environment library.) [3]	`NOTEST,NOOPTIMIZE,SOURCE,MAP,XREF,LIST`(or `OFFSET`) ***or***
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (for programs compiled with the NOTEST compiler option and linked with a non-Language Environment library.) [3]	`NOTEST,NOOPTIMIZE,SOURCE,MAP,XREF,LIST`(or `OFFSET`) ***or***
VS COBOL II Version 1 Release 3 and Version 1 Release 4 (for programs compiled with the TEST compiler option and linked with the Language Environment library.)	`TEST` ***or***
OS/VS COBOL, Version 1 Release 2.4 [3]	`NOTEST,SOURCE,DMAP,PMAP,VERB,XREF,NOLST,NOBATCH,NOSYMDMP,NOCOUNT` ***or***
Enterprise PL/I, Version 4 or Version 5[4] (31-bit)	`TEST(ALL,NOHOOK,SYM,SEPARATE)` and `LISTVIEW` and `GONUMBER(SEPARATE)` ***or***
Enterprise PL/I, Version 3.8 or later	`TEST(ALL,NOHOOK,SYM,SEPARATE)` and `LISTVIEW` ***or***
Enterprise PL/I, Version 3.7	`TEST(ALL,NOHOOK,SYM,SEPARATE,SOURCE)` ***or***

Table 8. Record the compiler options you need to use in this table. (continued)

Compiler or assembler	Compiler options you will use
Enterprise PL/I, Version 3.5 or later	`TEST(ALL,NOHOOK,SYM,SEPARATE)` ***or***
Enterprise PL/I, Version 3.4	`TEST(ALL,NOHOOK,SYM)` ***or***
Enterprise PL/I, Version 3.1 through Version 3.3	`TEST(ALL,SYM)` ***or***
PL/I for MVS & VM	`TEST(ALL,SYM)` ***or***
OS PL/I Version 2 Release 1, Version 2 Release 2, and Version 2 Release 3	`TEST(ALL,SYM)` ***or***
C/C++ feature of z/OS, Version 2.3 or later	`DEBUG(FORMAT(DWARF),NOFILE) GOFF` ***or***
C/C++ feature of z/OS, Version 1.6 or later (31-bit)	`DEBUG(FORMAT(DWARF))` ***or***
• C feature of OS/390 Version 2 Release 6 or later • C feature of z/OS, Version 1.5 or earlier	`TEST(HOOK)` ***or***
• AD/Cycle C/370 Version 1 Release 1 • C/C++ for MVS/ESA Version 3 Release 1 or later • C++ feature of OS/390 Version 2 Release 6 or later • C++ feature of z/OS, Version 1.5 or earlier	`TEST` ***or***
IBM High Level Assembler (HLASM), Version 1 Release 4, Version 1 Release 5, Version 1 Release 6[5]	`ADATA`

1. Enterprise COBOL for z/OS Version 6 Release 2 APAR PH04485: New suboptions `DSNAME|NODSNAME` are added to the `TEST|NOTEST(SEPARATE)` option to control whether the SYSDEBUG data set name used during compilation will or will not be stored in the object program.
2. Support for Enterprise COBOL for z/OS Version 6 is a superset of that for Version 5 in z/OS Debugger.
3. See Chapter 6, “Preparing a LangX COBOL program,” on page 67 for information on how to prepare a program of this type.
4. Support for Enterprise PL/I for z/OS Version 5 (31-bit) is the same as that for Version 4 in z/OS Debugger.
5. For more information, see Chapter 7, “Preparing an assembler program,” on page 71.

Choosing TEST or NOTEST compiler suboptions for COBOL programs

This topic describes the combination of TEST compiler option and suboptions you need to specify to obtain the debugging scenario. This topic assumes you are compiling your COBOL program with Enterprise COBOL for z/OS, Version 3.4, or later; however, the topics provide information about alternatives to use for older versions of the COBOL compiler.

The COBOL compiler provides the TEST compiler option and its suboptions to control the following actions:

- The generation and placement of hooks and symbol tables.
- The placement of debug information into the object file or a separate debug file.

The following instructions help you choose the combination of TEST compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:
   - Scenario A: If you are compiling with Enterprise COBOL for z/OS, Version 4, you can get the most z/OS Debugger functionality and a small program size by using `TEST(NOHOOK,SEPARATE)`. If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

     If you want to compile your program with the `OPT(STD)` or `OPT(FULL)` compiler option, you must also specify the `EJPD` suboption of the TEST compiler option to be able to do the following tasks:

     – Use the `GOTO` or `JUMPTO` commands.

  - Modify variables with predictable results.

  When you use the EJPD suboption, you might lose some optimization.

  If you are using other Application Delivery Foundation for z/OS tools, review the information in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

- Scenario B: If you are compiling with any of the following compilers, you can get the most z/OS Debugger functionality and a small program size by using TEST(NONE,SYM,SEPARATE):
  - Enterprise COBOL for z/OS and OS/390, Version 3 Release 2 or later
  - Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 with APAR PQ63235
  - COBOL for OS/390 & VM, Version 2 Release 2
  - COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ63234.

  If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

  If you want to compile your program with optimization and be able to get the most z/OS Debugger functionality, you must compile it with one of the following combination of compiler options:

  - OPT(STD) TEST(NONE,SYM)
  - OPT(STD) TEST(NONE,SYM,SEPARATE)
  - OPT(FULL) TEST(NONE,SYM)
  - OPT(FULL) TEST(NONE,SYM,SEPARATE)

  For these types of programs, you can modify variables, but the results might be unpredictable.

  If you are using other Application Delivery Foundation for z/OS tools, review the information in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

- Scenario C: To get all z/OS Debugger functionality but have a larger program size and do not want debug information in a separate debug file, compile with one of the following compiler options for the compilers specified:
  - TEST(HOOK,NOSEPARATE) with Enterprise COBOL for z/OS, Version 4.
  - TEST(ALL,SYM,NOSEPARATE) with any of the following compilers:
    - Enterprise COBOL for z/OS and OS/390, Version 3 Release 2 or later
    - Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 with APAR PQ63235
    - COBOL for OS/390 & VM, Version 2 Release 2
    - COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ40298

  If you are using other Application Delivery Foundation for z/OS tools, review the information in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

- Scenario D: If you are using COBOL for OS/390 & VM, Version 2 Release 1, or earlier, and you want to get all z/OS Debugger functionality, use TEST(ALL,SYM).

  If you are using other Application Delivery Foundation for z/OS tools, review the topic in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* that corresponds to the compiler that you are using from the following list to make sure that you specify all the compiler options that you need to create the files needed by all the Application Delivery Foundation for z/OS tools:

  - Enterprise COBOL for z/OS Version 3 and COBOL for OS/390 and VM programs

  - COBOL for MVS(tm) and VM programs
  - VS COBOL II programs
  - OS/VS COBOL programs
- Scenario E: You can get most of z/OS Debugger's functionality by compiling with the NOTEST compiler option and generating an *EQALANGX* file. This requires that you debug your program in LangX COBOL mode.
- Scenario F: You can get some z/OS Debugger's functionality by compiling with the NOTEST compiler option. This requires that you debug your program in disassembly mode.

  If you are using other Application Delivery Foundation for z/OS tools, review the topic in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* that corresponds to the compiler that you are using from the following list to make sure you specify all the compiler options that you need to create the files needed by all the Application Delivery Foundation for z/OS tools:

  - Enterprise COBOL for z/OS Version 4 programs
  - Enterprise COBOL for z/OS Version 3 and COBOL for OS/390 and VM programs
  - COBOL for MVS(tm) and VM programs
  - VS COBOL II programs
  - OS/VS COBOL programs
- Scenario G: If you are compiling with Enterprise COBOL for z/OS Version 5 or Version 6 Release 1, you can get the most z/OS Debugger functionality by using TEST(SOURCE). If you need to debug programs that are loaded into protected storage, you must verify that your site installed the Authorized Debug Facility. With the TEST(SOURCE) compiler option, the debug data is saved in the program object in a NOLOAD debug segment. The debug data does not increase the size of the loaded program. The debug data always matches the executable and is always available, so there is no need to search the lists of data sets. The size of the program object increases but not the footprint in memory, unless it is required to load the debug data while you are debugging a program.

  Note: Do not use the binder PAGE statement when you link-edit a load module that contains more than one Enterprise COBOL for z/OS Version 5 or later program that is compiled with a TEST compiler option where the debug data is saved in the program object in a NOLOAD debug segment.
- Scenario H: If you are compiling with Enterprise COBOL for z/OS Version 6 Release 2 or later, you can get the most z/OS Debugger functionality by using TEST(SOURCE) or TEST(SEPARATE,SOURCE). For Enterprise COBOL for z/OS Version 6 Release 2 with APAR PH04485 installed or later, you can specify TEST(SEPARATE(DSNAME),SOURCE) to store the separate debug file name, which is the SYSDEBUG DD data set name, in the object program. If you need to debug programs that are loaded into protected storage, you must verify that your site installed the Authorized Debug Facility.
  - With the TEST(SOURCE) compiler option, the debug data is saved in the program object in a NOLOAD debug segment. The debug data does not increase the size of the loaded program. The debug data always matches the executable and is always available, so there is no need to search the lists of data sets. The size of the program object increases but not the footprint in memory, unless it is required to load the debug data when you are debugging a program.

    Note: Do not use the binder PAGE statement when you link-edit a load module that contains more than one Enterprise COBOL for z/OS Version 5 or later program that is compiled with a TEST compiler option where the debug data is saved in the program object in a NOLOAD debug segment.
  - With the TEST(SEPARATE,SOURCE) compiler option, the debug data is saved in a separate debug file. The compiler uses the SYSDEBUG DD statement to specify the separate debug file.
    - For Enterprise COBOL for z/OS Version 6 Release 2 with APAR PH04485 installed or later, you can specify SEPARATE(DSNAME) to store the name of the separate debug file in the program object.
    - If you do not specify SEPARATE(DSNAME) or the location of the separate debug file has changed since the compilation, specify the separate debug file location with one of the following methods. z/OS Debugger looks for the separate debug file in the following order:

- **SET SOURCE** command to specify the exact location of the separate debug file
- EQAUEDAT user exit
- **SET DEFAULT LISTINGS** command
- EQADEBUG DD name
- EQA_DBG_SYSDEBUG environment variable

If you use an EQAUEDAT user exit, **SET DEFAULT LISTINGS** command, EQADEBUG DD name, or EQA_DBG_SYSDEBUG environment variable, specify a PDS data set or z/OS UNIX System Services directory as the separate debug file location.

If you use a **SET DEFAULT LISTINGS** command, EQADEBUG DD name, or EQA_DBG_SYSDEBUG environment variable, and if the separate debug file is not found because the file name does not match the CU name, z/OS Debugger will do an exhaustive search of the data sets specified by the same method to locate the matching debug file. The exhaustive search might be slow.

2. For COBOL programs using IMS, include the IMS interface module DFSLI000 from the IMS RESLIB library.
3. For scenarios A, B and E, do the following steps:
   a. If you use the Dynamic Debug facility to place hooks into programs that reside in read-only storage, verify with your system administrator that the Authorized Debug facility has been installed and that you are authorized to use it.
   b. After you start z/OS Debugger, verify that you have not deactivated the Dynamic Debug facility by entering the QUERY DYNDEBUG command.
   c. Verify that the separate debug file is a non-temporary file and is available during the debug session. The listing does not need to be saved.
4. Verify whether you need to do any of the following tasks:
   - If you specify NUMBER with TEST, make sure the sequence fields in your source code all contain numeric characters.
   - You need to specify the SYM suboption of the TEST compiler option to do the following actions:
     – To specify labels (paragraph or section names) as targets of the GOTO command.
     – To reference program variables by name.
     – To access a variable or expression through commands like LIST or DESCRIBE.
     – To use the DATA suboption of the PLAYBACK ENABLE command.

     You need to specify the SYM suboption to do these actions only if you are compiling with any of the following compilers:
     – Any release of Enterprise COBOL for z/OS and OS/390, Version 3
     – Any release of COBOL for OS/390 & VM, Version 2
   - The TEST compiler option and the DEBUG runtime option are mutually exclusive, with DEBUG taking precedence. If you specify both the WITH DEBUGGING MODE clause in your SOURCE-COMPUTER paragraph and the USE FOR DEBUGGING statement in your code, TEST is deactivated. The TEST compiler option appears in the list of options, but a diagnostic message is issued telling you that because of the conflict, TEST is not in effect.
   - For VS COBOL II programs, if you use the TEST compiler option, you must specify:
     – the SOURCE compiler option. This option is required to generate a listing file and save it at location *userid*.pgmname.list.
     – the RESIDENT compiler option. This option is required by Language Environment to ensure that the necessary z/OS Debugger routines are loaded dynamically at run time.

     In addition, you must link your program with the Language Environment SCEELKED library and not the VS COBOL II COB2LIB library.

After you have chosen the compiler options and suboptions, see Chapter 4, “Planning your debug session,” on page 25 to determine the next task you must complete.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

Description of the TEST compiler option in *Enterprise COBOL for z/OS Programming Guide*

The following table explains the effects of the NOTEST compiler option, the TEST compiler option, and some of the suboptions of the TEST compiler option on z/OS Debugger behavior or the availability of features, which are not described in *Enterprise COBOL for z/OS Programming Guide*:

Table 9. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on z/OS Debugger.

Name of compiler option or suboption	**Description of the effect**
NOTEST	• You cannot step through program statements. • You can suspend execution of the program only at the initialization of the main compile unit. • You can include calls to CEETEST in your program to allow you to suspend program execution and issue z/OS Debugger commands. • You cannot examine or use any program variables. • You can list storage and registers. • The source listing produced by the compiler cannot be used; therefore, no listing is available during a debug session. Using the SET DEFAULT LISTINGS command cannot make a listing available. • Because a statement table is not available, you cannot set any statement breakpoints or use commands such as GOTO or QUERY location. However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 35, “Debugging a disassembled program,” on page 327.

Table 9. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	Description of the effect
NONE and NOHOOK	• If you use one of the following compilers, you can use the GOTO or the JUMPTO commands when you debug a non-optimized program: – Enterprise COBOL for z/OS, Version 4 – Any release of Enterprise COBOL for z/OS and OS/390, Version 3 – Any release of COBOL for OS/390 & VM, Version 2 If you compile your program by using Enterprise COBOL for z/OS Version 4.1, you can use the GOTO or JUMPTO commands when you debug an optimized program. To enable the GOTO or JUMPTO commands, you must specify the EJPD suboption of the TEST option. When you specify the EJPD suboption, you might lose some optimization. You can use the SET WARNING OFF setting to obtain limited support for GOTO and JUMPTO when you compile with the NOEJPD suboption of the TEST compiler option. GOTO and JUMPTO are not enabled. • A call to CEETEST can be used at any point to start z/OS Debugger. • NONE and NOHOOK are not available with Enterprise COBOL for z/OS Version 5, but when you specify the TEST compile with this compiler, it creates an object similar to specifying NONE and NOHOOK with previous compilers.
EJPD	You can modify variables in an optimized program that was compiled with one the following compilers: • Enterprise COBOL for z/OS, Version 5 • Enterprise COBOL for z/OS, Version 4 • Enterprise COBOL for z/OS and OS/390, Version 3 Release 2 or later • Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 with APAR PQ63235 installed • COBOL for OS/390 & VM, Version 2 Release 2 • COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ63234 installed However, results might be unpredictable. To obtain more predictable results, compile your program with Enterprise COBOL for z/OS, Version 4 and 5, and specify the EJPD suboption of the TEST compiler option. However, variables that are declared with the VALUE clause to initialize them cannot be modified. **LOUD** The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.

Table 9. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	**Description of the effect**
NOSYM	• You cannot reference program variables by name. • You cannot use commands such as LIST or DESCRIBE to access a variable or expression. • You cannot use commands such as CALL variable to branch to another program, or GOTO to branch to another label (paragraph or section name). If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler ignores SYM or NOSYM and always creates a symbol table. This option is not available with Enterprise COBOL for z/OS Version 5.
STMT	• The COBOL compiler generates compiled-in hooks for date processing statements only when the DATEPROC compiler option is specified. A date processing statement is any statement that references a date field, or any EVALUATE or SEARCH statement WHEN phrase that references a date field. • You can set breakpoints at all statements and step through your program. • z/OS Debugger cannot gain control at path points unless they are also at statement boundaries. • Branching to all statements and labels using the z/OS Debugger command GOTO is allowed. If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the STMT suboption as if it were the HOOK suboption, which is equivalent to the ALL suboption for any release of Enterprise COBOL for z/OS and OS/390, Version 3, or COBOL for OS/390 & VM, Version 2. This option is not available with Enterprise COBOL for z/OS Version 5.
PATH	• z/OS Debugger can gain control only at path points and block entry and exit points. If you attempt to step through your program, z/OS Debugger gains control only at statements that coincide with path points, giving the appearance that not all statements are executed. • A call to CEETEST can be used at any point to start z/OS Debugger. • The z/OS Debugger command GOTO is valid for all statements and labels coinciding with path points. If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the PATH suboption as if it were the HOOK suboption, which is equivalent to the ALL suboption for any release of Enterprise COBOL for z/OS and OS/390, Version 3, or COBOL for OS/390 & VM, Version 2. This option is not available with Enterprise COBOL for z/OS Version 5.

Table 9. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	Description of the effect
BLOCK	• z/OS Debugger gains control at entry and exit of your program, methods, and nested programs. • z/OS Debugger can be explicitly started at any point with a call to CEETEST. • Issuing a command such as STEP causes your program to run until it reaches the next entry or exit point. • G0T0 can be used to branch to statements that coincide with block entry and exit points. If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the BL0CK suboption as if it were the H00K suboption, which is equivalent to the ALL suboption for any release of Enterprise COBOL for z/OS and OS/390, Version 3, or COBOL for OS/390 & VM, Version 2. This option is not available with Enterprise COBOL for z/OS Version 5.
ALL	• You can set breakpoints at all statements and path points, and step through your program. • z/OS Debugger can gain control of the program at all statements, path points, date processing statements, labels, and block entry and exit points, allowing you to enter z/OS Debugger commands. • Branching to statements and labels using the z/OS Debugger command G0T0 is allowed. If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the ALL suboption as if it were the H00K suboption, which is equivalent to the ALL suboption for any release of Enterprise COBOL for z/OS and OS/390, Version 3, or COBOL for OS/390 & VM, Version 2. This option is not available with Enterprise COBOL for z/OS Version 5, but when you specify the TEST compile with this compiler, it creates an object similar to specifying ALL with the exception that compiled-in hooks are not available.

Choosing TEST or NOTEST compiler suboptions for PL/I programs

This topic describes the combination of TEST compiler option and suboptions you need to specify to obtain the desired debugging scenario. This topic assumes you are compiling your PL/I program with Enterprise PL/I for z/OS, Version 3.5, or later; however, the topics provide information about alternatives to use for older versions of the PL/I compiler.

The PL/I compiler provides the TEST compiler option and its suboptions to control the following actions:

- The generation and placement of hooks and symbol tables.
- The placement of debug information into the object file or separate debug file.

z/OS Debugger does not support debugging optimized PL/I programs. Do not use compiler options other than NOOPTIMIZE,

The following instructions help you choose the combination of TEST compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:

   - Scenario A: If you are using Enterprise PL/I for z/OS, Version 3.8 or later, and you want to get the most z/OS Debugger functionality and a small program size, use `TEST(ALL,NOHOOK,SYM,SEPARATE)` and the `LISTVIEW(SOURCE)` compiler option. If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

     Consider the following options:

     – If you are using Enterprise PL/I for z/OS, Version 4 or later, you can specify the `GONUMBER(SEPARATE)` compiler option, which can help make the program size smaller.

     – You can specify any of the `LISTVIEW` sub-options (`SOURCE`, `AFTERALL`, `AFTERCICS`, `AFTERMACRO`, or `AFTERSQL`), as described in *Enterprise PL/I for z/OS Programming Guide*, to display either the original source or the source after the specified preprocessor.

     – If you are debugging in full-screen mode and you want to debug programs with INCLUDE files that have executable code, specify the `LISTVIEW(AFTERMACRO)` compiler option and, if you do not specify the `MACRO` compiler option, specify the `PP(MACRO(INCONLY))` compiler option.

     – If you are debugging in remote debug mode and you want to automonitor variables in INCLUDE files, specify the `LISTVIEW(AFTERMACRO)` compiler option and, if you do not specify the `MACRO` compiler option, specify the `PP(MACRO(INCONLY))` compiler option.

     If you are using other Application Delivery Foundation for z/OS tools, see topic *Enterprise PL/I Version 3.5 and Version 3.6 programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

   - Scenario B: If you are using Enterprise PL/I for z/OS, Version 3.7, and you want to get the most z/OS Debugger functionality and a small program size, use `TEST(ALL,NOHOOK,SYM,SEPARATE,SOURCE)`. If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

     Consider the following options:

     – You can substitute `SOURCE` with `AFTERALL`, `AFTERCICS`, `AFTERMACRO`, or `AFTERSQL`, as described in *Enterprise PL/I for z/OS Programming Guide*.

     – If you are debugging in full-screen mode and you want to debug programs with INCLUDE files that have executable code, specify the `TEST(ALL,NOHOOK,SYM,SEPARATE,AFTERMACRO)` compiler options and, if you do not specify the `MACRO` compiler option, specify the `PP(MACRO(INCONLY))` compiler option.

     – If you are debugging in remote debug mode and you want to automonitor variables in INCLUDE files, specify the `TEST(ALL,NOHOOK,SYM,SEPARATE,AFTERMACRO)` compiler options and, if you do not specify the `MACRO` compiler option, specify the `PP(MACRO(INCONLY))` compiler option.

     If you are using other Application Delivery Foundation for z/OS tools, see topic *Enterprise PL/I Version 3.5 and Version 3.6 programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

   - Scenario C: If you are using Enterprise PL/I for z/OS, Version 3.5 or 3.6, and you want to get most z/OS Debugger functionality and a small program size, use `TEST(ALL,NOHOOK,SYM,SEPARATE)`. If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

     If you are using other Application Delivery Foundation for z/OS tools, see topic *Enterprise PL/I Version 3.5 and Version 3.6 programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

- Scenario D: If you are using Enterprise PL/I for z/OS, Version 3.4, and you want to debug your program without compiled-in hooks, use TEST(ALL,NOHOOK,SYM). If you need to debug programs that are loaded into protected storage, verify that your site installed the Authorized Debug Facility.

  If you are using other Application Delivery Foundation for z/OS tools, see topic *Enterprise PL/I Version 3.4 and earlier programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.
- Scenario E: If you are using Enterprise PL/I for z/OS, Version 3.3 or earlier, and you want to get all z/OS Debugger functionality, use TEST(ALL,SYM).

  If you are using other Application Delivery Foundation for z/OS tools, see topic *Enterprise PL/I Version 3.4 and earlier programs* or *PL/I for MVS(tm) and VM and OS PL/I programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.
- Scenario F: You can get some z/OS Debugger functionality by compiling with the NOTEST compiler option. This requires that you debug your program in disassembly mode.

  If you are using other Application Delivery Foundation for z/OS tools, review the topic in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* that corresponds to the compiler that you are using from the following list to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools:

  – Enterprise PL/I Version 3.5 and Version 3.6 programs
  – Enterprise PL/I Version 3.4 and earlier programs
  – PL/I for MVS(tm) and VM and OS PL/I programs

2. For scenarios A, B, C, E, and F, do the following steps:

   a. If you use the Dynamic Debug facility to place hooks into programs that reside in read-only storage, verify with your system administrator that the Authorized Debug facility has been installed and that you are authorized to use it.

   b. After you start z/OS Debugger, verify that you have not deactivated the Dynamic Debug facility by entering the QUERY DYNDEBUG command.

   c. Verify that the separate debug file is a non-temporary file and is available during the debug session.

3. Verify whether you need to do any of the following tasks:

   - When you compile a program, do not associate SYSIN with an in-stream data set (for example // SYSIN DD *) because z/OS Debugger requires access to a permanent data set for the source of the program you are debugging.
   - If you are compiling a PL/I for MVS & VM or OS PL/I program and to be able to view your listing while debugging in full-screen mode, you must compile the program with the SOURCE compiler option. The SOURCE compiler option is required to generate a listing file. You must direct the listing to a non-temporary file that is available during the debug session. During a debug session, z/OS Debugger displays the first file it finds named `userid.pgmname.list` in the Source window. In addition, you must link your program with the Language Environment SCEELKED library; do not use the OS PL/I PLIBASE or SIBMBASE library.

     If z/OS Debugger cannot find the listing at this location, see "Changing which file appears in the Source window" on page 157.

After you have chosen the compiler options and suboptions, see Chapter 4, "Planning your debug session," on page 25 to determine the next task you must complete.

Table 10. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on z/OS Debugger.

Name of compiler option or suboption	Description of the effect
NOTEST	Some behaviors or features change when you debug a PL/I program compiled with the NOTEST compiler option. The following list describes these changes: • You can list storage and registers. • You can include calls to PLITEST or CEETEST in your program so you can suspend running your program and issue z/OS Debugger commands. • You cannot step through program statements. You can suspend running your program only at the initialization of the main compile unit. • You cannot examine or use any program variables. • Because hooks at the statement level are not inserted, you cannot set any statement breakpoints or use commands such as GOTO or QUERY LOCATION. • The source listing produced by the compiler cannot be used; therefore, no listing is available during a debug session. However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 35, "Debugging a disassembled program," on page 327.
NOHOOK	Some behaviors or features change when you debug a PL/I program compiled with the NOHOOK suboption of the TEST compiler option. The following list describes these changes: • For z/OS Debugger to generate overlay hooks, one of the suboptions ALL, PATH, STMT or BLOCK must be in effect, but HOOK need not be specified, and NOHOOK would be recommended. • If NOHOOK is specified, ENTRY and EXIT breakpoints are the only PATH breakpoints at which z/OS Debugger stops.
NONE	When you compile a PL/I program with the NONE suboption of the TEST compiler option, you can start z/OS Debugger at any point in your program by writing a call to PLITEST or CEETEST in your program.
SYM	Some behaviors or features change when you debug a PL/I program compiled with the SYM suboption of the TEST compiler option. The following list describes these changes: • You can reference all program variables by name, which allows you to examine them or use them in expressions and use the DATA parameter of the PLAYBACK ENABLE command. • Enables support for the SET AUTOMONITOR ON command. • Enables the support for labels as GOTO targets.

Table 10. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	**Description of the effect**
NOSYM	Some behaviors or features change when you debug a PL/I program compiled with the NOSYM suboption of the TEST compiler option. The following list describes these changes: • You cannot reference program variables by name. • You cannot use commands such as LIST or DESCRIBE to access a variable or expression. • You cannot use commands such as CALL variable to branch to another program, or GOTO to branch to another label (procedure or block name).
BLOCK	Some behaviors or features change when you debug a PL/I program compiled with the BLOCK suboption of the TEST compiler option. The following list describes these changes: • Enables z/OS Debugger to gain control at block boundaries: block entry and block exit. • When Dynamic Debug is not active and you use the HOOK compiler option, you can gain control only at the entry and exit points of your program and all entry and exit points of internal program blocks. When you enter the STEP command, for example, your program runs until it reaches the next block entry or exit point. • When Dynamic Debug is active, you can set breakpoints at all statements and step through your program. • You cannot gain control at path points unless you also specify PATH. • A call to PLITEST or CEETEST can be used to start z/OS Debugger at any point in your program. • Hooks are not inserted into an empty ON-unit or an ON-unit consisting of a single GOTO statement.
STMT	Some behaviors or features change when you debug a PL/I program compiled with the STMT suboption of the TEST compiler option. The following list describes these changes: • You can set breakpoints at all statements and step through your program. • z/OS Debugger cannot gain control at path points unless they are also at statement boundaries, unless you also specify PATH. • Branching to all statements and labels using the z/OS Debugger command GOTO is allowed.

Table 10. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	Description of the effect
ALL	Some behaviors or features change when you debug a PL/I program compiled with the ALL suboption of the TEST compiler option. The following list describes these changes: • You can set breakpoints at all statements and path points, and STEP through your program. • z/OS Debugger can gain control of the program at all statements, path points, labels, and block entry and exit points, allowing you to enter z/OS Debugger commands. • Enables branching to statements and labels using the z/OS Debugger command GOTO.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the TEST compiler option in *Enterprise PL/I for z/OS Programming Guide*.

Choosing TEST or DEBUG compiler suboptions for C programs

This topic describes the combination of TEST or DEBUG compiler options and suboptions you need to specify to obtain the desired debugging scenario. This topic assumes you are compiling your C program with z/OS C/C++, Version 1.6, or later; however, the topics provide information about alternatives to use for older versions of the C compiler.

Choosing between TEST and DEBUG compiler options

If you are compiling with z/OS C/C++, Version 1.5 or earlier, you must choose the TEST compiler option.

The C/C++ compiler option DEBUG was introduced with z/OS C/C++ Version 1.5. z/OS Debugger supports the DEBUG compiler option in z/OS C/C++ Version 1.6 or later. The DEBUG compiler option replaces the TEST compiler option that was available with previous versions of the compiler.

If you are compiling with z/OS C/C++, Version 1.6 or later, choose the DEBUG compiler option and take advantage of the following benefits:

- For C++ programs, you can specify the HOOK(NOBLOCK) compiler option, which can improve debug performance.
- For C and C++ programs, if you specify the FORMAT(DWARF) suboption of the DEBUG compiler option, the load modules are smaller; however, you must save the .dbg file in addition to the source file. z/OS Debugger needs both of these files to debug your program.
- For C and C++ programs compiled with z/OS XL C/C++, Version 1.10 or later, if you specify the FORMAT(DWARF) suboption of the DEBUG compiler option, the load modules are smaller and you can create .mdbg files with captured source. z/OS Debugger needs only the .mdbg file to debug your program.
- For C and C++ programs compiled with z/OS XL C/C++, Version 2.3 or later, if you specify the FORMAT(DWARF) and NOFILE suboptions of the DEBUG compiler option, along with the compiler option GOFF, the program objects are larger but you do not need to save the .dbg file. z/OS Debugger needs only the source file to debug your program.

Choosing DEBUG compiler suboptions for C programs

This topic describes the debugging scenarios available, and how to create a particular debugging scenario by choosing the correct DEBUG compiler suboptions.

The C compiler provides the DEBUG compiler option and its suboptions to control the following actions:

- The generation and placement of hooks and symbol tables.
- The placement of debug information into the object file or separate debug file.

z/OS Debugger does not support debugging optimized C programs. Do not use any OPTIMIZE compiler options other than NOOPTIMIZE or OPTIMIZE(0).

The following instructions help you choose the combination of DEBUG compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:
   - Scenario A: To get the most z/OS Debugger functionality, a smaller program size, and better performance, use one of the following combinations:

     ```
     DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL,FILE(file_location))
     ```

     The compiler options are the same whether you use only .dbg files or also use .mdbg files.
   - Scenario B: To get all z/OS Debugger functionality but have a larger program size and do not want the debug information in a separate file, use the following combination:

     ```
     DEBUG(FORMAT(ISD),HOOK(LINE,NOBLOCK,PATH),SYMBOL)
     ```
   - Scenario C: You can get some z/OS Debugger functionality by compiling with the NODEBUG compiler option. This requires that you debug your program in disassembly mode.
   - Scenario D: If you are compiling with z/OS C/C++ Version 2.3 or later, use the following combination to get the most z/OS Debugger functionality with no separate file for the debug information:

     ```
     DEBUG(FORMAT(DWARF),NOFILE,HOOK(LINE,NOBLOCK,PATH),SYMBOL) GOFF
     ```

     The debug data does not increase the size of the loaded program. The size of the program object increases but not the footprint in memory, unless it is required to load the debug data when you are debugging a program. The debug data always matches the executable and is always available, so there is no need to search the lists of data sets.

   For all scenarios, if you are using Application Delivery Foundation for z/OS tools, see topic *z/OS XL C and C++ programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.
2. For the scenario you selected, verify that you have the following resources:
   - For scenario A, do the following tasks:
     - If you create an .mdbg file, do the following tasks:
       a. Specify YES for the EQAOPTS MDBG command (which requires z/OS Debugger to search for a .dbg file in a .mdbg file)[2].
       b. Verify that the .dbg files are non-temporary files.
       c. Create the .mdbg file with captured source by using the -c option for the dbgld command or the CAPSRC option on the CDADBGLD utility.
       d. Verify that the .mdbg file is a non-temporary file.

[2] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

- If you use only .dbg files, verify that the .dbg files are non-temporary files and specify NO for the EQAOPTS MDBG command[3].

- For scenario C, do the following steps:

  a. If you use the Dynamic Debug facility to place hooks into programs that reside in read-only storage, verify with your system administrator that the Authorized Debug facility has been installed and that you are authorized to use it.

  b. After you start z/OS Debugger, verify that you have not deactivated the Dynamic Debug facility by entering the QUERY DYNDEBUG command.

3. Verify whether you need to do any of the following tasks:

   - You can specify any combination of the C DEBUG suboptions in any order. The default suboptions are BLOCK, LINE, PATH, and SYMBOL.
   - When you compile a program, do not associate SYSIN with an in-stream data set (for example // SYSIN DD *) because z/OS Debugger requires access to a permanent data set for the source of the program you are debugging.
   - z/OS Debugger does not support the LP64 compiler option. You must specify or have in effect the ILP32 compiler option.
   - If you specify the OPTIMIZE compiler option with a level higher than 0, then no hooks are generated for line, block or path points, and no symbol table is generated. Only hooks for function entry and exit points are generated for optimized programs. The TEST compiler option has the same restriction.
   - You cannot call user-defined functions from the command line.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the DEBUG compiler option in *z/OS XL C/C++ User's Guide*

Choosing TEST or NOTEST compiler suboptions for C programs

This topic describes the debugging scenarios available, and how to create a particular debugging scenario by choosing the correct TEST compiler suboptions.

The C compiler provides the TEST compiler option and its suboptions to control the generation and placement of hooks and symbol tables.

z/OS Debugger does not support debugging optimized C programs. Do not use compiler options other than NOOPTIMIZE,

The following instructions help you choose the combination of TEST compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:

   - Scenario A: To get all z/OS Debugger functionality but have a larger program size (compared to using DEBUG(FORMAT(DWARF))), use TEST(ALL,HOOK,SYMBOL).
   - Scenario B: You can get some z/OS Debugger functionality by compiling with the NOTEST compiler option. This requires that you debug your program in disassembly mode.
   - Scenario C: If you are debugging programs running in ALCS, you must compile with the HOOK suboption of the TEST compiler option.

   For all scenarios, if you are using other Application Delivery Foundation for z/OS tools, see topic *z/OS XL C and C++ programs* in *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

[3] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

2. For scenario B, do the following steps:
   a. If you use the Dynamic Debug facility to place hooks into programs that reside in read-only storage, verify with your system administrator that the Authorized Debug facility has been installed and that you are authorized to use it.
   b. After you start z/OS Debugger, verify that you have not deactivated the Dynamic Debug facility by entering the `SET DYNDEBUG OFF` command.
3. Verify whether you need to do any of the following tasks:
   - When you compile a program, do not associate SYSIN with an in-stream data set (for example // SYSIN DD *) because z/OS Debugger requires access to a permanent data set for the source of the program you are debugging.
   - If you are using `#pragma` statements to specify your TEST or NOTEST compiler options, see "Compiling your C program with the #pragma statement" on page 43.
   - The C TEST compiler option implicitly specifies the `GONUMBER` compiler option, which causes the compiler to generate line number tables that correspond to the input source file. You can explicitly remove this option by specifying `NOGONUMBER`. When the TEST and `NOGONUMBER` options are specified together, z/OS Debugger does not display the current execution line as you step through your code.
   - Programs that are compiled with both the TEST compiler option and either the `OPT(1)` or `OPT(2)` compiler option do not have hooks at line, block, and path points, or generate a symbol table, regardless of the TEST suboptions specified. Only hooks for function entry and exit points are generated for optimized programs.
   - You can specify any number of TEST suboptions, including conflicting suboptions (for example, both `PATH` and `NOPATH`). The last suboptions that are specified take effect. For example, if you specify `TEST(BLOCK, NOBLOCK, BLOCK, NOLINE, LINE)`, what takes effect is `TEST(BLOCK, LINE)` because `BLOCK` and `LINE` are specified last.
   - No duplicate hooks are generated even if two similar TEST suboptions are specified. For example, if you specify `TEST(BLOCK, PATH)`, the `BLOCK` suboption causes the generation of hooks at entry and exit points. The `PATH` suboption also causes the generation of hooks at entry and exit points. However, only one hook is generated at each entry and exit point.

Table 11. Description of the effects that the C NOTEST compiler option and the TEST compiler suboptions have on z/OS Debugger.

Name of compiler option or suboption	**Description of the effect**
NOTEST	The following list explains the effect the NOTEST compiler option will have on how z/OS Debugger behaves or the availability of features, which are not described in *z/OS XL C/C++ User's Guide*: • You cannot step through program statements. You can suspend execution of the program only at the initialization of the main compile unit. • You cannot examine or use any program variables. • You can list storage and registers. • You cannot use the z/OS Debugger command `GOTO`. However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 35, "Debugging a disassembled program," on page 327.

Table 11. Description of the effects that the C NOTEST compiler option and the TEST compiler suboptions have on z/OS Debugger. (continued)

Name of compiler option or suboption	Description of the effect
TEST	The following list explains the effect some of the suboptions of the TEST compiler option will have on how z/OS Debugger behaves or the availability of features, which are not described in *z/OS XL C/C++ User's Guide*: • The maximum number of lines in a single source file cannot exceed 131,072. • The maximum number of include files that have executable statements cannot exceed 1024.
NOSYM	The following list explains the effect the NOSYM suboption of the TEST compiler option will have on how z/OS Debugger behaves or the availability of features, which are not described in *z/OS XL C/C++ User's Guide*. • You cannot reference program variables by name. • You cannot use commands such as LIST or DESCRIBE to access a variable or expression. • You cannot use commands such as CALL or GOTO to branch to another label (paragraph or section name).

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the TEST compiler option in *z/OS XL C/C++ User's Guide*

Compiling your C program with the #pragma statement

The TEST/NOTEST compiler option can be specified either when you compile your program or directly in your program, using a #pragma.

This #pragma must appear before any executable code in your program.

The following example generates symbol table information, symbol information for nested blocks, and hooks at line numbers:

```
#pragma options (test(SYM,BLOCK,LINE))
```

This is equivalent to TEST(SYM,BLOCK,LINE,PATH).

You can also use a #pragma to specify runtime options.

Delay debug mode for C requires the FUNCEVENT(ENTRYCALL) compiler suboption

You must specify the FUNCEVENT(ENTRYCALL) compiler option when you compile your programs for delay debug usage.

Usage notes:

- The FUNCEVENT(ENTRYCALL) compiler option is available in the z/OS 2.1 XL C/C++ compiler with the PTF for APAR PI19326 applied.

- If your C application runs on UNIX System Services with imported functions from a DLL module and you want to delay the starting of a debug session until one of those functions is called, the DLL module name must be the same as the load library name.

Rules for the placement of hooks in functions and nested blocks

The following rules apply to the placement of hooks for getting in and out of functions and nested blocks:

- The hook for function entry is placed before any initialization or statements for the function.
- The hook for function exit is placed just before actual function return.
- The hook for nested block entry is placed before any statements or initialization for the block.
- The hook for nested block exit is placed after all statements for the block.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS XL C/C++ User's Guide

Rules for placement of hooks in statements and path points

The following rules apply to the placement of hooks for statements and path points:

- Label hooks are placed before the code and all other statement or path point hooks for the statement.
- The statement hook is placed before the code and path point hook for the statement.
- A path point hook for a statement is placed before the code for the statement.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS XL C/C++ User's Guide

Choosing TEST or DEBUG compiler suboptions for C++ programs

This topic describes the combination of TEST or DEBUG compiler options and suboptions you need to specify to obtain the desired debugging scenario. This topic assumes you are compiling your C++ program with z/OS C/C++, Version 1.6, or later; however, the topics provide information about alternatives to use for older versions of the C++ compiler.

Choosing between TEST and DEBUG compiler options

If you are compiling with z/OS C/C++, Version 1.5 or earlier, you must choose the TEST compiler option.

The C/C++ compiler option DEBUG was introduced with z/OS C/C++ Version 1.5. z/OS Debugger supports the DEBUG compiler option in z/OS C/C++ Version 1.6 or later. The DEBUG compiler option replaces the TEST compiler option that was available with previous versions of the compiler.

If you are compiling with z/OS C/C++, Version 1.6 or later, choose the DEBUG compiler option and take advantage of the following benefits:

- For C++ programs, you can specify the HOOK(NOBLOCK) compiler option, which can improve debug performance.
- For C and C++ programs, if you specify the FORMAT(DWARF) suboption of the DEBUG compiler option, the load modules are smaller; however, you must save the .dbg file in addition to the source file. z/OS Debugger needs both of these files to debug your program.
- For C and C++ programs compiled with z/OS XL C/C++, Version 1.10 or later, if you specify the FORMAT(DWARF) suboption of the DEBUG compiler option, the load modules are smaller and you can create .mdbg files with captured source. z/OS Debugger needs only the .mdbg file to debug your program.
- For C and C++ programs compiled with z/OS XL C/C++, Version 2.3 or later, if you specify the FORMAT(DWARF) and NOFILE suboptions of the DEBUG compiler option, along with the compiler option

GOFF, the program objects are larger but you do not need to save the .dbg file. z/OS Debugger needs only the source file to debug your program.

Choosing DEBUG compiler suboptions for C++ programs

This topic describes the debugging scenarios available, and how to create a particular debugging scenario by choosing the correct DEBUG compiler suboptions.

The C++ compiler provides the DEBUG compiler option and its suboptions to control the following actions:

- The generation and placement of hooks and symbol tables.
- The placement of debug information into the object file or separate debug file.

z/OS Debugger does not support debugging optimized C programs. Do not use any OPTIMIZE compiler options other than NOOPTIMIZE or OPTIMIZE(0).

The following instructions help you choose the combination of DEBUG compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:
   - Scenario A: To get the most z/OS Debugger functionality, a smaller program size, and better performance, use one of the following combinations:

     ```
     DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL,FILE(file_location))
     ```

     The compiler options are the same whether you use only .dbg files or also use .mdbg files.
   - Scenario B: To get all z/OS Debugger functionality but have a larger program size and do not want the debug information in a separate file, use the following combination:

     ```
     DEBUG(FORMAT(ISD),HOOK(LINE,NOBLOCK,PATH),SYMBOL)
     ```
   - Scenario C: You can get some z/OS Debugger functionality by compiling with the NODEBUG compiler option. This requires that you debug your program in disassembly mode.
   - Scenario D: If you are compiling with z/OS C/C++ Version 2.3 or later, use the following combination to get the most z/OS Debugger functionality with no separate file for the debug information:

     ```
     DEBUG(FORMAT(DWARF),NOFILE,HOOK(LINE,NOBLOCK,PATH),SYMBOL) GOFF
     ```

     The debug data does not increase the size of the loaded program. The size of the program object increases but not the footprint in memory, unless it is required to load the debug data when you are debugging a program. The debug data always matches the executable and is always available, so there is no need to search the lists of data sets.

   For all scenarios, if you are using other Application Delivery Foundation for z/OS tools, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.
2. For the scenario you selected, verify that you have the following resources:
   - For scenario A, do the following tasks:
     - If you create an .mdbg file, do the following tasks:
       a. Specify YES for the EQAOPTS MDBG command (which requires z/OS Debugger to search for a .dbg file in a .mdbg file)[4].
       b. Verify that the .dbg files are non-temporary files.

[4] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

c. Create the .mdbg file with captured source by using the -c option for the dbgld command or the CAPSRC option on the CDADBGLD utility.

d. Verify that the .mdbg file is a non-temporary file.

– If you use only .dbg files, verify that the .dbg files are non-temporary files and specify NO for the EQAOPTS MDBG command[5].

- For scenario C, do the following steps:

a. If you use the Dynamic Debug facility to place hooks into programs that reside in read-only storage, verify with your system administrator that you are authorized to do so

b. After you start z/OS Debugger, verify that you have not deactivated the Dynamic Debug facility by entering the QUERY DYNDEBUG command.

3. Verify whether you need to do any of the following tasks:

- You can specify any combination of the C++ DEBUG suboptions in any order. The default suboptions are BLOCK, LINE, PATH, and SYM.
- When you compile a program, do not associate SYSIN with an in-stream data set (for example // SYSIN DD *) because z/OS Debugger requires access to a permanent data set for the source of the program you are debugging.
- z/OS Debugger does not support the LP64 compiler option. You must specify or have in effect the ILP32 compiler option.
- If you specify the OPTIMIZE compiler option with a level higher than 0, then no hooks are generated for line, block or path points, and no symbol table is generated. Only hooks for function entry and exit points are generated for optimized programs. The TEST compiler option has the same restriction.
- You cannot call user defined functions from the command line.

After you have chosen the compiler options and suboptions, see Chapter 4, "Planning your debug session," on page 25 to determine the next task you must complete.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the DEBUG compiler option in *z/OS XL C/C++ User's Guide*

Choosing TEST or NOTEST compiler options for C++ programs

This topic describes the debugging scenarios available, and how to create a particular debugging scenario by choosing the correct TEST compiler suboptions.

The C++ compiler provides the TEST compiler option and its suboptions to control the generation and placement of hooks and symbol tables.

z/OS Debugger does not support debugging optimized C++ programs. Do not use compiler options other than NOOPTIMIZE,

The following instructions help you choose the combination of TEST compiler suboptions that provide the functionality you need to debug your program:

1. Choose a debugging scenario, keeping in mind your site's resources, from the following list:

- Scenario A: To get all z/OS Debugger functionality but have a larger program size (compared to using DEBUG(FORMAT(DWARF))), use TEST.
- Scenario B: You can get some z/OS Debugger functionality by compiling with the NOTEST compiler option. This requires that you debug your program in disassembly mode.
- Scenario C: If you are debugging programs running in ALCS, you must compile with the HOOK suboption of the TEST compiler option.

[5] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

For all scenarios, if you are using other Application Delivery Foundation for z/OS tools, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

2. Verify whether you need to do any of the following tasks:
   - When you compile a program, do not associate SYSIN with an in-stream data set (for example // SYSIN DD *) because z/OS Debugger requires access to a permanent data set for the source of the program you are debugging.
   - The C++ TEST compiler option implicitly specifies the GONUMBER compiler option, which causes the compiler to generate line number tables that correspond to the input source file. You can explicitly remove this option by specifying NOGONUMBER. When the TEST and NOGONUMBER options are specified together, z/OS Debugger does not display the current execution line as you step through your code.
   - Programs that are compiled with both the TEST compiler option and either the OPT(1) or OPT(2) compiler option do not have hooks at line, block, and path points, or generate a symbol table. Only hooks for function entry and exit points are generated for optimized programs.

After you have chosen the compiler options and suboptions, see Chapter 4, "Planning your debug session," on page 25 to determine the next task you must complete.

Table 12. Description of the effects that the C++ NOTEST and TEST compiler option have on z/OS Debugger.

Name of compiler option or suboption	**Description of the effect**
NOTEST	The following list explains the effect of the NOTEST compiler has on z/OS Debugger behavior, which are not described in *z/OS XL C/C++ User's Guide*: • You cannot step through program statements. You can suspend execution of the program only at the initialization of the main compile unit. • You cannot examine or use any program variables. • You can list storage and registers. • You cannot use the z/OS Debugger command GOTO. However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 35, "Debugging a disassembled program," on page 327.
TEST	The following list explains the effect the TEST compiler has on z/OS Debugger behavior, which are not described in *z/OS XL C/C++ User's Guide*: • The maximum number of lines in a single source file cannot exceed 131,072. • The maximum number of include files that have executable statements cannot exceed 1024.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Description of the TEST compiler option in *z/OS XL C/C++ User's Guide*

Rules for the placement of hooks in functions and nested blocks

The following rules apply to the placement of hooks for functions and nested blocks:

- The hook for function entry is placed before any initialization or statements for the function.
- The hook for function exit is placed just before actual function return.
- The hook for nested block entry is placed before any statements or initialization for the block.
- The hook for nested block exit is placed after all statements for the block.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS XL C/C++ User's Guide

Rules for the placement of hooks in statements and path points

The following rules apply to the placement of hooks for statements and path points:

- Label hooks are placed before the code and all other statement or path point hooks for the statement.
- The statement hook is placed before the code and path point hook for the statement.
- A path point hook for a statement is placed before the code for the statement.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS XL C/C++ User's Guide

Understanding how hooks work and why you need them

Hooks enable you to set breakpoints. Hooks are instructions that can be inserted into a program by a compiler at compile time. Hooks can be placed at the entrances and exits of blocks, at statement boundaries, and at points in the program where program flow might change between statement boundaries (called path points). If you compile a program with the TEST compiler option and specify any suboption except NONE or NOHOOK, the compiler inserts hooks into your program (except for Enterprise COBOL for z/OS Version 5, which never generates compiled in hooks).

How the Dynamic Debug facility can help you get maximum performance without hooks

In the following situations, you can compile or create a program without hooks. Then, you can use the Dynamic Debug facility to insert hooks at runtime whenever you set a breakpoint or enter the STEP command:

- Assembler, disassembly, and LangX COBOL programs do not contain hooks.
- Enterprise COBOL for z/OS Version 5 always generates programs without hooks.
- If you use Enterprise COBOL for z/OS, Version 4, you can compile your programs without hooks by using the TEST(NOHOOK) compiler option.
- If you use one of the following compilers, you can compile your programs without hooks by using the TEST(NONE) compiler option:
  - Enterprise COBOL for z/OS and OS/390, Version 3
  - COBOL for OS/390 & VM, Version 2 Release 2
  - COBOL for OS/390 & VM, Version 2 Release 1, with APAR PQ40298
- If you use the Enterprise PL/I for z/OS, Version 3.4 or later, compiler, you can compile your programs without hooks by using the TEST(NOHOOK) compiler option.

The Dynamic Debug facility can also help improve the performance of z/OS Debugger while debugging programs compiled with any of the following compilers:

- any COBOL compiler supported by z/OS Debugger
- any PL/I compiler supported by z/OS Debugger
- any C/C++ compiler supported by z/OS Debugger

When you compile with one the following compilers and have the compiler insert hooks, you can enhance the program's performance while you debug it by using the Dynamic Debug facility:

- any COBOL compiler supported by z/OS Debugger
- any PL/I compiler supported by z/OS Debugger
- any C/C++ compiler supported by z/OS Debugger

When you start z/OS Debugger, the Dynamic Debug facility is activated unless you change the default by using the DYNDEBUG EQAOPTS command. If the DYNDEBUG EQAOPTS command was used to change the default to DYNDEBUG OFF, you can activate it by using the SET DYNDEBUG ON z/OS Debugger command. Note that the SET DYNDEBUG ON z/OS Debugger command must be issued before you enter the STEP or GO command. If the Dynamic Debug facility is not active, z/OS Debugger uses the hooks inserted by the compiler, instead of the hooks inserted by the Dynamic Debug facility.

Understanding what symbol tables do and why saving them elsewhere can make your application smaller

The symbol table contains descriptions of variables, their attributes, and their location in storage. z/OS Debugger uses these descriptions when it references variables. The symbol tables can be stored in the object file of the program or in a separate debug file. You can save symbol tables in a separate debug file if you compile or assemble your programs with one of the following compilers or assemble:

- Enterprise COBOL for z/OS, Version 4
- Enterprise COBOL for z/OS and OS/390, Version 3
- COBOL for OS/390 & VM, Version 2 Release 2
- COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ40298
- OS/VS COBOL Version 1, Release 2.4
- Enterprise PL/I for z/OS, Version 3 Release 5 or later
- High Level Assembler for MVS & VM & VSE, Release 4 or later

Saving symbol tables in a separate debug file can reduce the size of the load module for your program.

For C and C++ programs, debug tables can be saved in a separate debug file (.dbg file) by specifying the FORMAT(DWARF) suboption of the DEBUG compiler option. z/OS Debugger supports the DEBUG compiler option shipped with z/OS C/C++ Version 1.6 or later.

Programs compiled with the Enterprise COBOL for z/OS Version 5 compiler, Version 6 Release 1 compiler or Version 6 Release 2 and above compiler with the TEST(NOSEPARATE) compiler option have all of their debug information (including the symbol table) stored in a NOLOAD segment of the program object. This segment is only loaded into memory when you are debugging the program object.

Choosing a debugging mode

Use the following list to determine which debugging mode to use for your programs:

For TSO programs
Choose full-screen mode. If you want to use a supported remote debugger, choose remote debug mode.

For JES batch programs
If you want to interact with your batch program, choose full-screen mode using the Terminal Interface Manager. If you want to interact with your batch program using a supported remote debugger, choose remote debug mode. If you don't want to interact with your batch program, use batch mode and specify commands through a commands file and review results in a log file.

For UNIX System Services programs
Choose full-screen mode using the Terminal Interface Manager. If you want to use a supported remote debugger, choose remote debug mode.

For CICS programs

If you want to interact with z/OS Debugger on a 3270 device, choose full-screen mode and one of the following terminal modes:

- Single terminal mode: The application program and z/OS Debugger share the same terminal. Use this terminal mode to debug a transaction that interacts with a 3270 terminal. When you create your CADP or DTCN profile, set the Display Device to the terminal ID that the application program uses.
- Screen control mode: z/OS Debugger displays its screens on a terminal running the DTSC transaction.

  If you use screen control mode, the DTSC transaction runs in the same region as your application program on a terminal of your choice, and displays z/OS Debugger screens on behalf of the task you are debugging, which might not have its own terminal.

  Use screen control mode to debug application programs which are not typically associated with a terminal, and which are running in an MRO environment.

  Screen control mode works in the following manner:

  1. Enter DTSC on the terminal that you want to use to display z/OS Debugger. This terminal can be connected directly to the region where the application program runs, or connected to the region with CRTE or Transaction Routing. If you use Transaction Routing, you must ensure that DTSC runs in the same region as the application program using it.
  2. Set the Display Device in your DTCN or CADP profile to the terminal running the DTSC transaction.
  3. Start the application program.
  4. Press Enter on the terminal running the DTSC transaction to connect to z/OS Debugger.
- Separate terminal mode (formerly called *Dual Terminal Mode*): z/OS Debugger dynamically starts the CDT# transaction on a terminal.

  Use separate terminal mode to debug application programs which are not typically associated with a terminal, and your terminal is connected directly to the region running your application program.

  Separate terminal mode works in the following manner:

  1. Set the Display Device in your DTCN or CADP profile to an available terminal and that terminal can be located by the CICS region running z/OS Debugger.
  2. Start the application program.

If you want to debug your program with a remote debugger, select remote debug mode. Make note of the TCP/IP address of your remote debugger because you will need it when you update your CADP or DTCN profile.

If you do not use single terminal mode and your program sends a screen to the terminal without the WAIT option, CICS Terminal Control holds that screen until the program runs an `EXEC CICS SEND` or `EXEC CICS RECEIVE` statement.

If you want to debug programs that use Distributed Program Link (DPL), you can select one of the following debugging modes:

- Select remote debug mode and use the remote debugger to debug both the DPL client and DPL server.
- Select full screen mode and use two 3270 terminals, one for the DPL client and one for the DPL server.

  You can connect the 3270 terminal to the DPL server in one of the following ways:

  – Directly to the server region.

  – To the client region. If you choose this option, use one of the following terminal modes:

    - Screen Control Mode with DTSC running on a terminal that is connected to the server with CRTE

- Separate Terminal Mode with the terminal connected to the client region and configure the server region so that it looks for the terminal in the client region. To configure the server region, see "Separate terminal mode terminal connects to a TOR and application runs in an AOR" in the *IBM z/OS Debugger Customization Guide*.

For Db2 programs
Choose full-screen mode using the Terminal Interface Manager. If you want to use a supported remote debugger, choose remote debug mode.

For Db2 Stored Procedures
Choose full-screen mode using the Terminal Interface Manager. If you want to use a supported remote debugger, choose remote debug mode.

For IMS TM programs
Choose full-screen mode using the Terminal Interface Manager. If you want to use a supported remote debugger, choose remote debug mode.

For IMS batch programs
If you want to interact with your IMS batch programs, choose full-screen mode using the Terminal Interface Manager. If you want to interact with your IMS batch programs with a supported remote debugger, choose remote debug mode. If you do not want to interact with your IMS batch program, choose batch mode and specify commands through a commands file and review results in a log file.

For IMS BTS programs
If you want your program and your debugging session to run on a single screen, choose full-screen mode. If you want your BTS data to display on your TSO terminal and your debugging session to display on another terminal, choose full-screen mode using the Terminal Interface Manager. If you want your BTS data to display on your TSO terminal and your debugging session to display on a supported remote debugger, choose remote debug mode.

For ALCS programs
You must choose remote debug mode.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IMS/VS Batch Terminal Simulator Program Reference and Operations Manual

Debugging in browse mode

When you debug in some production environments, it might be necessary to restrict your ability to change storage contents and execution flow. Debugging in browse mode enables you to debug your programs while restricting your ability to change storage contents and execution flow. z/OS Debugger uses the RACF® authority of the current user, an EQAOPTS command, or both to determine whether to operate in browse mode.

When you debug in browse mode, you can not do the following actions:

- Modify the contents of memory or registers
- Alter the sequence of program execution

You can use the `QUERY BROWSE MODE` command to determine if browse mode is active.

For information on how to install and control browse mode, see *IBM z/OS Debugger Customization Guide*.

Browse mode debugging in full screen, line, and batch mode

If you are debugging in full screen, line, or batch mode; browse mode is active; and you enter any of the following commands, z/OS Debugger displays a message that the command is not permitted in browse mode:

- `ALLOCATE` command
- Assignment command (assembler and disassembly)
- Assignment command (LangX COBOL)

- Assignment command (PL/I)
- `CALL %CECI` command
- `CALL entry_name` (COBOL)
- `CALL %FM` command
- `CALL %HOGAN` command
- `CLEAR LOG` command
- `COMPUTE` command
- `FREE` command
- `GO BYPASS` command
- `GOTO` command
- `GOTO LABEL` command
- `INPUT` command
- `JUMPTO` command
- `JUMPTO LABEL` command
- `MEMORY` command (z/OS Debugger displays the Memory window, but you cannot modify anything)
- `MOVE` command
- `QUIT` command
- `QUIT expression` command
- `QQUIT` command
- `SET INTERCEPT` command
- `SET` command (COBOL)
- `STORAGE` command
- `SYSTEM` command
- `TRIGGER` command
- `TSO` command

If you enter a command with an *expression* or *condition* that might alter any storage, register, or similar data, or the command invokes any user-written function or alters the sequence of execution, z/OS Debugger displays a message that the command is not permitted in browse mode:

- do/while
- `DO` command (PL/I)
- `EVALUATE` command (COBOL)
- *expression* command (C and C++)
- `for` command (C and C++)
- `%IF` command
- `IF` command
- `LIST expression` command
- `switch` command
- `while` command

Browse mode debugging in remote debug mode

When you use the remote debugger and browse mode is active, the remote debugger does not allow you to do the following actions:

- JumpTo Location – Source window RMB action
- Change Value – Expression, Variable, and Registers RMB action

- Typing over memory in the Memory window

In addition, the remote debugger enforces following restrictions:

- Change Value – the remote debugger does not allow Registers RMB action and displays an error message
- Terminate Button – the program terminates with an abend (instead, click on Disconnect to continue running the program without the debugger)

Also, the remote debugger does not allow you to enter the following Debug Console commands:

- `JUMPTO` (and `JUMPTO` in the Action field of the Add a Breakpoint window)
- `SET INTERCEPT`
- `QUIT`

If an abend occurs while debugging in remote debug mode and browse mode is active, the remote debugger does not give you any continuation options. You can not continue program execution after the abend occurs.

Controlling browse mode

Browse mode can be controlled (activated or deactivated) by changing RACF access, specifying the EQAOPTS BROWSE command, both of these, or neither of these. To control browse mode through RACF access, change your RACF access to the following RACF Facilities:

- For CICS: EQADTOOL.BROWSE.CICS
- For non-CICS: EQADTOOL.BROWSE.MVS

To control browse mode through an EQAOPTS command, specify either ON or OFF for the EQAOPTS BROWSE command.

The following table shows how combinations of these control methods (by RACF access or by the EQAOPTS BROWSE command) can activate or deactivate browse mode. For instructions using these controls see *IBM z/OS Debugger Customization Guide*.

Table 13. How different combinations of RACF access and the EQAOPTS BROWSE command activate or deactivate browse mode.

	Setting of the EQAOPTS BROWSE command		
Status of RACF access	**Not set (use RACF status)**	**ON**	**OFF**
facility (access) not defined	normal mode (browse mode is not active)	browse mode is active	normal mode
ACCESS=NONE	Cannot use z/OS Debugger	Cannot use z/OS Debugger	Cannot use z/OS Debugger
ACCESS=READ	browse mode is active	browse mode is active	browse mode is active
ACCESS=UPDATE (or higher)	normal mode	browse mode is active	normal mode

Choosing a method or methods for starting z/OS Debugger

Table 14 on page 54 indicates that there are several different methods to start z/OS Debugger for each type of program. In this topic, you will read about the circumstances in which each applicable method works for each type of program. Then you can select which method would work best for your site. After you complete this topic, you will have selected the methods that work best for your programs.

Table 14. Methods for specifying the TEST runtime options and the subsystems that support these methods.

	TSO	JES batch	UNIX System Services [1]	CICS	Db2	Db2 stored procedures (PROGRAM TYPE=MAIN)	Db2 stored procedures (PROGRAM TYPE=SUB)	IMS TM	IMS batch	IMS BTS
Use the DFSBXITA user exit								X	X	X
Use the CADP transaction				X						
Use the DTCN transaction				X						
Use the Db2 catalog						X[3]	X			
From within a program by coding a call to CEETEST, __ctest(), or PLITEST	X	X	X	X	X	X	X	X	X	X
Through CEEUOPT or CEEROPT	X	X	X	X[2]	X[2]	X[2,3]		X	X	X
Use the CEEOPTS DD statement in JCL or CEEOPTS allocation in TSO	X	X	X		X				X	X
Use the parameters on the EXEC statement when you start your program		X								
Use the parameters on the RUN statement when you start your program					X					
Use the parameters on the CALL statement when you start your program	X									
Through the EQASET transaction[4]								X[4]		
Through the EQANMDBG program[5]	X[5]	X[5]							X[5]	X[5]
Use the EQAD3CXT user exit routine		X				X	X[6]	X	X	X

Note:

1. Go programs only run under UNIX System Services, with the following limitations:
   - __ctest() might not work as expected when cgo is in use.
   - CEEOPTS allocation cannot be used.
2. You cannot use CEEROPT to specify TEST runtime options.
3. The Db2 catalog method always takes precedence over CEEUOPT.
4. This method is only for non-Language Environment assembler programs.
5. This method is only for non-Language Environment programs.
6. EQAD3CXT supports Db2 stored procedures `PROGRAM TYPE=SUB` if you set the `RRTN_SW` flag as x`'01'`.

For each subsystem, Table 14 on page 54 shows that you can choose from several different methods of specifying the TEST runtime options. The following list can help you select the method that best applies to your situation, ordered by flexibility and convenience:

For TSO programs

- For programs that start in Language Environment, specify the TEST runtime options using the CEEOPTS allocation in TSO for the most flexible method of specifying the runtime options.
- Specify the TEST runtime options using the parameters on the CALL statement if you have a small number of runtime options or need to invoke EQANMDBG for a non-Language Environment program.
- If you specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST, you will have to recompile your program every time you want to change the options.

For JES batch programs

- For programs that start in Language Environment, specify the TEST runtime options using the CEEOPTS DD statement in your JCL for the most flexible method of specifying runtime options.
- Specify the TEST runtime options using the parameters on the EXEC statement option if you have a small number of runtime options or need to invoke EQANMDBG for a non-Language Environment program.
- If you specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST, you will have to recompile your program every time you want to change the options.

For UNIX System Services programs

- Specify the TEST runtime options by setting the _CEE_RUNOPTS environment variable.
- If you specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST, you will have to recompile your program every time you want to change the options.

For CICS programs

- Specify the TEST runtime options using either the DTCN or CADP transaction to create and store a profile that contains the TEST runtime options.
- If you specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST, you will have to recompile your program every time you want to change the options.

For Db2 programs

- Specify the TEST runtime options using the CEEOPTS DD statement in JCL or CEEOPTS allocation in TSO for the most flexible method of specifying runtime options.
- Specify the TEST runtime options using the parameters on the RUN statement option if you have a small number of runtime options.
- If you specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST, you will have to recompile your program every time you want to change the options.

For Db2 stored procedures that have the PROGRAM TYPE of MAIN

- Specify the TEST runtime options using the Language Environment EQAD3CXT user exit routine. You can run the stored procedure with your own set of suboptions. Another user can run or debug the stored procedure with a separate set of suboptions. Therefore, multiple users can run or debug the stored procedure at the same time.
- If the exit routine is not available at your site, specify the TEST runtime options using the Db2 catalog. However, you are limited to specifying one specific set of suboptions, which means that every user that runs or debugs that stored procedure uses the same set of suboptions.

If you implement both methods, the Language Environment exit routine takes precedence over the Db2 catalog.

For Db2 stored procedures that have the PROGRAM TYPE of SUB

- For programs defined as PROGRAM TYPE=SUB, specify the TEST runtime options using the Language Environment EQAD3CXT exit routine. You can run or debug the Db2 stored procedure with your own set of suboptions, while another user can run or debug the Db2 stored procedure with a separate set of suboptions.
- If the exit routine is not available at your site, specify the TEST runtime options using the Db2 catalog. You are limited to specifying one set of suboptions, which means that every user that runs or debugs that stored procedure uses the same set of suboptions.

If you implement both methods, the Language Environment exit routine takes precedence over the Db2 catalog.

For programs invoked by any other method, specify the TEST runtime options using the Db2 catalog. You are limited to specifying one set of suboptions, which means that every user that runs or debugs that stored procedure uses the same set of suboptions.

For IMS TM programs

- Specify the TEST runtime options using the Language Environment EQAD3CXT user exit routine.
- If your program is a non-Language Environment program, issue the EQASET transaction to setup your debugging preference.
- If the EQAD3CXT user exit routine is not available at your site, specify the TEST runtime options using the DFSBXITA user exit routine.

- If the EQAD3CXT or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
- If none of the previous options is available at your site, specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST. However, you will have to recompile your program every time you want to change the options.

For IMS batch programs

- For programs that start in Language Environment, specify the TEST runtime options using the CEEOPTS allocation in JCL because this method can be the most flexible method.
- Specify the TEST runtime options using the EQAD3CXT user exit routine.
- If your program is a non-Language Environment program, use the EQANMDBG program to start your debugging session.
- If the EQAD3CXT user exit routine is not available at your site, specify the TEST runtime options using the DFSBXITA user exit routine; however, you must specify PROGRAM rather than TRANSACTION.
- If the EQAD3CXT or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
- If none of the previous options is available at your site, specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST. However, you will have to recompile your program every time you want to change the options.

For IMS BTS programs

- For programs that start in Language Environment, specify the TEST runtime options using the CEEOPTS allocation in JCL because this method can be the most flexible method.
- Specify the TEST runtime options using the EQAD3CXT user exit routine.
- If your program is a non-Language Environment program, use the EQANIAFE application front-end program to start your debug session. For more information, see "Debugging non-Language Environment IMS BTS programs" on page 340.
- If the EQAD3CXT user exit routine is not available at your site, specify the TEST runtime options using the DFSBXITA user exit routine.
- If the EQAD3CXT or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
- If none of the previous options is available at your site, specify the TEST runtime options by coding a call to CEETEST, __ctest(), or PLITEST. However, you will have to recompile your program every time you want to change the options.

After you have identified the method or methods you will use to start z/OS Debugger, see Chapter 4, "Planning your debug session," on page 25 to determine the next task you must complete.

Choosing how to debug old COBOL programs

Programs compiled with the OS/VS COBOL compiler can be debugged by doing one of the following:

- Debug them as LangX COBOL programs.
- Convert them to the 1985 COBOL Standard level and compile them with the Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM compiler. You can use the Load Module Analyzer to identify OS/VS COBOL programs in a load module, then use COBOL and CICS Command Level Conversion Aid (CCCA) to convert the programs.

To convert an OS/VS COBOL program to 1985 COBOL Standard, do the following steps:

1. Identify the OS/VS COBOL programs in your load module by using the Load Module Analyzer. For instructions on using Load Module Analyzer, see Appendix I, "z/OS Debugger Load Module Analyzer," on page 505.

2. Convert your OS/VS COBOL source by using COBOL and CICS Command Level Conversion Aid (CCCA). For instructions on using CCCA, see *COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM User's Guide*.
3. Compile the new source with either the Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM.

   You can combine steps 2 and 3 by using the **Convert and Compile** option of IBM z/OS Debugger Utilities.
4. Debug the object module by using z/OS Debugger.

After you convert and debug your program, you can do one of the following options:

- Continue to use the OS/VS COBOL compiler. Every time you want to debug your program, you need to do the steps described in this section.
- Use the new source that was produced by the steps described in this section. You can compile the source and debug it without repeating the steps described in this section.

CCCA can use any level of COBOL source program as input, including VS COBOL II, COBOL for MVS & VM, and COBOL for OS/390 & VM programs that were previously compiled with the CMPR2 compiler option.

Creating deferred breakpoints for COBOL and PL/I programs

Creating a list of breakpoints before starting the z/OS Debugger session reduces system resource usage and the time spent in the debugging session.

To create and use the deferred breakpoints, complete the following steps:

- Create breakpoints and save the definitions in a file-based repository using the Create breakpoints option in the z/OS Debugger Deferred Breakpoints selection in DTU. You can also use IBM Fault Analyzer to create breakpoints. See *IBM Fault Analyzer User's Guide and Reference* for details.
- View the breakpoints in the repository and save the definitions in a commands file in the z/OS Debugger command format using the View breakpoints option in the z/OS Debugger Deferred Breakpoints selection in DTU.
- Set the breakpoints that are defined in the commands file during the debug session by using one of the methods where the commands file is accepted like a commands file, a preference file, or a USE command.

The breakpoint types supported are AT STATEMENT and AT LABEL.

The following programming languages and side file configurations are supported:

Table 15. The supported programming languages and side file configurations

Programming language	**Side file**	**Compiled with**
Enterprise COBOL V4 or earlier	LANGX	NOTEST
Enterprise COBOL V4 or earlier	SYSDEBUG	TEST (SEPARATE)
Enterprise COBOL V5	Program Object	TEST (SOURCE)
Enterprise PL/I	SYSDEBUG	TEST (SYM,SEPARATE)

Chapter 5. Updating your processes so you can debug programs with z/OS Debugger

After you have completed the tasks in Chapter 4, "Planning your debug session," on page 25, you can use the information you have collected to update the following processes:

- Your compilation and linking processes so that programs are compiled with the correct compiler options and suboptions and that the required files are saved (for example, the separate debug file).
- Your library or promotion processes so that files containing information that z/OS Debugger needs to debug your programs are available.
- Your libraries or security systems so that you have access to the files that z/OS Debugger needs to debug your programs. For example, if you have RACF security measures, you might need to update them so that z/OS Debugger can access the files it needs.

For more information about how to update these processes, see the following topics:

- "Update your compilation, assembly, and linking process" on page 59
- "Update your library and promotion process" on page 64
- "Make the modifications necessary to implement your preferred method of starting z/OS Debugger" on page 64

Update your compilation, assembly, and linking process

This topic describes the changes you must make to your compilation, assembly, and linking process to implement the choices you made in Chapter 4, "Planning your debug session," on page 25. If you are familiar with managing JCL and with your site's compilation or assembly process, see "Compiling your program without using IBM z/OS Debugger Utilities" on page 59 for instructions on the specific changes you need to make. If your site uses IBM z/OS Debugger Utilities to manage these processes, see "Compiling your program by using IBM z/OS Debugger Utilities" on page 61 for instructions on how to use the **Program Preparation** option to update these processes.

Compiling your program without using IBM z/OS Debugger Utilities

Create or modify JCL so that it includes all the statements you need to compile or assemble your programs, then properly link any libraries. The following list describes the changes you need to make:

- Specify the correct compiler options and suboptions that you chose from Table 8 on page 26.

  For each compiler, there might be additional updates you might need to make so that z/OS Debugger starts. The following list describes these updates:

  - If you are compiling an Enterprise PL/I program on an HFS or zFS file system, see "Compiling a Enterprise PL/I program on an HFS or zFS file system" on page 62.
  - If you are compiling a C program on an HFS or zFS file system, see "Compiling a C program on an HFS or zFS file system" on page 63.
  - If you are compiling a C program with c89 or c++, see "Compiling your C program with c89 or c++" on page 62.
  - If you are compiling a C++ program on an HFS or zFS file system, see "Compiling a C++ program on an HFS or zFS file system" on page 63.
- Specify the statements to save the files that z/OS Debugger needs. Table 16 on page 60 can help you identify which file you need to save for a particular compiler option. For example, if you are compiling a COBOL program with the SEPARATE suboption of the TEST compiler option, make sure you specify the DD statement with the name of the separate debug file.

© Copyright IBM Corp. 1992, 2023

- If you are using other Application Delivery Foundation for z/OS tools, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* that correspond to the compilers or assembler that you are using. Those topics contain instructions on other updates you must make to your compilation, assembler, and linking processes.
- If YES is specified for the EQAOPT MDBG command (which requires z/OS Debugger to search for a .dbg file in a .mdbg file)[6], verify that the .mdbg file is a non-temporary file and is available during the debug session. Ensure that the .mdbg file was created with captured source by using the -c option for the dbgld command or the CAPSRC option on the CDADBGLD utility.
- For LangX COBOL programs, write JCL that generates the EQALANGX file, as described in "Creating the EQALANGX file for LangX COBOL programs" on page 68.
- For assembler programs, write a SYSADATA DD statement that generates the EQALANGX files, as described in "Creating the EQALANGX file for an assembler program" on page 71.
- For Db2 programs, specify the correct Db2 preprocessor and coprocessor, as described in "Processing SQL statements" on page 75.

Table 16. Files that you need to save when compiling with a particular compiler option or suboption

Programming language	Compiler suboption or assembler option	File you need to save
COBOL		
	SEPARATE	separate debug file
	any other	listing[7]
	NOTEST	listing[7]
LangX COBOL		
	"Compiling your OS/VS COBOL program " on page 67 "Compiling your VS COBOL II program " on page 68 "Compiling your Enterprise COBOL program " on page 68	EQALANGX
	any other	listing file containing pseudo-assembler code
PL/I		
	SEPARATE	separate debug file
	any other (pre-Enterprise PL/I)	listing file
	any other (Enterprise PL/I)	source file that was used as input to the compiler
	NOTEST	listing file containing pseudo-assembler code
C/C++		
	DEBUG(DWARF)	the .dbg file and source file If you are using an .mdbg file that stores the source file, then save that .mdbg file.
	TEST	source file that was used as input to the compiler

[6] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

[7] It is except for Enterprise COBOL for z/OS Version 5.

Table 16. Files that you need to save when compiling with a particular compiler option or suboption (continued)		
Programming language	**Compiler suboption or assembler option**	**File you need to save**
	NOTEST	listing file containing pseudo-assembler code
assembler		
	ADATA	EQALANGX
	no debug information saved	listing file containing pseudo-assembler code

After you complete this task, see "Update your library and promotion process" on page 64.

Compiling your program by using IBM z/OS Debugger Utilities

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

z/OS Debugger Utilities provides several utilities than can help you compile your programs and start z/OS Debugger. The steps described in this topic apply to the following category of compilers and assemblers:

- Enterprise PL/I
- Enterprise COBOL
- C/C++
- Assembler

If you are using IBM z/OS Debugger Utilities to prepare your program and start z/OS Debugger, read Appendix C, "Examples: Preparing programs and modifying setup files with IBM z/OS Debugger Utilities," on page 415, which describes how to prepare a sample program and start z/OS Debugger by using IBM z/OS Debugger Utilities. After you read the sample and understand how to use IBM z/OS Debugger Utilities, do the following steps:

1. Start IBM z/OS Debugger Utilities.
2. Type in "1" to select Program Preparation, then press Enter.
3. Type in the number that corresponds to the compiler you want to use, then press Enter.
4. Type in the information about the program you are compiling and select the appropriate options for the CICS and Db2/SQL fields.

   If the program source is a sequential data set and the Db2 precompiler is selected, make sure the DBRMLIB data set field in panel EQAPPC1B, EQAPPC2B, EQAPPC3B, EQAPPC4B, or EQAPPC5B is a partitioned data set with a member name. For example, `DEBUG.TEST.DBRMLIB(PROG1)`.

   Type in the backslash character ("/") in the **Enter / to edit options and data set name patterns** field, then press Enter.
5. Using the information you collected in Table 8 on page 26, fill out the fields with the appropriate values. After you have made all the changes you want to make, press PF3 to save this information and return to the previous panel.
6. Review the choices you made. Press Enter.
7. Verify your selections, then press Enter.
8. After the compilation is done, a panel is displayed. If there were errors in the compilation, review the messages and make any changes. Return to step 1 to repeat the compilation.
9. Press PF3 until you return to the Program Preparation panel.
10. In the Program Preparation panel, type in "L", then press Enter.
11. In the Link Edit panel, specify whether you want the link edit to run in the foreground or background. Specify the name of other libraries you need to link to your program. After you are done making all your changes, press Enter.

12. Verify any selections, then press Enter.
13. After the link edit is done, if there were errors in the link edit, review the messages and make any changes. Return to step 1 to repeat the process.
14. Press PF3 until you return to the main IBM z/OS Debugger Utilities panel.

After you complete this task, see "Update your library and promotion process" on page 64.

Compiling a Enterprise PL/I program on an HFS or zFS file system

If you are compiling and launching Enterprise PL/I programs on an HFS or zFS file system, you must do one of the following:

- Compile and launch the programs from the same location, or
- specify the full path name when you compile the programs.

By default, the Enterprise PL/I compiler stores the relative path and file names in the object file. When you start a debug session, if the source is not in the same location as where the program is launched, z/OS Debugger does not locate the source. To avoid this problem, specify the full path name for the source when you compile the program. For example, if you execute the following series of commands, z/OS Debugger does not find the source because it is located in another directory (`/u/myid/mypgm`):

1. Change to the directory where your program resides and compile the program.

```
cd /u/myid/mypgm
pli -g "//TEST.LOAD(HELLO)" hello.pli
```

2. Exit UNIX System Services and return to the TSO READY prompt.
3. Launch the program with the TEST run-time option.

```
call TEST.LOAD(HELLO) 'test/'
```

z/OS Debugger does find the source if you change the compile command to:

```
pli -g "//TEST.LOAD(HELLO)" /u/myid/mypgm/hello.pli
```

The same restriction applies to programs that you compile to run in a CICS environment.

Compiling your C program with c89 or c++

If you build your application using the c89 or c++, do the following steps:

1. Compile your source code as usual, but specify the –g option to generate debugging information. The –g option is equivalent to the TEST compiler option under TSO or MVS batch. For example, to compile the C source file `fred.c` from the u/mike/app directory, specify:

```
cd /u/mike/app
c89 –g –o "//PROJ.LOAD(FRED)" fred.c
```

   Note: The quotation marks (") in the command line above are required.
2. Set up your TSO environment, as described in "Compiling your program without using IBM z/OS Debugger Utilities" on page 59 or "Compiling your program by using IBM z/OS Debugger Utilities" on page 61.
3. Debug the program under TSO by entering the following:

```
FRED TEST ENVAR('PWD=/u/mike/app') / asis
```

   Note: The apostrophes (') in the command line above are required. `ENVAR('PWD=/u/mike/app')` sets the environment variable `PWD` to the path from where the source files were compiled. z/OS Debugger uses this information to determine from where it should read the source files.

If you are creating .mdbg files, capture the source files into the .mdbg file by specify the -c option with the dbgld command, or the CAPSRC option with the CDADBGLD utility. To learn how to use the dbgld command and the CDADBGLD utility, see *z/OS XL C/C++ User's Guide*. z/OS Debugger needs access to the .mdbg file to debug your program.

Compiling a C program on an HFS or zFS file system

If you are compiling and launching programs on an HFS or zFS file system, you must do one of the following:

- Compile and launch the programs from the same location.
- Specify the full path name when you compile the programs.

By default, the C compiler stores the relative path and file names of the source files in the object file. When you start a debug session, if the source is not in the same location as where the program is launched, z/OS Debugger does not find the source. To avoid this problem, specify the full path name of the source when you compile the program. For example, if you execute the following series of commands, z/OS Debugger does not find the source because it is located in another directory (`/u/myid/mypgm`):

1. Change to the directory where your program resides and compile the program.

   ```
   cd /u/myid/mypgm
   c89 -g -o "//TEST.LOAD(HELLO)" hello.c
   ```

2. Exit UNIX System Services and return to the TSO READY prompt.
3. Launch the program with the TEST run-time option.

   ```
   call TEST.LOAD(HELLO) 'test/'
   ```

z/OS Debugger finds the source if you change the compile command to:

```
c89 -g -o "//TEST.LOAD(HELLO)" /u/myid/mypgm/hello.c
```

The same restriction applies to programs that you compile to run in a CICS environment.

If you are creating .mdbg files, capture the source files into the .mdbg file by specify the -c option with the dbgld command, or the CAPSRC option with the CDADBGLD utility. To learn how to use the dbgld command and the CDADBGLD utility, see *z/OS XL C/C++ User's Guide*. z/OS Debugger needs access to the .mdbg file to debug your program.

Compiling a C++ program on an HFS or zFS file system

If you are compiling and launching programs on an HFS or zFS file system, you must do one of the following:

- Compile and launch the programs from the same location, or
- specify the full path name when you compile the programs.

By default, the C++ compiler stores the relative path and file names of the source files in the object file. When you start a debug session, if the source is not in the same location as where the program is launched, z/OS Debugger does not locate the source. To avoid this problem, specify the full path name of the source when you compile the program. For example, if you execute the following series of commands, z/OS Debugger does not find the source because it is located in another directory (`/u/myid/mypgm`):

1. Change to the directory where your program resides and compile the program.

   ```
   cd /u/myid/mypgm
   c++ -g -o "//TEST.LOAD(HELLO)" hello.cpp
   ```

2. Exit UNIX System Services and return to the TSO READY prompt.

3. Launch the program with the TEST run-time option.

```
call TEST.LOAD(HELLO) 'test/'
```

z/OS Debugger finds the source if you change the compile command to:

```
c++ -g -o "//TEST.LOAD(HELLO)" /u/myid/mypgm/hello.cpp
```

The same restriction applies to programs that you compile to run in a CICS environment.

If you are creating .mdbg files, capture the source files into the .mdbg file by specify the -c option with the dbgld command, or the CAPSRC option with the CDADBGLD utility. To learn how to use the dbgld command and the CDADBGLD utility, see *z/OS XL C/C++ User's Guide*. z/OS Debugger needs access to the .mdbg file to debug your program.

Update your library and promotion process

If you use a library to maintain your program and a promotion process to move programs through levels of quality and testing, you might have to update these processes to ensure that z/OS Debugger can find the files it needs to obtain information about your programs. For example, if your final production level does not have access to the same libraries as your development level, and you want to be able to debug programs that are in the final product level, you might need to update the environment in your final production level so that it can access to the following resources:

- All the data sets required to debug your program, for example, the source file, listing file, separate debug file, or EQALANGX file.
- Access to all the libraries required by your program or z/OS Debugger.

If you are using other Application Delivery Foundation for z/OS tools, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* that correspond to the compilers or assembler that you are using. Those topics give instructions on which files to move through your levels so that the Application Delivery Foundation for z/OS tools can find the files they need.

If you manage your source code with a library system that requires you specify the SUBSYS=*ssss* parameter when you allocate a data set, you or your site need to specify the EQAOPTS SUBSYS command, which provides the value for *ssss*. You must do this for the following types of programs:

- Enterprise PL/I program that was compiled without the SEPARATE suboption of TEST compiler option
- C/C++ programs

This support is not available for CICS programs. To learn how to specify EQAOPTS commands, see the *IBM z/OS Debugger Reference and Messages* or the *IBM z/OS Debugger Customization Guide*.

Make the modifications necessary to implement your preferred method of starting z/OS Debugger

In this topic, you will use the information you gathered after completing 2 in Chapter 4, “Planning your debug session,” on page 25 and “Choosing a method or methods for starting z/OS Debugger” on page 53 to write the TEST runtime options string, then save that string in the appropriate location.

You might have to write several different TEST runtime options strings. For example, the TEST runtime options string that you write for your CICS programs might not be the same TEST runtime options string you can use for your IMS programs. For this situation, you might want to use Table 17 on page 64 to record the string you want to use for each type of program you are debugging.

Table 17. Record the TEST runtime options strings you need for your site

	Test runtime options string (for example, `TEST(ALL,,,MFI%SYSTEM01.TRMLU001:)`)
TSO	

Table 17. Record the TEST runtime options strings you need for your site (continued)	
	Test runtime options string (for example, `TEST(ALL,,,MFI%SYSTEM01.TRMLU001:)`)
JES batch	
UNIX System Services	
CICS	
Db2	
Db2 stored procedures (PROGRAM TYPE=MAIN)	
Db2 stored procedures (PROGRAM TYPE=SUB)	
IMS TM	
IMS batch	
IMS BTS	

If you are not familiar with the format of the TEST runtime option string, see the following topics:

- Description of the TEST runtime option in *IBM z/OS Debugger Reference and Messages*
- Chapter 13, “Writing the TEST runtime option string,” on page 107

After you have written the TEST runtime option strings, you need to save them in the appropriate location. Using the information you recorded in Table 14 on page 54, review the following list, which directs you to the instructions on where and how to save the TEST runtime options strings:

Through the EQAD3CXT user exit routine
See Chapter 12, “Specifying the TEST runtime options through the Language Environment user exit,” on page 97.

Through the DFSBXITA user exit routine
See “Setting up the DFSBXITA user exit routine” on page 96.

Using the CADP transaction
See “Creating and storing debugging profiles with CADP” on page 92.

Using the DTCN transaction
See “Creating and storing a DTCN profile” on page 82.

Using the Db2 catalog
See Chapter 9, “Preparing a Db2 stored procedures program,” on page 79.

By coding a call to CEETEST, __ctest(), or PLITEST
See one of the following topics:

- “Starting z/OS Debugger with CEETEST” on page 119
- “Starting z/OS Debugger with the __ctest() function” on page 126
- “Starting z/OS Debugger with PLITEST” on page 125

Through CEEUOPT or CEEROPT
See one of the following topics:

- “Starting z/OS Debugger under CICS by using CEEUOPT” on page 141

- “Linking Db2 programs for debugging” on page 76
- “Starting z/OS Debugger under IMS by using CEEUOPT or CEEROPT” on page 95

Using the CEEOPTS DD statement in JCL or CEEOPTS allocation in TSO
Use the **JCL for Batch Debugging** option in IBM z/OS Debugger Utilities.

Using the parms on the EXEC statement when you start your program
When you specify the EXEC statement, include the TEST runtime option as a parameter.

Use the parms on the RUN statement when you start your program
When you specify the RUN statement, include the TEST runtime option as a parameter.

Using the parms on the CALL statement when you start your program
See the example in “Starting z/OS Debugger” on page 12.

Through the EQASET transaction
See “Running the EQASET transaction for non-Language Environment IMS MPPs” on page 342.

Through the EQANMDBG program
See “Starting z/OS Debugger for programs that start outside of Language Environment” on page 134.

Chapter 6. Preparing a LangX COBOL program

Note: This chapter is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

This chapter describes how to prepare a LangX COBOL program that you can debug with z/OS Debugger.

The term *LangX COBOL* refers to any of the following programs:

- A program compiled with the IBM OS/VS COBOL compiler.
- A program compiled with the IBM VS COBOL II compiler with the NOTEST compiler option.
- A program compiled with the IBM Enterprise COBOL for z/OS Version 3 or Version 4 compiler with the NOTEST compiler option.

To prepare a LangX COBOL program, you must do the following steps:

1. Compile your program with the IBM OS/VS COBOL, the IBM VS COBOL II, or the IBM Enterprise COBOL compiler using the proper options.
2. Create the EQALANGX file.
3. Link-edit your program.

As you read through the information in this document, remember that OS/VS COBOL programs are non-Language Environment programs, even though you might have used Language Environment libraries to link and run your program.

VS COBOL II programs are non-Language Environment programs when you link them with the non-Language Environment library. VS COBOL II programs are Language Environment programs when you link them with the Language Environment library.

Enterprise COBOL programs are always Language Environment programs. Note that COBOL DLL's cannot be debugged as LangX COBOL programs.

Read the information regarding non-Language Environment programs for instructions on how to start z/OS Debugger and debug non-Language Environment COBOL programs, unless information specific to LangX COBOL is provided.

Compiling your OS/VS COBOL program

You must compile your OS/VS COBOL program with the IBM OS/VS COBOL compiler and use the following options:

- NOTEST
- SOURCE
- DMAP
- PMAP
- VERB
- XREF
- NOLST
- NOBATCH
- NOSYMDMP
- NOCOUNT

If you are using other Application Delivery Foundation for z/OS tools (for example, Application Performance Analyzer), you might need to specify additional compiler options. To understand how the Application Delivery Foundation for z/OS tools work together, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*. To learn which additional compiler

© Copyright IBM Corp. 1992, 2023

options you might need to specify, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

Compiling your VS COBOL II program

You must compile your VS COBOL II program with the IBM VS COBOL II compiler and use the following options:

- NOTEST
- NOOPTIMIZE
- SOURCE
- MAP
- XREF
- LIST or OFFSET

If you are using other Application Delivery Foundation for z/OS tools (for example, Application Performance Analyzer), you might need to specify additional compiler options. To understand how the Application Delivery Foundation for z/OS tools work together, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*. To learn which additional compiler options you might need to specify, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

Compiling your Enterprise COBOL program

You must compile your Enterprise COBOL program with the IBM Enterprise COBOL compiler and use the following options:

- NOTEST
- NOOPTIMIZE
- SOURCE
- MAP
- XREF
- LIST

Creating the EQALANGX file for LangX COBOL programs

Note: The EQALANGX program is part of IBM Application Delivery Foundation for z/OS Common Components, which is not shipped with IBM Z and Cloud Modernization Stack (Wazi Code).

Use the EQALANGX program to create the EQALANGX file. The EQALANGX program is an alias of IPVLANGX, which is shipped as part of the ADFz Common Components. It is in IPV.SIPVMODA. It is the same as the IDILANGX alias that Fault Analyzer uses and the CAZLANGX alias that Application Performance Analyzer uses. The module names can be used interchangeably.

For further information about the xxxLANGX program, look for IDILANGX in the *Fault Analyzer User's Guide and Reference*. For return codes and messages, look for IPVLANGX in the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

To create the EQALANGX file, do the following steps:

1. Create JCL similar to the following:

```
//XTRACT EXEC PGM=EQALANGX,REGION=32M,
// PARM='(COBOL ERROR LOUD'
//STEPLIB DD DISP=SHR,DSN=IPV.SIPVMODA
//LISTING DD DISP=SHR,DSN=yourid.langxcompiler.listing
//IDILANGX DD DISP=OLD,DSN=yourid.EQALANGX
```

The following list describes the variables used in this example and the parameters you can use with the EQALANGX program:

PARM=

COBOL
The COBOL parameter indicates that a LangX COBOL module is being processed.

ERROR
The ERROR parameter is suggested, but optional. If you specify it, additional information is displayed when an error is detected.

LOUD
The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.

64K CREF
The 64K and CREF parameters are optional. Previously, these options were required.

The messages displayed by specifying the ERROR and LOUD parameters are Write To Operator or Write To Programmer (WTO or WTP) messages. See the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* for detailed information about the messages and return codes displayed by the IPVLANGX program.

IPV.SIPVMODA
The name of the data set that contains the ADFz Common Components load modules. If the ADFz Common Components load modules are in a system linklib data set, you can omit the following line:

```
//STEPLIB DD DISP=SHR,DSN=IPV.SIPVMODA
```

yourid.langxcompiler.listing
The name of the listing data set generated by the IBM OS/VS COBOL, IBM VS COBOL II, or IBM Enterprise COBOL compiler. If this is a partitioned data set, the member name must be specified. For information about the characteristics of this data set, see *IBM OS/VS COBOL Compiler and Library Programmer's Guide*, *VS COBOL II Application Programming Guide for MVS and CMS*, or *Enterprise COBOL for z/OS Programming Guide*.

***yourid*.EQALANGX**
The name of the data set where the EQALANGX debug file is to be placed. This data set must have variable block record format (RECFM=VB) and a logical record length of 1562 (LRECL=1562).

z/OS Debugger searches for the EQALANGX debug file in a partitioned data set with the name *yourid*.EQALANGX and a member name that matches the name of the program. If you want the member name of the EQALANGX debug file to match the name of the program, you do not need to specify a member name on the DD statement.

2. Submit the JCL and verify that the EQALANGX file is created in the location you specified on the IDILANGX DD statement.

Link-editing your program

You can link-edit your program by using your normal link-edit procedures.

After you link-edit your program, you can run your program and start z/OS Debugger.

Chapter 7. Preparing an assembler program

To debug an assembler program with the full capabilities of z/OS Debugger, you need to prepare the program.

1. Assemble your program with the proper options.
2. Create the EQALANGX file.
3. Link-edit your program.

If you use IBM z/OS Debugger Utilities to prepare your assembler program, you can do steps 1 and 2 in one step.

Before you assemble your program

When you debug an assembler program, you can use most of the z/OS Debugger commands. There are three differences between debugging an assembler program and debugging programs written in other programming languages supported by z/OS Debugger:

- After you assemble your program, you must create a debug information file, also called the EQALANGX file. z/OS Debugger uses this file to obtain information about your assembler program.
- z/OS Debugger assumes all compile units are written in some high-level language (HLL). You must inform z/OS Debugger that a compile unit is an assembler compile unit and instruct z/OS Debugger to load the assembler compile unit's debug information. Do this by entering the LOADDEBUGDATA (or LDD) command.
- Assembler does not have language elements you can use to write expressions. z/OS Debugger provides assembler-like language elements you can use to write expressions for z/OS Debugger commands that require an expression. See *IBM z/OS Debugger Reference and Messages* for a description of the syntax of the assembler-like language.

After you verify that your assembler program meets these requirements, prepare your assembler program by doing the following tasks:

1. “Assembling your program” on page 71.
2. “Creating the EQALANGX file for an assembler program” on page 71.

“Assembling your program and creating EQALANGX” on page 72 describes how to prepare an assembler program by using IBM z/OS Debugger Utilities.

Assembling your program

If you assemble your program without using IBM z/OS Debugger Utilities, you must use the High Level Assembler (HLASM) and specify a SYSADATA DD statement and the ADATA option. This causes the assembler to create a SYSADATA file. The SYSADATA file is required to generate the debug information (the EQALANGX file) used by z/OS Debugger.

If you are using other Application Delivery Foundation for z/OS tools, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* to make sure you specify all the assembler options you need to create the files needed by all the Application Delivery Foundation for z/OS tools.

Creating the EQALANGX file for an assembler program

Note: The EQALANGX program is part of IBM Application Delivery Foundation for z/OS Common Components, which is not shipped with IBM Z and Cloud Modernization Stack (Wazi Code).

Use the EQALANGX program to create the EQALANGX file. The EQALANGX program is an alias of IPVLANGX, which is shipped as part of the ADFz Common Components. It is in IPV.SIPVMODA. It is

© Copyright IBM Corp. 1992, 2023

the same as the IDILANGX alias that Fault Analyzer uses and the CAZLANGX alias that Application Performance Analyzer uses. The module names can be used interchangeably.

For further information about the xxxLANGX program, look for IDILANGX in the *Fault Analyzer User's Guide and Reference*. For return codes and messages, look for IPVLANGX in the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

To create the EQALANGX files without using IBM z/OS Debugger Utilities, use JCL similar to the following:

```
//XTRACT EXEC PGM=EQALANGX,REGION=32M,
//  PARM='(ASM ERROR LOUD'
//STEPLIB DD DISP=SHR,DSN=IPV.SIPVMODA
//SYSADATA DD DISP=SHR,DSN=yourid.sysadata
//IDILANGX DD DISP=OLD,DSN=yourid.EQALANGX
```

The following list describes the variables used in this example the parameters you can use with the EQALANGX program:

PARM=

(ASM
Indicates that an assembler module is being processed.

ERROR
This parameter is suggested but optional. If you specify it, additional information is displayed when an error is detected.

LOUD
The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.

The messages displayed by specifying the ERROR and LOUD parameters are Write To Operator or Write To Programmer (WTO or WTP) messages. See the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide* for detailed information about the messages and return codes displayed by the IPVLANGX program.

IPV.SIPVMODA
The name of the data set that contains the ADFz Common Components load modules. If the ADFz Common Components load modules are in a system linklib data set, you can omit the following line:

```
//STEPLIB DD DISP=SHR,DSN=IPV.SIPVMODA
```

yourid.sysadata
The name of the data set containing the SYSADATA output from the assembler. If this is a partitioned data set, the member name must be specified. For information about the characteristics of this data set, see *HLASM Programmer's Guide*.

***yourid*.EQALANGX**
The name of the data set where the EQALANGX debug file is to be placed. This data set must have variable block record format (RECFM=VB) and a logical record length of 1562 (LRECL=1562).

z/OS Debugger searches for the EQALANGX debug file in a partitioned data set with the name *yourid*.EQALANGX and a member name that matches the name of the first CSECT in the assembly. If you want the member name of the EQALANGX debug file to match the first CSECT in the assembly, you do not need to specify a member name on the DD statement. Otherwise, you must specify a member name on the DD statement. In this case, you must use the SET SOURCE command to direct z/OS Debugger to the member containing the EQALANGX data.

z/OS Debugger does not support debugging of Private Code (unnamed CSECT).

Assembling your program and creating EQALANGX

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can assemble your program and create the EQALANGX file at the same time by using IBM z/OS Debugger Utilities. Do the following:

1. Start IBM z/OS Debugger Utilities. The `IBM z/OS Debugger Utilities` panel is displayed.
2. Select option 1, "Program Preparation" . The `z/OS Debugger Program Preparation` panel is displayed.
3. Select option 5, "Assemble". The `z/OS Debugger Program Preparation - High Level Assembler` panel is displayed. In this panel, specify the name of the source file and the assemble options that are used by High Level Assembler (HLASM) to assemble the program.

   If option 5 is not available, contact your system administrator.
4. Press Enter. The `High Level Assembler - Verify Selections` panel is displayed. Verify that the information on the panel is correct and then press Enter.
5. If any of the output data sets you specified do not existed, you are asked to verify the options used to create them.
6. If you specified that the processing be completed by batch, the JCL created to run the batch job is displayed. Verify that the JCL is correct, type `Submit` in the command line, press Enter and then press PF3.
7. After the processing is completed, the `High Level Assembler - View Outputs` panel is displayed. This panel displays the return code of each process completed and enables you to view, edit, or browse the input and output data sets.

To read more information about a field in any panel, place the cursor in the input field and press PF1. To read more information about a panel, place the cursor anywhere on the panel that is not an input field and press PF1.

After you assemble your program and create the EQALANGX file, you can link-edit your program.

Link-editing your program

You can link-edit your program by using your normal link-edit procedures or you can use IBM z/OS Debugger Utilities by doing the following:

Note: z/OS Debugger Utilities is not available in IBM Developer for z/OS (non-Enterprise Edition), IBM Z and Cloud Modernization Stack (Wazi Code).

1. From the `z/OS Debugger Program Preparation` panel, select option L, "Link Edit". The `z/OS Debugger Program Preparation - Link Edit` panel is displayed. In this panel, specify the input data sets and link edit options that you need the linker to use.
2. Press Enter. The `Link Edit - Verify Selections` panel is displayed. Verify that the information on the panel is correct and then press Enter.
3. If any of the output data sets you specified do not exist, you are asked to verify the options used to create them. Press Enter after you verify the options.
4. If you specified that the processing be completed by batch, the JCL created to run the batch job is displayed. Verify that the JCL is correct and press PF3.
5. After the processing is completed, the `Link Edit - View Outputs` panel is displayed. This panel displays the return code of each process completed and enables you to view, edit, or browse the input and output data sets.

To read more information about a field in any panel, place the cursor in the input field and press PF1. To read more information about a panel, place the cursor anywhere on the panel that is not an input field and press PF1.

After you link-edit your program, you can run your program and start z/OS Debugger.

Restrictions for link-editing your assembler program

z/OS Debugger cannot find the EQALANGX member when you change the name with a CHANGE link statement. For example, the message "EQALANGX debug file cannot be found for PGM1TEST" is displayed when you use the following link statements:

```
CHANGE PGMTEST1(PGM1TEST)
INCLUDE LINKLIB(PGMTEST1)
```

Chapter 8. Preparing a Db2 program

You do not need to use any special coding techniques to debug Db2 programs with z/OS Debugger.

The following sections describe the tasks you need to do to prepare a Db2 program for debugging:

1. “Processing SQL statements” on page 75.
2. “Linking Db2 programs for debugging” on page 76.
3. “Binding Db2 programs for debugging” on page 77.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
DB2® UDB for z/OS Application Programming and SQL Guide

Processing SQL statements

You must run your program through the Db2 preprocessor or coprocessor, which processes SQL statements, either before or as part of the compilation. In this section, we describe how and when each compiler uses the Db2 preprocessor or coprocessor. Then you can choose the right method so that you can debug the program with z/OS Debugger.

- If you are preparing a COBOL program using a compiler earlier than Enterprise COBOL for z/OS and OS/390 Version 2 Release 2 , use the Db2 precompiler. Then compile your program as described in the appropriate section for your programming language.
- If you are preparing a COBOL program using Enterprise COBOL for z/OS and OS/390 Version 2 Release 2 or later, do one of the following tasks:
  - Use the Db2 precompiler. Then compile your program as described in the appropriate section for your programming language.
  - Use the SQL compiler option so that the SQL statements are processed by the Db2 coprocessor during compilation. Save the program listing if you compiled with the NOSEPARATE suboption of the TEST compiler option or the separate debug file if you compiled with the SEPARATE suboption of the TEST compiler option.
- If you are preparing a PL/I program using a compiler earlier than Enterprise PL/I for z/OS and OS/390 Version 3 Release 1, use the Db2 precompiler. Then compile your program as described in the appropriate section for your programming language.
- The following table describes your options for specific PL/I compilers.

If you are using any of the following PL/I compilers:	**Choose one of the following tasks:**
– Enterprise PL/I for z/OS and OS/390 Version 3 Release 1 through Version 3 Release 4 – Enterprise PL/I for z/OS, Version 3.5 or later, and you do not specify the SEPARATE suboption of the TEST compiler option	– Use the Db2 precompiler. Save the program source files generated by the Db2 precompiler, which z/OS Debugger uses to debug your program. Then compile your program as described in the appropriate section for your programming language. – Use the PP(SQL:('*option*,...')) compiler option so that the SQL statements are processed by the Db2 coprocessor during compilation. Save the program source file that you used as input to the compiler.

- If you are preparing a program using Enterprise PL/I for z/OS, Version 3.5 or later, and you specify the SEPARATE suboption of the TEST compiler option, do one of the following tasks:

© Copyright IBM Corp. 1992, 2023

  - Use the Db2 precompiler. Compile the program source files generated by the Db2 precompiler with the appropriate compiler options, as described in "Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34, select scenario B. Save the separate debug file created by the compiler.
  - Use the PP(SQL:('*option*,...')) compiler option so that the SQL statements are processed by the Db2 coprocessor during compilation. Save the separate debug file created by the compiler.
- If you are preparing a C or C++ program using a compiler earlier than C/C++ for z/OS Version 1 Release 5, use the Db2 precompiler. Save the program source files generated by the Db2 precompiler, which z/OS Debugger uses to debug your program. Then compile your program as described in the appropriate section for your programming language.
- If you are preparing a C or C++ program using C/C++ for z/OS Version 1 Release 5 or later, do one of the following tasks:
  - Use the Db2 precompiler. Save the program source files generated by the Db2 precompiler, which z/OS Debugger uses to debug your program. Then compile your program as described in the appropriate section for your programming language.
  - Specify the SQL compiler option so that the SQL statements are processed by the Db2 coprocessor during compilation. Save the program source file that you used as input to the compiler.
- If you are using an assembler program, first run your program through the Db2 precompiler, then assemble your program using the output of the Db2 precompiler. Generate a EQALANGX file from the assembler output and save the EQALANGX file.

Important: Ensure that your program source, separate debug file, or program listing is stored in a permanent data set that is available to z/OS Debugger.

To enhance the performance of z/OS Debugger, use a large block size when you save these files. If you are using COBOL or Enterprise PL/I separate debug files, it is important that you allocate these files with the correct attributes to optimize the performance of z/OS Debugger. Use the following attributes for the PDS that contains the COBOL or PL/I separate debug file:

- RECFM=FB
- LRECL=1024
- BLKSIZE set so the system determines the optimal size

Refer to the following topics for more information related to the material discussed in this topic.

Related references
DB2 UDB for OS/390 Application Programming and SQL Guide

Linking Db2 programs for debugging

To debug Db2 programs, you must link the output from the compiler into your program load library. You can include the user runtime options module, CEEUOPT, by doing the following:

1. Find the user runtime options program CEEUOPT in the Language Environment SCEESAMP library.
2. Change the NOTEST parameter into the desired TEST parameter. For example:

```
old:  NOTEST=(ALL,*,PROMPT,INSPPREF),
new:  TEST=(,*,;,*),
```

If you are using remote debug mode, specify the TCPIP suboption, as in the following example:

```
TEST=(,,,TCPIP&&9.2404.79%8001:*)
```

Note: Double ampersand is required.

If you are using full-screen mode using a dedicated terminal without Terminal Interface Manager, specify the MFI suboption with a VTAM LU name, as in the following example:

```
Test=(,,,MFI%TRMLU001)
```

If you are using full-screen mode using the Terminal Interface Manager, specify the VTAM suboption with your user ID, as in the following example:

```
Test=(,,,VTAM%USERABCD)
```

3. Assemble the CEEUOPT program and keep the object code.
4. Link-edit the CEEUOPT object code with any program to start z/OS Debugger.

The modified assembler program, CEEUOPT, is shown below.

```
*/****************************************************************/
*/* LICENSED MATERIALS - PROPERTY OF IBM                         */
*/*                                                              */
*/* 5694-A01                                                     */
*/*                                                              */
*/* (C) COPYRIGHT IBM CORP. 1991, 2001                           */
*/*                                                              */
*/* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                 */
*/* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA ADP              */
*/* SCHEDULE CONTRACT WITH IBM CORP.                             */
*/*                                                              */
*/* STATUS = HLE7705                                             */
*/****************************************************************/
CEEUOPT  CSECT
CEEUOPT  AMODE ANY
CEEUOPT  RMODE ANY
         CEEXOPT TEST=(,*,:,*)
         END
```

The user runtime options program can be assembled with predefined TEST runtime options to establish defaults for one or more applications. Link-editing an application with this program results in the default options when that application is started.

If your system programmer has not already done so, include all the proper libraries in the SYSLIB concatenation. For example, the ISPLOAD library for ISPLINK calls, and the Db2 DSNLOAD library for the Db2 interface modules (DSN*xxxx*).

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 15, “Starting z/OS Debugger from a program,” on page 119

Binding Db2 programs for debugging

Before you can run your Db2 program, you must run a Db2 bind in order to bind your program with the relevant DBRM output from the precompiler step. No special requirements are needed for z/OS Debugger.

Chapter 9. Preparing a Db2 stored procedures program

This topic describes the information you need to collect and the steps you must take to prepare a Db2 stored procedure for debugging with z/OS Debugger. z/OS Debugger can debug stored procedures where PROGRAM TYPE is MAIN or SUB; the preparation steps are the same.

Before you begin, verify that you can use the supported debugging modes. z/OS Debugger can debug stored procedures written in assembler, C, C++, COBOL and Enterprise PL/I in any of the following debugging modes:

- remote debug
- full-screen mode using the Terminal Interface Manager
- batch

Review the topic "Creating a stored procedure" in the *Db2 Application Programming and SQL Guide* to verify that your stored procedure complies with the format and restrictions for external stored procedures. z/OS Debugger supports debugging only external stored procedures.

To prepare a Db2 stored procedure, do the following steps:

1. Verify that your Db2 system administrator has completed the tasks described in section Preparing your environment to debug a Db2 stored procedures" of *IBM z/OS Debugger Customization Guide*. The Db2 system administrator must define the address space where the stored procedure runs, give Db2 programs the appropriate RACF read authorizations, and recycle the address space so that the updates take effect.
2. If you are not familiar with the parameters used to create the Db2 stored procedure you want to debug, you can enter the SELECT statement, as illustrated in the following example, to obtain this information:

   ```
   SELECT PROGRAM_TYPE,STAYRESIDENT,RUNOPTS,LANGUAGE
      FROM SYSIBM.SYSROUTINES
      WHERE NAME='name_of_Db2_stored_procedure';
   ```

3. When you define your stored procedure, verify the following items:
   - Specify the correct value for the LANGUAGE parameter and the PROGRAM TYPE parameter. For C, C++, COBOL or Enterprise PL/I, the PROGRAM TYPE can be either MAIN or SUB. For assembler, the PROGRAM TYPE must be MAIN.
   - For stored procedures of program type SUB, review the following options:
     - If you plan to specify the TEST runtime options through the Language Environment EQAD3CXT exit routine, specify `STAY RESIDENT NO`.
     - If you plan to specify the TEST runtime options through the Db2 catalog, you can specify either `YES` or `NO` for `STAY RESIDENT`.
4. Compile or assemble your program, as described in Part 2, “Preparing your program for debugging,” on page 21. For Enterprise PL/I programs, also specify the RENT compiler option.
5. Review the following list to determine how to specify the TEST runtime options:
   - For stored procedures of program type MAIN, you can specify the TEST runtime option either through the Language Environment EQAD3CXT exit routine, or through the Db2 catalog. If you use both methods, the Language Environment EQAD3CXT exit routine take precedence over the Db2 catalog.
   - For stored procedures of program type SUB, you can specify the TEST runtime option either through the Language Environment EQAD3CXT exit routine or through the Db2 catalog. If you choose to use

© Copyright IBM Corp. 1992, 2023

the Language Environment EQAD3CXT exit routine, you must specify the NOTEST runtime option for the RUN OPTIONS parameter when you define the stored procedure.

6. To specify the TEST runtime options through the Language Environment EQAD3CXT exit routine, prepare a copy of the EQAD3CXT user exit as described in Chapter 12, “Specifying the TEST runtime options through the Language Environment user exit,” on page 97.

   Remember that if you want to debug an *existing* stored procedure of program type SUB, you must modify the stored procedure so that it uses the NOTEST runtime option for the RUN OPTIONS parameter. The following example shows how to use the ALTER PROCEDURE statement to make this modification:

   ```
   ALTER PROCEDURE name_of_Db2_stored_procedure RUN OPTIONS 'NOTEST';
   ```

7. To specify the TEST runtime options through the Db2 catalog, do the following steps:

   a. If you have not created the stored procedure, write the stored procedure using the CREATE PROCEDURE statement. You can use the following example as a guide:

      ```
      CREATE PROCEDURE SPROC1
         LANGUAGE COBOL
         EXTERNAL NAME SPROC1
         PARAMETER STYLE GENERAL
         WLM ENVIRONMENT WLMENV1
         RUN OPTIONS 'TEST(,,,TCPIP&9.112.27.99%8001:*)'
         PROGRAM TYPE SUB;
      ```

      This example creates a stored procedure for a COBOL program called SPROC1, the program type is SUB, it runs in a WLM address space called WLMENV1, and it is debugged in remote debug mode.

   b. If you need to modify an existing stored procedure, use the ALTER PROCEDURE statement. You can use the following example as a guide:

      The IP address for the remote debugger changed from 9.112.27.99 to 9.112.27.21. To modify the stored procedure, enter the following statement:

      ```
      ALTER PROCEDURE name_of_Db2_stored_procedure
         RUN OPTIONS 'TEST(,,,TCPIP&9.112.27.21%8001:*)';
      ```

   c. Verify that the stored procedure is defined correctly by entering the SELECT statement. For example, you can enter the following SELECT statement:

      ```
      SELECT * FROM SYSIBM.SYSROUTINES;
      ```

Chapter 10. Preparing a CICS program

To prepare a CICS program for debugging, you must do the following tasks:

1. Complete the program preparation tasks for COBOL, PL/I, C, C++, assembler, or LangX COBOL, as described in the following sections:

   "Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27
   "Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34
   "Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
   "Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44
   Chapter 7, "Preparing an assembler program," on page 71
   Chapter 6, "Preparing a LangX COBOL program," on page 67
2. Determine if your site uses CADP or DTCN debugging profiles and verify that your system has been configured to use the chosen debugging profile.
3. Determine if you need to link edit EQADCCXT into your program by reviewing the instructions in "Link-editing EQADCCXT into your program" on page 81.
4. Do one of the following tasks:
   - If your site is using DTCN debugging profiles, create and store a DTCN debugging profile. Instructions for creating a DTCN debugging profile are in "Creating and storing a DTCN profile" on page 82.
   - If you are using CICS Transaction Server for z/OS Version 2 Release 3 or later and your site uses CADP to manage debugging profiles, create and store a CADP debugging profile. See "Creating and storing debugging profiles with CADP" on page 92 for more information about using CADP.

Link-editing EQADCCXT into your program

z/OS Debugger provides an Language Environment CEEBXITA assembler exit called EQADCCXT to help you activate, by using the DTCN transaction, a debugging session under CICS. You do not need to use this exit if you are running any of the following options:

- You are running under CICS Transaction Server for z/OS Version 2 Release 3 or later and you use the CADP transaction to define debug profiles.
- You are using the DTCN transaction and you are debugging non-Language Environment Assembler programs.
- You are using the DTCN transaction and you are debugging COBOL programs, or PL/I programs in the following situation:
  - Compiled with Enterprise PL/I for z/OS, Version 3 Release 4 with the PTF for APAR PK03264 applied, or later

When you use EQADCCXT, be aware of the following conditions:

- If your site does not use an Language Environment assembler exit (CEEBXITA), then link-edit member EQADCCXT, which contains the CSECT CEEBXITA and is in library *hlq*.SEQAMOD, into your main program.
- If your site uses an existing CEEBXITA, the EQADCCXT exit provided by z/OS Debugger must be merged with it. The source for EQADCCXT is in *hlq*.SEQASAMP(EQADCCXT). Link the merged exit into your main program.

After you link-edit your program, use the DTCN transaction to create a profile that specifies the combination of resources that you want to debug. See "Creating and storing a DTCN profile" on page 82.

© Copyright IBM Corp. 1992, 2023

Creating and storing a DTCN profile

You can create and store DTCN profiles, or CICS profiles, in the following ways:

- By using the DTCN transaction. The rest of the information in these topics describe how to do this.
- By creating a CICS profile from the **z/OS Debugger Profiles** view. For more information about creating a debug configuration for a CICS application, see topic "Creating a debug profile for a CICS application" in IBM Documentation.

The DTCN transaction stores debugging profiles in a repository. The repository can be either a CICS temporary storage queue or a VSAM file. The following list describes the differences between using a CICS temporary storage queue or a VSAM file:

- If you don't log on to CICS or you log on as the default user, you cannot use a VSAM file. You must use a CICS temporary storage queue.
- If you use a CICS temporary storage queue, the profile will be deleted if the terminal that created the profile has been disconnected or the CICS region is terminated. If you use a VSAM file, the profile will persist through disconnections or CICS region restarts.
- If you use a CICS temporary storage queue, there can be only one profile on a single terminal. If you use a VSAM file, there can be multiple profiles, each created by a different user, on a single terminal.

z/OS Debugger determines which storage method is used based on the presence of a debugging profile VSAM file. If z/OS Debugger finds a debugging profile VSAM file allocated to the CICS region, it assumes you are using a VSAM file as the repository. If it doesn't find a debugging profile VSAM file, it assumes you are using a CICS temporary storage queue as the repository. See the *IBM z/OS Debugger Customization Guide* or contact your system programmer for more information about how the VSAM files are created and managed.

If the repository is a temporary storage queue, each profile is retained in the repository until one of the following events occurs:

- The profile is explicitly deleted by the terminal that entered it.
- DTCN detects that the terminal which created the profile has been disconnected.
- The CICS region is terminated.

If the repository is a VSAM file, each profile is retained until it is explicitly deleted. The DTCN transaction uses the user ID to identify a profile. Therefore, each user ID can have only one profile stored in the VSAM file.

Profiles are either active or inactive. If a profile is active, DTCN tries to match it with a transaction that uses the resources specified in the profile. DTCN does not try to match a transaction with an inactive profile. To make a profile active or inactive, use the **z/OS Debugger CICS Control - Primary Menu** panel (the main DTCN panel) to make the profile active or inactive, then save it. If the repository is a VSAM file, when DTCN detects that the terminal is disconnected, it makes the profile inactive.

To create and store a DTCN profile:

1. Log on to a CICS terminal and enter the transaction ID DTCN. The DTCN transaction displays the main DTCN screen, z/OS Debugger CICS Control - Primary Menu, shown below.

```
DTCN               z/OS Debugger CICS Control - Primary Menu            S07CICPD
                         * VSAM storage method *    1
Select the combination of resources to debug (see Help for more information)
 Terminal Id     ==> 0090
 Transaction Id  ==>
 LoadMod::>CU(s) ==>          ::>               ==>          ::>
                 ==>          ::>               ==>          ::>
                 ==>          ::>               ==>          ::>
                 ==>          ::>               ==>          ::>
 User Id         ==> CICSUSER
 NetName         ==>
 IP Name/Address ==>
Select type and ID of debug display device
 Session Type    ==> MFI                MFI, TCP, DIR, DTC, DBM
 Port Number     ==>                    TCP Port
 Display Id      ==> 0090

Generated String:  TEST(ERROR,'*',PROMPT,'MFI%0090:*')

Repository String: No string currently saved in repository

Profile Status:    No Profile Saved. Press PF4 to save current settings.

PF1=HELP 2=GHELP 3=EXIT 4=SAVE 5=ACT/INACT 6=DEL 7=SHOW 8=ADV 9=OPT 10=CUR TRM
```

Line 1 displays a message to indicate that DTCN will store the profile in a temporary storage queue or in a VSAM file. Some of the entry fields are filled in with values from one of the following sources:

- If the temporary storage queue is the type of repository, the fields are filled in with default values that start z/OS Debugger, in full-screen mode, for tasks running on this terminal.
- If a VSAM file is the type of repository and a profile exists for the current user, the fields are filled in with data found in that profile. If a VSAM file is the type of repository and a profile does not exist for the current user, the fields are filled in with default values that start z/OS Debugger, in full-screen mode, for tasks running on this terminal.

If you do not want to change these fields, you can skip the next two steps and proceed to step "4" on page 83. If you want to change the settings on this panel, continue to the next step.

2. Specify the combination of resources that identify the transaction or program that you want to debug. For more information about these fields, do one of the following tasks:
   - Read "Description of fields on the DTCN Primary Menu screen" on page 86.
   - Place the cursor next to the field and press PF1 to display the online help.
3. Specify the type of debugging session you want to run and the ID of the device that displays the debugging session. For more information about these fields, do one of the following tasks:
   - Read "Description of fields on the DTCN Primary Menu screen" on page 86.
   - Place the cursor next to the field and press PF1 to display the online help.
4. Specify the TEST runtime options, other runtime options, commands file, preferences file, and EQAOPTS file that you want to use for the debugging session by pressing PF9 to display the secondary options menu, which looks like the following example:

```
DTCN               z/OS Debugger CICS Control - Menu 2              S07CICPD

Select z/OS Debugger options
Test Option     ==> TEST               Test/Notest
Test Level      ==> ERROR              All/Error/None
Commands File   ==> *
Prompt Level    ==> PROMPT
Preference File ==> *

EQAOPTS File    ==>

Any other valid Language Environment options
==>



PF1=HELP 2=GHELP 3=RETURN
```

Some of the entry fields are filled in with default values that start z/OS Debugger, in full-screen mode, for tasks running on this terminal. If you do not want to change the defaults, you can skip the rest of this step and proceed to step “5” on page 84. If you want to change the settings on this panel, continue with this step.

5. Press PF3 to return to the main DTCN panel.
6. If you want to use data passed through COMMAREA or containers to help identify transactions and programs that you want to debug, press PF8. The **Advanced Options** panel is displayed, which looks like the following example:

```
DTCN               z/OS Debugger CICS Control - Advanced Options         S07CICPD

Select advanced program interruption criteria:

Commarea Offset  ==> 0
Commarea Data    ==>

Container Name   ==>
Container Offset ==> 0
Container Data   ==>

URM Debugging    ==> NO




Default offset and data representation is decimal/character.
See Help for more information.


PF1=HELP 2=GHELP 3=RETURN
```

   You can specify data in the COMMAREA or containers, but not both. You can also use this panel to indicate whether you want to debug user replaceable modules (URMs). For more information about these fields, do one of the following tasks:

   - Read “Description of fields on the DTCN Primary Menu screen” on page 86.
   - Place the cursor next to the field and press PF1 to display the online help.

7. Press PF3 to return to the main DTCN panel.
8. Press PF4 to save the profile. DTCN performs data verification on the data that you entered in the DTCN panel. When DTCN discovers an error, it places the cursor in the erroneous field and displays a message. You can use context-sensitive help (PF1) to find what is wrong with the input.
9. Press PF5 to change the status from active to inactive, or from inactive to active. A profile has three possible states:

   No profile saved
   A profile has not yet been created for this terminal.

   Active
   The profile is active for pattern matching.

   Inactive
   Pattern matching is skipped for this profile.

10. After you save the profile in the repository, DTCN shows the saved TEST string in the display field Repository String. If you are satisfied with the saved profile, press PF3 to exit DTCN.

Now, any tasks that run in the CICS system and match the resources that you specified in the previous steps will start z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Displaying a list of active DTCN profiles and managing DTCN profiles” on page 85

Related references

"Description of fields on the DTCN Primary Menu screen" on page 86

Description of the DTCD transaction in *IBM z/OS Debugger Customization Guide*

Displaying a list of active DTCN profiles and managing DTCN profiles

To display all of the active DTCN profiles in the CICS region, do the following steps:

1. If you have not started the DTCN transaction, Log on to a CICS terminal and enter the transaction ID DTCN. The DTCN transaction displays the main DTCN screen, z/OS Debugger CICS Control - Primary Menu.
2. Press PF7. The z/OS Debugger CICS Control - All Sessions screen displays, shown below.

```
 DTCN               z/OS Debugger CICS Control - All Sessions             S07CICPD

Overtype "_" with "D" to delete, "A" to activate, "I" to inactivate a profile.

  Owner___ Sta Term Tran__ User Id___ NetName_ Applid__ Display Id___________

_ 0090     ACT 0090 TRN1   USER1      0072     S07CICPD 0090

           LoadMod::>CU(s) CIC9060  ::> CS9060          CBLAC?3  ::> *9361
                           ________ ::> ________        ________ ::> ________
                           ________ ::> ________        ________ ::> ________
                           ________ ::> ________        ________ ::> ________
           IP Name/Addr  ____________________________________________________
```

The column titles are defined below:

Owner
The ID of the terminal that created the profile by using DTCN.

Sta
Indicates if the profile is active (ACT) or inactive (INA).

Term
The value that was entered on the main DTCN screen in the **Terminal Id** field.

Tran
The value that was entered on the main DTCN screen in the **Transaction Id** field.

User Id
The value that was entered on the main DTCN screen in the **User Id** field.

Netname
The value the entered on the main DTCN screen in the **Netname** field.

Applid
The application identifier associated with this profile.

Display Id
Identifies the target destination for z/OS Debugger information.

LoadMod(s)
The values that were entered on the main DTCN screen in the **LoadMod(s)** field.

CU(s)
The values that were entered on the main DTCN screen in the **CU(s)** field.

IP Name/Addr
The value that was entered on the main DTCN screen in the **IP Name/Address** field.

DTCN also reads the Language Environment NOTEST option supplied to the CICS region in CEECOPT or CEEROPT. You can supply suboptions, such as the name of a preferences file, with the NOTEST option to supply additional defaults to DTCN.

3. To delete a profile, move your cursor to the underscore character (_) that is next to the profile you want to delete. Type in "D" and then press Enter.

4. To make a profile inactive, move your cursor to the underscore character (_) that is next to the profile you want to make inactive. Type in "I" and then press Enter.
5. To make a profile active, move your cursor to the underscore character (_) that is next to the profile you want to make active. Type in "A" and then press Enter.
6. To leave this panel and return to the DTCN primary menu, press PF3.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Creating and storing a DTCN profile" on page 82

Description of fields on the DTCN Primary Menu screen

This topic describes the fields that are displayed on the DTCN Primary Menu screen.

The following list describes the resources you can specify to help identify the program or transaction that you want to debug:

Terminal Id

Specify the CICS terminal identifier associated with the transaction you want to debug. By default, DTCN sets the ID by one of the following rules:

- If the type of repository is a VSAM file and the current user ID has a saved profile, DTCN fills in the field with the terminal ID that is in the repository. You can change the terminal ID to the ID of the terminal you are currently running on, by placing your cursor on the terminal ID field and then pressing PF10. Press PF4 to save the profile with this new value.
- If the type of repository is a VSAM file and the current user ID does not have a saved profile, the terminal ID field is filled in with the ID of the terminal you are currently running on.
- If the type of repository is a temporary storage queue, the terminal ID field is filled in with the ID of the terminal you are currently running on.
- If the CICS transaction or program that you want to debug is not associated with a specific terminal (for example, the request to start a debug session comes from a browser), make this field blank.

If YES is specified for the EQAOPTS `DTCNFORCETERMID` command, you must specify a terminal identifier. To learn about the EQAOPTS `DTCNFORCETERMID` command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

Transaction Id

Specify the CICS transaction to debug. If you specify a transaction ID without any other resource, z/OS Debugger is started **every** time any of the following situations occurs:

- You run the transaction.
- The first program run by the transaction is started.
- Any other user runs the transaction.
- Any enabled DFH* module is the first program run by the transaction.

To start z/OS Debugger at the desired program that the transaction runs, specify the program name in the Program Id(s) field.

If YES is specified for the EQAOPTS `DTCNFORCETRANID` command, you must specify a transaction ID. To learn about the EQAOPTS `DTCNFORCETRANID` command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

LoadMod::>CU(s)

Specify the resource pair or pairs, consisting of a load module name and a compile unit (CU) name that you want to debug. Type in the load module name after the ==> and the corresponding CU name after the `::>`. You can specify any of the following names:

LoadMod

The name of a load module that you want to debug. The load module must comply with the following requirements:

- For z/OS Debugger initialization, the load module can be any CICS load module if it is invoked as an Language Environment enclave or over a CICS Link Level. This includes the following types of load modules:
  - The initial load module in a transaction.
  - A load module invoked by CICS LINK or XCTL.
- Any non-Language Environment assembler load module which is loaded through an `EXEC CICS LOAD` command.

CU

The name of the compile unit (CU) that you want to debug. The CU must comply with the following requirements:

- Any CICS CU if it is invoked as an Language Environment enclave or over a CICS Link Level. This includes the following types of CUs:
  - The initial CU in a transaction
  - A CU invoked by CICS LINK or XCTL
- Any COBOL CU, even if it is a CU within a composite load module invoked by a static CALL or it is a CU in a load module that is invoked by a dynamic CALL.
- Any Enterprise PL/I for z/OS Version 3 Release 4 CU (with the PTF for APAR PK03264 applied), or later, even if it is a CU within a composite load module invoked by a static CALL or it is a CU in a load module that is invoked by a dynamic CALL.
- Any non-Language Environment assembler CU which is loaded through an `EXEC CICS LOAD` command.

Usage Notes®:

- If you specify a LoadMod and leave the corresponding CU field blank, the CU field defaults to an asterisk (*).
- If you specify a CU and leave the corresponding LoadMod field blank, the LoadMod field defaults to an asterisk (*).
- If you leave all LoadMod and CU fields blank and you set the Prompt Level on the "z/OS Debugger CICS Control - Menu 2" to PROMPT, z/OS Debugger initializes for the first program invoked.
- If you migrate from a version of z/OS Debugger prior to Version 10.1, you can obtain the same behavior produced by the `DTCN Program Id` resource by using the `LoadMod::>CU` resource pair and specifying only the CU resource. The LoadMod resource defaults to an asterisk (*).
- You can specify wildcard characters (*) and (?).
- If z/OS Debugger was started by another program before the `EXEC CICS LOAD` command that starts this non-Language Environment assembler program, you need to enter one of the following commands so that z/OS Debugger gains control of this program:
  - `LDD`
  - `SET ASSEMBLER ON`
  - `SET DISASSEMBLY ON`
- When you specify a CU for C/C++ and Enterprise PL/I programs (languages that use a fully qualified data set name as the compile unit name), you must specify the correct part of the compile unit

name in the CU field. Use the following rules to determine which part of the compile unit name you need to specify:

  - If you are using a PDS or PDSE, you must specify the member name. For example, if the compile unit names are DEV1.TEST.ENTPLI.SOURCE(ABC) and DEV1.TEST.C.SOURCE(XYZ), you must specify ABC and XYZ in the program ID field.
  - If you are using a sequential data set, specify one of the following:
    - The last qualifier of the sequential data set. For example, if the compile unit names are DEV1.TEST.ENTPLI.SOURCE.ABC and DEV1.TEST.C.SOURCE.XYZ, you must specify ABC and XYZ in the program ID field.
    - Wildcards. For example, if the compile unit names are DEV1.TEST.ENTPLI.ABC.SOURCE and DEV1.TEST.C.XYZ.SOURCE, you must specify *ABC* and *XYZ* in the program ID field.
  - If you compiled your PL/I program with the following compiler and it is running in the following environment, you need to use the package name or the main procedure name:
    - Enterprise PL/I for z/OS, Version 3.5, with the PTFs for APARs PK35230 and PK35489 applied, or Enterprise PL/I for z/OS, Version 3.6 or later
    - Language Environment, Version 1.6 through 1.8 with the PTF for APAR PK33738 applied, or later

- Specifying a CICS program in the LoadMod::>CU field is similar to setting a breakpoint by using the `AT ENTRY` command and z/OS Debugger stops each time you enter LoadMod::>CU.
- If z/OS Debugger is already running and it cannot find the separate debug file, then z/OS Debugger does not stop at the CICS program specified in the LoadMod::>CU field. Use the `AT APPEARANCE` or `AT ENTRY` command to stop at this CICS program.
- If YES is specified for the EQAOPTS `DTCNFORCELOADMODID` command, you must specify a value for the LoadMod field. To learn about the EQAOPTS `DTCNFORCELOADMODID` command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.
- If YES is specified for the EQAOPTS `DTCNFORCEPROGID` or `DTCNFORCECUID` commands, you must specify a value for the CU field. To learn about the EQAOPTS `DTCNFORCEPROGID` or `DTCNFORCECUID` commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

User Id

Specify the user identifier associated with the transaction you want to debug. The following list can help you decide what to enter in this field:

- If the user identifier is the same one that is currently running DTCN, use the default user identifier.
- If the user identifier is different than the one currently running DTCN and you know the user identifier, enter that user identifier.
- If you do not know the user identifier or the transaction is not associated with a user identifier, specify the wild character or blanks.

If YES is specified for the EQAOPTS `DTCNFORCEUSERID` command, you must specify a user identifier. To learn about the EQAOPTS `DTCNFORCEUSERID` command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

NetName

Specify the four character name of a CICS terminal or a CICS system that you want to use to run your debugging session. This name is used by VTAM to identify the CICS terminal or system.

If YES is specified for the EQAOPTS `DTCNFORCENETNAME` command, you must specify a value for the **NetName** field. To learn about the EQAOPTS `DTCNFORCENETNAME` command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

IP Name/Address
The client IP name or IP address that is associated with a CICS application. All IP names are treated as upper case. Wildcards (* and ?) are permitted. z/OS Debugger is invoked for every task that is started for that client.

If YES is specified for the EQAOPTS DTCNFORCEIP command, you must specify an IP address. To learn about the EQAOPTS DTCNFORCEIP command, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

The following list describes the fields that you can use to indicate which type of debugging session you want to run.

Session Type
Select one of the following options:

MFI
Indicates that z/OS Debugger initializes on a 3270 type of terminal.

TCP
Indicates that you want to interact with z/OS Debugger using a remote debugger in Debug Tool compatibility mode connected with a TCP/IP host name or address.

z/OS Debugger is progressing towards one remote debug mode based on Debug Tool compatibility mode. In support of this direction, **TCP** is the preferred option to debug CICS transactions with a remote IDE.

DIR
Indicates that you want to interact with z/OS Debugger using a remote debugger in standard mode connected with a TCP/IP host name or address.

Standard mode is not available in IBM Z and Cloud Modernization Stack (Wazi Code).

DTC
Indicates that you want to interact with z/OS Debugger using a remote debugger in Debug Tool compatibility mode connected with a z/OS Debugger Debug Manager userid.

z/OS Debugger is progressing towards one remote debug mode based on Debug Tool compatibility mode. In support of this direction, **DTC** is the preferred option to debug CICS transactions using Debug Manager with a remote IDE.

DBM
Indicates that you want to interact with z/OS Debugger using a remote debugger in standard mode connected with a z/OS Debugger Debug Manager userid.

Standard mode is not available in IBM Z and Cloud Modernization Stack (Wazi Code).

RDS
Indicates that you want to start a debug session with IBM Z Open Debug using Remote Debug Service.

Port Number
Specifies the TCP/IP port number that the debug daemon is listening for debug or code coverage sessions. The debug daemon default port is 8001. If you entered DTC or DBM in the Session Type field, this field must be left blank.

Note: Code coverage is not supported by IBM Z and Cloud Modernization Stack (Wazi Code).

Display Id
Identifies the target destination for z/OS Debugger.

If you entered DTC or DBM in the **Session Type** field, enter the userid that your workstation is using to connect to the z/OS remote system.

If you entered TCP or DIR in the **Session Type** field, determine the IP address or host name of the workstation that is running the remote debugger. Change the value in the **Display Id** field by doing the following steps:

1. Place your cursor on the **Display Id** field.

2. Type in the IP address or host name of the workstation that is running the remote debugger.
3. To save the profile with this new value, press PF4.

If you entered MFI in the **Session Type** field, DTCN fills in the **Display Id** field according to the following rules:

- If the type of repository is a VSAM file and the current user ID has a saved profile, DTCN fills in the field with the display ID that is in the repository.
- If the type of repository is a VSAM file and the current user ID does not have a saved profile, DTCN fills in the field with the ID of the terminal you are currently running on.
- If the type of repository is a temporary storage queue, DTCN fills in the field with the ID of the terminal you are currently running on.

You can use one of the following terminal modes to display z/OS Debugger on a 3270 terminal:

- Single terminal mode: z/OS Debugger and the application program share the same terminal. To use this mode, enter the ID of the terminal being used by your application program or move the cursor to the **Display ID** field and press PF10.
- Screen control mode: z/OS Debugger displays its screens on a terminal which is running the DTSC transaction. To use this mode, start the DTSC transaction on a terminal and specify that terminal's ID in the **Display ID** field.
- Separate terminal mode: z/OS Debugger displays its screens on a terminal which is available for use (not associated with any transaction) and can be located by CICS. To use this mode, specify the terminal's ID in the **Display ID** field.

Description of fields on the DTCN Menu 2 screen

The following list describes the fields that you can use to specify the TEST runtime options, other runtime options, commands file, and preferences file that you want to use for the debugging session:

Test Option
TEST/NOTEST specifies the conditions under which z/OS Debugger assumes control during the initialization of your application.

Test Level
ALL/ERROR/NONE specifies what conditions need to be met for z/OS Debugger to gain control.

Commands File
A valid fully qualified data set name that specifies the commands file for this run. Do not enclose the name of the data set in quotation marks (") or apostrophes ('). The CICS region must have read authorization to the commands file.

If you leave this field blank and have a value for a default user commands file set through the EQAOPTS COMMANDSDSN command, z/OS Debugger does the following tasks to find a commands file:

1. z/OS Debugger constructs the name of a data set from the naming pattern specified in the command.
2. z/OS Debugger locates the data set.
3. If the data set contains a member with a name that matches the name of the initial load module in the first enclave, it processes that member as a commands file.

If you do not want specify a commands file, and want to prevent z/OS Debugger from using the file specified by the EQAOPTS COMMANDSDSN command, specify NULLFILE for the commands file.

To learn how to specify the EQAOPTS COMMANDSDSN command, see the topic "EQAOPTS commands" in either the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

Prompt Level
Specifies whether z/OS Debugger is started at Language Environment initialization.

Preferences File
A valid fully qualified data set name that specifies the preferences file for this run. Do not enclose the name of the data set in quotation marks (") or apostrophes ('). The CICS region must have read authorization to the preferences file.

If you leave this field blank and have a value for a default user preferences file set through the EQAOPTS PREFERENCESDSN command, z/OS Debugger does the following tasks to find a preferences file:

1. z/OS Debugger constructs the name of a data set from the naming pattern specified in the command.
2. z/OS Debugger locates the data set and processes it as a preferences file.

If you do not want to specify a preferences file, and want to prevent z/OS Debugger from using the file specified by the EQAOPTS PREFERENCESDSN command, specify NULLFILE for the preferences file.

To learn how to specify the EQAOPTS PREFERENCESDSN command, see the topic "EQAOPTS commands" in either the *IBM z/OS Debugger Customization Guide* or *IBM z/OS Debugger Reference and Messages*.

EQAOPTS File
A valid fully qualified data set name that specifies the EQAOPTS file for this run. Do not enclose the name of the data set in quotation marks (") or apostrophes ('). The CICS region must have read authorization to the EQAOPTS file.

Any other valid Language Environment Options
You can change any Language Environment option that your site has defined as overrideable except the STACK option. For additional information about Language Environment options, see *z/OS Language Environment Programming Reference* or contact your CICS system programmer.

Description of fields on the DTCN Advanced Options screen

The following list describes the fields that you can use to specify the data passed through COMMAREA or containers that can help identify transactions and programs that you want to debug:

Commarea offset
Specifies the offset of data within a commarea passed to a program on invocation. You can specify the offset in decimal format (for example, 13) or in hexadecimal format (for example, X'D'). If you specify data in hexadecimal format, you must specify an even number of hexadecimal digits.

Commarea data
Specifies the data within a commarea that is passed to a program on invocation. You can specify the data in character format (for example, "ABC") or in hexadecimal format (for example, X'C1C2C3').

Container name
Specifies the name of a container within the current channel passed to a program on invocation. Container names are case sensitive.

Container offset
Specifies the offset of data in the named container passed to a program in the current channel on invocation. You can specify the offset in decimal format (for example, 13) or in hexadecimal format (for example, X'D').

Container data
Specified the data in the named container passed to a program in the current channel on invocation. You can specify the data in character format (for example, "ABC") or in hexadecimal format (for example, X'C1C2C3'). If you specify data in hexadecimal format, you must specify an even number of hexadecimal digits.

URM debugging
Specifies whether you want z/OS Debugger to include the debugging of URMs as part of the debug session. Choose from the following options:

YES
z/OS Debugger debugs URMs which match normal z/OS Debugger debugging criteria.

NO
z/OS Debugger excludes URMs form debugging sessions.

Creating and storing debugging profiles with CADP

CADP is an interactive transaction supplied by CICS Transaction Server for z/OS Version 2 Release 3, or later. CADP helps you maintain persistent debugging profiles. These profiles contain a pattern of CICS resource names that identify a task that you want to debug. When CICS programs are started, CICS tries to match the executing resources to find a profile whose resources match those that are specified in a CADP profile. During this pattern matching, CICS selects the best matching profile, which is the one with greatest number of resources that match the active task.

Before using CADP, verify that you have done the following tasks:

- Compiled and linked your program as described in Chapter 10, “Preparing a CICS program,” on page 81.
- Verified that your site uses CADP and that all the tasks required to customize z/OS Debugger so that it can debug CICS programs described in *IBM z/OS Debugger Customization Guide* are completed. In particular, verify that the DEBUGTOOL system initialization parameter is set to YES so that z/OS Debugger uses the CADP profile repository instead of the DTCN profile repository to find a matching debugging profile.

See *CICS Supplied Transactions* for instructions on how to use the CADP utility transaction. If you are going to debug user-replaceable modules (URMs), specify `ENVAR("INCLUDEURM=YES")` in the **Other Language Environment Options** field.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
CICS Application Programming Guide for a description of debugging profiles.

Starting z/OS Debugger for non-Language Environment programs under CICS

You can start z/OS Debugger to debug a program that does not run in the Language Environment run time by using the existing debug profile maintenance transactions DTCN and CADP. You must use DTCN with versions of CICS prior to CICS Transaction Server for z/OS Version 2 Release 3.

To debug CICS non-Language Environment programs, the z/OS Debugger non-Language Environment Exits must have been previously started.

To debug non-Language Environment assembler programs or non-Language Environment COBOL programs that run under CICS, you must start the required z/OS Debugger global user exits before you start the programs. z/OS Debugger provides the following global user exits to help you debug non-Language Environment applications: XPCFTCH, XEIIN, XEIOUT, XPCTA, and XPCHAIR. The exits can be started by using either the DTCX transaction (provided by z/OS Debugger), or using a PLTPI program that runs during CICS region startup. DTCXXO activates the non-Language Environment Exits for z/OS Debugger in CICS. DTCXXF inactivates the non-Language Environment Exits for z/OS Debugger in CICS.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
IBM z/OS Debugger Customization Guide

Passing runtime parameters to z/OS Debugger for non-Language Environment programs under CICS

When you define your debugging profile using the DTCN Options Panel (PF9) or the CADP Create/Modify Debugging Profile Panel, you can pass a limited set of runtime options that will take effect during your debugging session when you debug programs that do not run in Language Environment. You can pass the following runtime options:

- TEST/NOTEST: must be TEST
- TEST LEVEL: must be ALL
- Commands file
- Prompt Level: must be PROMPT
- Preferences file
- You can also specify the following runtime options in a TEST string:
  - NATLANG: to specify the National Language used to communicate with z/OS Debugger
  - COUNTRY: to specify a Country Code for z/OS Debugger
  - TRAP: to specify whether z/OS Debugger is to intercept Abends

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

Chapter 11. Preparing an IMS program

To prepare an IMS program, do the following tasks:

1. Verify that Chapter 4, “Planning your debug session,” on page 25 and Chapter 5, “Updating your processes so you can debug programs with z/OS Debugger,” on page 59 have been completed.
2. Contact your system programmer to find out the preferred method for starting z/OS Debugger and which of the following methods you need to use to specify TEST runtime options:
   - Specifying the TEST runtime options in a data set, which is then extracted by a customized version of the Language Environment user exit routine CEEBXITA. See Chapter 12, “Specifying the TEST runtime options through the Language Environment user exit,” on page 97 for instructions.
   - Specifying the TEST runtime options in a CEEUOPT (application level, which you link-edit to your application program) or CEEROPT module, (region level). See “Starting z/OS Debugger under IMS by using CEEUOPT or CEEROPT” on page 95 for instructions.
   - Specifying the TEST runtime options through the EQASET transaction for non-Language Environment assembler programs running in IMS TM. See “Running the EQASET transaction for non-Language Environment IMS MPPs” on page 342 for instructions.
   - “Managing runtime options for IMSplex users by using IBM z/OS Debugger Utilities” on page 95.

Starting z/OS Debugger under IMS by using CEEUOPT or CEEROPT

You can specify your TEST runtime options by using CEEUOPT (which is an assembler module that uses the CEEXOPT macro to set application level defaults, and is link-edited into an application program) or CEEROPT (which is an assembler module that uses the CEEXOPT macro to set region level defaults). Every time your application program runs, z/OS Debugger is started.

To use CEEUOPT to specify your TEST runtime options, do the following steps:

1. Code an assembler program that includes a CEEXOPT macro invocation that specifies your application program's runtime options.
2. Assemble the program.
3. Link-edit the program into your application program by specifying an INCLUDE LibraryDDname(CEEUOPT-member name)
4. Place your application program in the load library used by IMS.

To use CEEROPT to specify your TEST runtime options, do the following steps:

1. Code an assembler program that includes a CEEXOPT macro invocation that specifies your region's runtime options.
2. Assemble the program.
3. Link-edit the program into a load module named CEEROPT by specifying an INCLUDE LibraryDDname(CEEROPT-member name)
4. Place the CEEROPT load module into the load library used by IMS.

Managing runtime options for IMSplex users by using IBM z/OS Debugger Utilities

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

This topic describes how to add, delete, or modify TEST runtime options that are stored in the IMS Language Environment runtime parameter repository. To manage the items in this repository, do the following steps:

© Copyright IBM Corp. 1992, 2023

1. From the main IBM z/OS Debugger Utilities panel (EQA@PRIM), type 4 in the Option line and press Enter.
2. In the Manage IMS Programs panel (EQAPRIS), type 1 in the Option line and press Enter.
3. In the Manage LE Runtime Options in IMS panel (EQAPRI), type in the IMSplex ID and optional qualifiers. IBM z/OS Debugger Utilities uses this information to search through the IMS Language Environment runtime parameter repository and find the entries that most closely match the information you typed in. You can use wild cards (* and %) to increase the chances of a match. After you type in your search criteria, press Enter.
4. In the Edit LE Runtime Options Entries in IMS panel (EQAPRIM), a table displays all the entries found in the IMS Language Environment runtime parameter repository that most closely match your search criteria. You can do the following tasks in this panel:
   - Delete an entry.
   - Add a new entry.
   - Edit an existing entry.
   - Copy an existing entry.

   For more information about a command or field, press PF1 to display a help panel.
5. After you finish making your changes, press PF3 to save your changes and close the panel that is displayed. If necessary, press the PF3 repeatedly to close other panels until you reach the Manage IMS Programs panel (EQAPRIS).

Setting up the DFSBXITA user exit routine

To make the debug session use the options you specified in the Manage LE Runtime Options in IMS function, you must use the DFSBXITA user exit supplied by IMS. This exit contains a copy of the Language Environment CEEBXITA user exit that is customized for IMS. The DFSBXITA user exit either replaces the exit supplied by Language Environment in CEEBINIT, or is placed in your load module.

- To make the user exit available installation-wide, do a replace link edit of the IMS CEEBXITA into the CEEBINIT load module in your system *hlq*.SCEERUN Language Environment runtime library.
- To make the user exit available region-wide, copy the CEEBINIT in your *hlq*.SCEERUN library into a private library, and then do a replace link edit of the IMS CEEBXITA into the CEEBINIT load module in your private library. Then place your private library in the STEPLIB DD concatenation sequence before the system *hlq*.SCEERUN data set in the MPR region startup job.
- To make the user exit available to a specific application, link the IMS CEEBXITA into your load module. The user exit runs only when the application is run.

The following sample JCL describes how to do a replace link edit of the IMS CEEBXITA into a CEEBINIT load module:

```
INCLUDE MYOBJ(CEEBXITA) 1
REPLACE CEEBXITA
INCLUDE SYSLIB(CEEBINIT)
ORDER CEEBINIT MODE AMODE(24),RMODE(24)
ENTRY CEEBINIT
ALIAS CEEBLIBM
NAME CEEBINIT(R)
```

When you assembled the IMS user exit DFSBXITA, if you named the resulting object member DFSBXITA, replace CEEBXITA on line 1 with DFSBXITA.

Chapter 12. Specifying the TEST runtime options through the Language Environment user exit

z/OS Debugger provides a customized version of the Language Environment user exit (CEEBXITA). The user exit returns a TEST runtime option when called by the Language Environment initialization logic. z/OS Debugger provides a user exit that supports three different environments. This topic is also described in *IBM z/OS Debugger Customization Guide* with information specific to system programmers.

The user exit extracts the TEST runtime option from a user controlled data set with a name that is constructed from a naming pattern. The naming pattern can include the following tokens:

&USERID
z/OS Debugger replaces the &USERID token with the user ID of the current user. Each user can specify an individual TEST runtime option when debugging an application. This token is optional.

&PGMNAME
z/OS Debugger replaces the &PGMNAME token with the name of the main program (load module). Each program can have its own TEST runtime options. This token is optional.

z/OS Debugger provides the user exit in two forms:

- A load module. The load modules for the three environments are in the *hlq*.SEQAMOD data set. Use this load module if you want the default naming patterns and message display level. The default naming pattern is &USERID.DBGTOOL.EQAUOPTS and the default message display level is X'00'.
- Sample assembler user exit that you can edit. The assembler user exits for the three environments are in the *hlq*.SEQASAMP data set. You can also merge this source with an existing version of CEEBXITA. Use this source code if you want naming patterns or message display levels that are different than the default values.

z/OS Debugger provides a customized version of the Language Environment user exit named EQAD3CXT. The following table shows the environments in which this user exit can be used. The EQAD3CXT user exit determines the runtime environment internally and can be used in multiple environments.

Table 18. Language Environment user exits for various environments

Environment	User exit name
The following types of Db2 stored procedures that run in WLM-established address spaces: • type MAIN[1] • type SUB[2]	EQAD3CXT
IMS TM[3] and BTS[4]	EQAD3CXT
Batch	EQAD3CXT

Note:

1. EQAD3CXT is supported for DB2 version 7 or later. If Db2 RUNOPTS is specified, EQAD3CXT takes precedence over Db2 RUNOPTS.
2. EQAD3CXT supports Db2 stored procedures PROGRAM TYPE=SUB if you set the RRTN_SW flag as x'01'.
3. For IMS TM, if you do not sign on to the IMS terminal, you might need to run the EQASET transaction with the TSOID option. For instructions on how to run the EQASET transaction, see "Debugging Language Environment IMS MPPs without issuing /SIGN ON" in the *IBM z/OS Debugger User's Guide*.
4. For BTS, you need to specify Environment command (./E) with the user ID of the IO PCB. For example, if the user ID is ECSVT2, then the Environment command is ./E USERID=ECSVT2.

© Copyright IBM Corp. 1992, 2023

Each user exit can be used in one of the following ways:

- You can link the user exit into your application program.
- You can link the user exit into a private copy of a Language Environment module (CEEBINIT, CEEPIPI, or both), and then, only for the modules you might debug, place the SCEERUN data set containing this module in front of the system Language Environment modules in CEE.SCEERUN in the load module search path.

To learn about the advantages and disadvantages of each method, see "Comparing the two methods of linking CEEBXITA" on page 100.

To prepare a program to use the Language Environment user exit, do the following tasks:

1. "Editing the source code of CEEBXITA" on page 98.
2. "Linking the CEEBXITA user exit into your application program" on page 100 or "Linking the CEEBXITA user exit into a private copy of a Language Environment runtime module" on page 101.
3. "Creating and managing the TEST runtime options data set" on page 101.

Editing the source code of CEEBXITA

You can edit the sample assembler user exit that is provided in *hlq*.SEQASAMP to customize the naming patterns or message display level by doing one of the following tasks:

- Use SMP/E USERMOD EQAUMODK[8] to update the copy of the exit in the *hlq*.SEQAMOD data set. The system programmer usually implements the USERMOD. The USERMOD is in *hlq*.SEQASAMP.
- Create a private load module for the customized exit. Copy the assembler user exit that has the same name as the user exit from *hlq*.SEQASAMP to a local data set. Edit the patterns or message display level. Customize and run the JCL to generate a load module.

Modifying the naming pattern

The naming pattern of the data set that has the TEST runtime option is in the form of a sequential data set name. You can optionally specify a &USERID token, which z/OS Debugger substitutes with the user ID of the current user. You can also add a &PGMNAME token, which z/OS Debugger substitutes with the name of the main program (load module). However, if users create and manage the TEST runtime option data set with the **DTSP Profile** view in the remote debugger, do not specify the &PGMNAME token because the view does not support that token.

In some cases, the first character of a user ID is not valid for a name qualifier. A character can be concatenated before the &USERID token to serve as the prefix character for the user ID. For example, you can prefix the token with the character "P" to form P&USERID, which is a valid name qualifier after the current user ID is substituted for &USERID. For IMS, &USERID token might be substituted with one of the following values:

- IMS user ID, if users sign on to IMS.
- TSO user ID, if users do not sign on to IMS.

The default naming pattern is &USERID.DBGTOOL.EQAUOPTS. This is the pattern that is in the load module provided in *hlq*.SEQAMOD.

The following table shows examples of naming patterns and the corresponding data set names after z/OS Debugger substitutes the token with a value.

Table 19. Data set naming patterns, values for tokens, and resulting data set names

Naming pattern	User ID	Program name	Name after user ID substitution
&USERID.DBGTOOL.EQAUOPTS	JOHNDOE		JOHNDOE.DBGTOOL.EQAUOPTS

[8] USERMOD EQAUMODK is provided for updating EQAD3CXT. See "SMP/E USERMODs" in the *IBM z/OS Debugger Customization Guide* for an SMP/E USERMOD for this customization.

Table 19. Data set naming patterns, values for tokens, and resulting data set names (continued)			
Naming pattern	**User ID**	**Program name**	**Name after user ID substitution**
P&USERID.EQAUOPTS	123456		P123456.EQAUOPTS
DT.&USERID.TSTOPT	TESTID		DT.TESTID.TSTOPT
DT.&USERID.&PGMNAME.TSTOPT	TESTID	IVP1	DT.TESTID.IVP1.TSTOPT

To customize the naming pattern of the data set that has TEST runtime option, change the value of the DSNT DC statement in the sample user exit. For example:

```
* Modify the value in DSNT DC field below.
*
* Note: &USERID below has one additional '&', which is an escape
*       character.
*
DSNT_LN         DC  A(DSNT_SIZE)  Length field of naming pattern
DSNT            DC  C'&&USERID.DBGTOOL.EQAUOPTS'
DSNT_SIZE       EQU *-DSNT        Size of data set naming pattern
*
```

Modifying the message display level

You can modify the message display level for CEEBXITA. The following values set WTO message display level:

X'00'
Do not display any messages.

X'01'
Display error and warning messages.

X'02'
Display error, warning, and diagnostic messages.

The default value, which is in the load module in *hlq*.SEQAMOD, is X'00'.

To customize the message display level, change the value of the MSGS_SW DC statement in the sample user exit. For example:

```
* The following switch is to control WTO message display level.
*
*   x'00' - no messages
*   x'01' - error and warning messages
*   x'02' - error, warning, and diagnostic messages
*
MSGS_SW         DC  X'00'         message level
*
```

Modifying the call back routine registration

You can register a call back routine to the Language Environment. The Language Environment invokes the call back routine prior to calling a type SUB program using CALL_SUB API in the CEEPIPI environment. The call back routine performs a pattern match to determine if the type SUB program is to be debugged.

To customize the registration, change the value of the RRTN_SW DC statement.

x'00'
No registration of the call back routine.

x'01'
Registration of the call back routine.

Activate the cross reference function and modifying the cross reference table data set name

You can activate the cross reference function of the IMS Transaction and User ID Cross Reference Table and provide a cross reference table data set name. When an IMS transaction is initiated from the web or MQ gateway, it runs with a generic ID. If a user wants to debug the transaction, the cross reference function provides a way to associate the transaction with his or her user ID.

To customize the activation, change the value of the XRDSN_SW DC statement.

x'00'
Cross reference function is not activated.

x'01'
Cross reference function is activated.

To customize the cross reference table data set name, change the value of the XRDSN DC statement. You must provide a fully qualified MVS sequential data set name.

Comparing the two methods of linking CEEBXITA

You can link in the user exit CEEBXITA in the following ways:

- Link it into the application program.

  Advantage
  The user exit affects only the application program being debugged. This means you can control when z/OS Debugger is started for the application program. You might also not need to make any changes to your JCL to start z/OS Debugger.

  Disadvantage
  You must remember to remove the user exit for production or, if it isn't part of your normal build process, you must remember to relink it to the application program.

- Link it into a private copy of a Language Environment runtime load module (CEEBINIT, CEEPIPI, or both)

  Advantage
  You do not have to change your application program to use the user exit. In addition, you do not have to link edit extra modules into your application program.

  Disadvantage
  You need to take extra steps in preparing and maintaining your runtime environment:

  - Make a private copy of one or more Language Environment runtime routines
  - Only for the modules you might debug, customize your runtime environment to place the private copies in front of the system Language Environment modules in CEE.SCEERUN in the load module search path
  - When you apply maintenance to Language Environment, you might need to relink the routines.
  - When you upgrade to a new version of Language Environment, you must relink the routines.

If you link the user exit into the application program and into a private copy of a Language Environment runtime load module, which is in the load module search path of your application execution, the copy of the user exit in the application load module is used.

Linking the CEEBXITA user exit into your application program

If you choose to link the CEEBXITA user exit into your application program, use the following sample JCL, which links the user exit with the program TESTPGM. If you have customized the user exit and placed it in a private library, replace the data name, (*hlq*.SEQAMOD) of the first SYSLIB DD statement with the data set name that contains the modified user exit load module.

```
//SAMPLELK JOB ,
// MSGCLASS=H,TIME=(,30),MSGLEVEL=(2,0),NOTIFY=&SYSUID,REGION=0M
```

```
//*
//LKED    EXEC PGM=HEWL,REGION=4M,
//             PARM='CALL,XREF,LIST,LET,MAP,RENT'
//SYSLMOD  DD DISP=SHR,DSN=USERID.OUTPUT.LOAD
//SYSPRINT DD DISP=OLD,DSN=USERID.OUTPUT.LINKLIST(TESTPGM)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(1024,(200,20))
//*
//SYSLIB   DD DISP=SHR,DSN=hlq.SEQAMOD
//         DD DISP=SHR,DSN=CEE.SCEELKED
//*
//OBJECT   DD DISP=SHR,DSN=USERID.INPUT.OBJECT
//SYSLIN   DD *
     INCLUDE OBJECT(TESTPGM)
     INCLUDE SYSLIB(EQAD3CXT)
     NAME TESTPGM(R)
/*
```

Linking the CEEBXITA user exit into a private copy of a Language Environment runtime module

If you choose to customize a private copy of a Language Environment runtime load module, you need to ensure that your private copy of these load modules is placed ahead of your system copy of CEE.SCEERUN in your runtime environment.

The following table shows the Language Environment runtime load module and the user exit needed for each environment.

Table 20. Language Environment runtime module and user exit required for various environments

Environment	User exit name	CEE load module
The following types of Db2 stored procedures that run in WLM-established address spaces: • type MAIN • type SUB[1]	EQAD3CXT	CEEPIPI
IMS TM and BTS	EQAD3CXT	CEEBINIT
Batch	EQAD3CXT	CEEBINIT

Note:

1. EQAD3CXT supports Db2 stored procedures `PROGRAM TYPE=SUB` if you set the `RRTN_SW` flag as `x'01'`.

Edit and run sample *hlq*.SEQASAMP(EQAWLCE3) to create these updated Language Environment runtime modules. This is typically done by the system programmer installing z/OS Debugger. The sample creates the following load module data sets:

- *hlq*.DB2SP.SCEERUN(CEEPIPI)
- *hlq*.IMSTM.SCEERUN(CEEBINIT)
- *hlq*.BATCH.SCEERUN(CEEBINIT)

When you apply service to Language Environment that affects either of these modules (CEEPIPI or CEEBINIT) or you move to a new level of Language Environment, you need to rebuild your private copy of these modules by running the sample again.

Option 8 of the Debug Tool Utilities ISPF panel, "JCL for Batch Debugging", uses *hlq*.BATCH.SCEERUN if you use Invocation Method E.

Creating and managing the TEST runtime options data set

The TEST runtime options data set is an MVS data set that contains the Language Environment runtime options. The z/OS Debugger Language Environment user exit EQAD3CXT constructs the name of this data

set based on a naming pattern described in "Modifying the naming pattern" in the *IBM z/OS Debugger Customization Guide*.

You can create this data set in one of the following ways:

- By using Terminal Interface Manager (TIM), as described in “Creating and managing the TEST runtime options data set by using Terminal Interface Manager (TIM)” on page 102.
- By using IBM z/OS Debugger Utilities option 6, "z/OS Debugger User Exit Data Set", as described in “Creating and managing the TEST runtime options data set by using IBM z/OS Debugger Utilities” on page 103.
- By using the **z/OS Debugger Profiles** view. For more information, see the "Working with the z/OS Debugger Profiles view" topic in IBM Documentation.
- By specifying a non-CICS profile in the z/OS Batch Application with existing JCL launch configuration. For more information, see the "Launching a debug session using existing JCL" topic in IBM Documentation.
- By configuring the **Remote Profile** tab from Remote IMS Application with Isolation debug configurations.

Creating and managing the TEST runtime options data set by using Terminal Interface Manager (TIM)

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

Before you begin, verify that the user ID that you use to log on to Terminal Interface Manager (TIM) has permission to read and write the TEST runtime options data set.

To create the TEST runtime options data set by using Terminal Interface Manager, do the following steps:

1. Log on to Terminal Interface Manager.
2. In the **z/OS Debugger TERMINAL INTERFACE MANAGER** panel, press PF10.
3. In the *** Specify TEST Run-time Option Data Set *** panel, type in the name of a data set which follows the naming pattern specified by your system administrator, in the **Data Set Name** field. If the data set is not cataloged, type in a volume serial.
4. Press Enter. If Terminal Interface Manager cannot find the data set, it displays the *** Allocate TEST Run-time Option Data Set *** panel. Specify allocation parameters for the data set, then press Enter. Terminal Interface Manager creates the data set.
5. In the *** Edit TEST Run-time Option Data Set *** panel, make the following changes:

   Program name(s)
   Specify the names of up to eight programs you want to debug. You can specify specific names (for example, EMPLAPP), names appended with a wildcard character (*), or just the wildcard character (which means you want to debug all Language Environment programs).

   Test Option
   Specify whether to use TEST or NOTEST runtime option.

   Test Level
   Specify which TEST level to use: ALL, ERROR, or NONE.

   Commands File
   If you want to use a commands file, specify the name of a commands file in the format described in the *commands_file_designator* section of the topic "Syntax of the TEST run-time option" in the IBM z/OS Debugger Reference and Messages manual.

   Prompt Level
   Specify whether to use PROMPT or NOPROMPT.

Preferences File
If you want to use a preferences file, specify the name of a preferences file in the format described in the *preferences_file_designator* section of the topic "Syntax of the TEST run-time option" in the IBM z/OS Debugger Reference and Messages manual.

EQAOPTS File
If you want z/OS Debugger to run any EQAOPTS commands at run time, specify the name of the EQAOPTS file as a fully-qualified data set name.

Other run-time options
Type in any other Language Environment runtime options.

6. Terminal Interface Manager displays the part of the TEST runtime option that specifies which session type (debugging mode and display information) you want to use under the **Current debug display information** field. To change the session type, do the following steps:
   a. Press PF9.
   b. In the **Change session type** panel, select one of the following options:

      Full-screen mode using the z/OS Debugger Terminal Interface Manager
      Type in the user ID you will use to log on to Terminal Interface Manager and debug your program in the **User ID** field.

      Remote debug mode
      Type in the IP address in the **Address** field and port number in the **Port** field of the remote debugger's daemon.

   c. (Optional) Press Enter. Terminal Interface Manager accepts the changes and refreshes the panel.
   d. Press PF4. Terminal Interface Manager displays the *** Edit TEST Run-time Option Data Set *** panel and under the **Current debug session type string:** displays one of the following strings:
      - VTAM%*userid*, if you selected **Full-screen mode using the z/OS Debugger Terminal Interface Manager**.
      - TCPIP&*IP_address*%*port*, if you selected **Remote debug mode**.
7. Press PF4 to save your changes to the TEST runtime options data set and to return to the main Terminal Interface Manager screen.

Refer to the following topics for more information related to the material discussed in this topic.

- For more information about the values to specify for the Test Option, Test Level, and Prompt Level fields, see the topic "Syntax of the TEST run-time option" in the IBM z/OS Debugger Reference and Messages manual.
- For instructions on creating a commands file or preferences file, see the topics “Creating a commands file” on page 171 or “Creating a preferences file” on page 156.
- For instructions on creating an EQAOPTS file, see the topic "Providing EQAOPTS commands at run time" in the IBM z/OS Debugger Reference and Messages manual or IBM z/OS Debugger Customization Guide.
- For more information about other Language Environment runtime options, see *Language Environment Programming Reference*, SA22-7562.
- For more information about the values to specify for the **Full-screen mode using the z/OS Debugger Terminal Interface Manager** field, see “Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 131.
- For more information about the values to specify for the **Remote debug mode** field, see the online help for the remote GUI.

Creating and managing the TEST runtime options data set by using IBM z/OS Debugger Utilities

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

To create the TEST runtime options data set by using IBM z/OS Debugger Utilities, do the following steps:

1. Start IBM z/OS Debugger Utilities and select option 6, "z/OS Debugger User Exit Data Set".
2. Provide the name of a new or existing data set. Make sure the name matches the naming pattern. If you do not know the naming pattern, ask your system administrator. Remember the following rules:
   - Substitute the &PGMNAME token with the name of the program you want to debug. The program must be the main CSECT of the load module in a Language Environment enclave.
   - For IMS, &USERID token might be substituted with one of the following values:
     - IMS user ID, if users sign on to IMS.
     - TSO user ID, if users do not sign on to IMS.
3. Fill out the rest of the fields with the TEST runtime options you want to use and the names of up to eight additional programs to debug.
4. For IMS, you can also fill out the IMS Subsystem ID, or IMS Transaction ID field, or both. If provided, the IDs are used as additional filtering criteria.
5. For batch, you can also specify the **Job name** or **Step name** fields, or both. If provided, the names are used as additional filtering criteria.

   You can use a wildcard (*) at the end of a job name or step name. For example, a job name of JOB1* means that a job name that starts with JOB1 passes the matching test, like JOB1, JOB1A, or JOB1ABC; a job name of * means that any job name passes the matching test.

Part 3. Starting z/OS Debugger

© Copyright IBM Corp. 1992, 2023

Chapter 13. Writing the TEST runtime option string

The instructions in this section apply to programs that run in Language Environment. For programs that do not run in Language Environment, refer to the instructions in "Starting z/OS Debugger for programs that start outside of Language Environment" on page 134.

This topic describes some of the factors that you need to consider when you use the TEST runtime option, provides examples, and describes other runtime options that you might need to specify. The syntax of the TEST runtime option is described in the topic "TEST run-time option" in *IBM z/OS Debugger Reference and Messages*.

To specify how z/OS Debugger gains control of your application and begins a debug session, use the TEST runtime option.

The simplest form of the TEST option is TEST with no suboptions specified. If Debug Profile Service is active, a simple TEST option enables delay debug mode, regardless of whether the **DLAYDBG EQAOPTS** command is in effect. z/OS Debugger acquires the naming pattern for the delay debug data set from the profile service. For more information about this behavior, see "Simple TEST option" on page 107.

If you choose a more detailed TEST option, suboptions provide you with more flexibility. There are four types of suboptions available:

test_level
Determines what high-level language conditions raised by your program cause z/OS Debugger to gain control of your program.

commands_file
Determines which primary `commands file` is used as the initial source of commands.

prompt_level
Determines whether an initial commands list is unconditionally run during program initialization.

preferences_file
Specifies the session parameter and a file that you can use to specify default settings for your debugging environment, such as customizing the settings on the z/OS Debugger Profile panel.

Special considerations while using the TEST run-time option

When you use the TEST run-time option, there are several implications to consider, which are described in this section.

Simple TEST option

You can add a simple TEST option with no suboptions, or specify the default TEST suboptions of `TEST(ALL,*,PROMPT,INSPPREF)` to start z/OS Debugger in delay debug mode under most conditions for non-CICS tasks if the Debug Profile Service API is available.

- For batch applications, add TEST to the PARM string or CEEOPTS DD.
- For IMS dependent regions, add TEST to the CEEOPTS DD.
- For WLM procedures used by Db2 Stored Procedures, add TEST to the CEEOPTS DD.
- For Unix Systems Services processes, add TEST to the _CEE_RUNOPTS environment variable.

The following rules apply:

- If you define the TEST suboptions in your program with `#pragma runopts` or the PLIXOPT string, those suboptions are in effect. For more information, see Defining TEST suboptions in your program.
- If you are executing a process in a TSO interactive session from z/OS Debugger Setup Utility or using a TSO command, the debugger starts under your TSO session, as though you had specified `TEST(ALL,*,PROMPT,MFI:INSPPREF)`.

© Copyright IBM Corp. 1992, 2023

- In all other cases, when Debug Profile Service is active, z/OS Debugger operates in delay debug mode. The following delay debug commands are in effect unless you explicitly specify them using the EQAOPTS load module:
  - DLAYDBGCND: ALL.
  - DLAYDBGTRC: 0.
  - DLAYDBGXRF is not in effect.

  DLAYDBGDSN is set based on the value supplied to the Debug Profile Service API. If you specify DLAYDBGDSN in your EQAOPTS load module, the EQAOPTS setting is ignored.

  For more information on delay debug mode, see "Using delay debug mode to delay starting of a debug session " on page 397.

Defining TEST suboptions in your program

In C, C++ or PL/I, you can define TEST with suboptions using a #pragma runopts or PLIXOPT string, then specify TEST with no suboptions at run time. This causes the suboptions specified in the #pragma runopts or PLIXOPT string to take effect.

You can change the TEST/NOTEST run-time options at any time with the SET TEST command.

Suboptions and NOTEST

Some suboptions are disabled with NOTEST, but are still allowed. This means you can start your program using the NOTEST option and specify suboptions you might want to take effect later in your debug session. The program begins to run without z/OS Debugger taking control.

To enable the suboptions you specified with NOTEST, start z/OS Debugger during your program's run time by using a library service call such as CEETEST, PLITEST, or the __ctest() function.

Implicit breakpoints

If the test level in effect causes z/OS Debugger to gain control at a condition or at a particular program location, an implicit breakpoint with no associated action is assumed. This occurs even though you have not previously defined a breakpoint for that condition or location using an initial command string or a primary commands file. Control is given to your terminal or to your primary commands file.

Primary commands file and USE file

The primary commands file acts as a surrogate terminal. After it is accessed as a source of commands, it continues to act in this capacity until all commands have been run or the application has ended. This differs from the USE file in that, if a USE file contains a command that returns control to the program (such as STEP or GO), all subsequent commands are discarded. However, USE files started from within a primary commands file take on the characteristics of the primary commands file and can be run until complete.

The initial command list, whether it consists of a command string included in the run-time options or a primary commands file, can contain a USE command to get commands from a secondary file. If started from the primary commands file, a USE file takes on the characteristics of the primary commands file.

Running in batch mode

In batch mode, when the end of your commands file is reached, a GO command is run at each request for a command until the program terminates. If another command is requested after program termination, a QUIT command is forced.

Starting z/OS Debugger at different points

If z/OS Debugger is started during program initialization, it is started before all the instructions in the main prolog are run. At that time, no program blocks are active and references to variables in the main

procedure cannot be made, compile units cannot be called, and the GOTO command cannot be used. However, references to static variables can be made.

If you enter the STEP command at this point, before entering any other commands, both program and Language Environment initialization are completed and you are given access to all variables. You can also enter all valid commands.

If z/OS Debugger is started while your program is running (for example, by using a CEETEST call), it might not be able to find all compile units associated with your application. Compile units located in load modules that are not currently active are not known to z/OS Debugger, even if they were run prior to z/OS Debugger's initialization.

For example, suppose load module `mod1` contains compile units `cu1` and `cu2`, both compiled with the TEST option. The compile unit `cu1` calls `cux`, contained in load module `mod2`, which returns after it completes processing. The compile unit `cu2` contains a call to the CEETEST library service. When the call to CEETEST initializes z/OS Debugger, only `cu1` and `cu2` are known to z/OS Debugger. z/OS Debugger does not recognize `cux`.

The initial command string is run only once, when z/OS Debugger is first initialized in the process.

Commands in the preferences file are run only once, when z/OS Debugger is first initialized in the process.

Session log

The session log stores the commands entered and the results of the execution of those commands. The session log saves the results of the execution of the commands as comments. This allows you to use the session log as a commands file.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Link-editing EQADCCXT into your program" on page 81

Related references

IBM z/OS Debugger Reference and Messages

Precedence of Language Environment runtime options

The Language Environment runtime options have the following order of precedence (from highest to lowest):

1. Installation options in the CEEDOPT file that were specified as nonoverrideable with the NONOVR attribute.
2. Options specified by the Language Environment assembler user exit. In the CICS environment, z/OS Debugger uses the DTCN transaction and the customized Language Environment user exit EQADCCXT, which is link-edited with the application. In the IMS Version 8 environment, IMS retrieves the options that most closely match the options in its Language Environment runtime options table. You can edit this table by using IBM z/OS Debugger Utilities.
3. Options specified at the invocation of your application, using the TEST runtime option, unless accepting runtime options is disabled by Language Environment (EXECOPS/NOEXECOPS).
4. Options specified within the source program (with `#pragma` or PLIXOPT) or application options specified with CEEUOPT and link-edited with your application.

   If the object module for the source program is input to the linkage editor before the CEEUOPT object module, *then* these options override CEEUOPT defaults. You can force the order in which objects modules are input by using linkage editor control statements.
5. Region-wide CICS or IMS options defined within CEEROPT.
6. Option defaults specified at installation in CEEDOPT.
7. IBM-supplied defaults.

Suboptions are processed in the following order:

1. Commands entered at the command line override any defaults or suboptions specified at run time.
2. Commands run from a preferences file override the command string and any defaults or suboptions specified at run time.
3. Commands from a commands file override default suboptions, suboptions specified at run time, commands in a command string, and commands in a preferences file.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS Language Environment Programming Guide

Example: TEST runtime options

The following examples of using the TEST runtime option are provided to illustrate runtime options available for your programs. These commands do not illustrate complete commands. The complete syntax of the TEST runtime option can be found in "Syntax of the TEST run-time option" in *IBM z/OS Debugger Reference and Messages*.

Remote debugging
If you are working in remote debug mode, that is, you are debugging your host application from your workstation, the following examples apply:

Table 21. TEST runtime option examples for remote debugging

Scenario	TEST runtime option usage
Eclipse IDE using Debug Manager	`TEST(,,,DBMDT:*)` Indicates that you want to start a debug session in Debug Tool compatibility mode for an Eclipse debug client. The address of the client is automatically determined by Debug Manager for the current user ID.
Eclipse IDE using workstation IP address	`TEST(,,,TCPIP&abc.example.com%8001:*)` Indicates that you want to start a debug session in Debug Tool compatibility mode for an Eclipse debug client. In this example, the TCP/IP address of the client is manually specified as *abc.example.com* and the debug daemon is listening on port *8001*.

IBM Z Open Debug	`TEST(,,,RDS:*)`
	Indicates that you want to start a debug session using Remote Debug Service for Wazi for VS Code or Wazi for Dev Spaces. In this scenario, Remote Debug Service must be running and configured.
	`TEST(,,,TCPIP&127.0.0.1%8001:*)`
	Indicates that you want to start a debug session using Remote Debug Service for Wazi for VS Code or Wazi for Dev Spaces. In this scenario, Remote Debug Service is running on the local z/OS machine using the TCP/IP address of *127.0.0.1* and it is listening on port *8001* for internal z/OS Debugger connections.

When Debug Profile Service is active, optionally you can use TEST with no suboptions specified to enable delay debug mode. For more information, see "Simple TEST option" on page 107.

Code coverage

If you want to start code coverage sessions, the following examples apply:

Table 22. TEST runtime option examples for code coverage

Scenario	TEST runtime option usage
Code coverage with Eclipse IDE using Debug Manager	`TEST(,,,DBMDT:*)` Indicates that you want to start a code coverage session in Debug Tool compatibility mode for an Eclipse IDE. The address of the client is automatically determined by Debug Manager for the current user ID.
Code coverage with Eclipse IDE using workstation IP address	`TEST(,,,TCPIP&abc.example.com%8001:*)` Indicates that you want to start a code coverage session in Debug Tool compatibility mode for an Eclipse IDE. In this example, the TCP/IP address of the client is manually specified as *abc.example.com* and the debug daemon is listening on port *8001*.
Headless code coverage using Remote Debug Service	`TEST(,,,RDS:*)` Indicates that you want to run a code coverage session and connect to Remote Debug Service. In this scenario, Remote Debug Service must be running and configured to collect code coverage.
Headless code coverage on z/OS	`TEST(,,,TCPIP&127.0.0.1%8001:*)` Indicates that you want to run a code coverage session using headless code coverage. In this scenario, headless code coverage is running on the local z/OS machine using the TCP/IP address of *127.0.0.1* and it is listening on port *8001* for z/OS Debugger connections.
Headless code coverage with a Windows or Linux client	`TEST(,,,TCPIP&cde.example.com%8001:*)` Indicates that you want to start a code coverage session using headless code coverage. In this scenario, the headless code coverage daemon is running on a Windows or Linux machine using the TCP/IP address of *cde.example.com* and it is listening on port *8001* for z/OS Debugger connections.

Notes:

- The EQA_STARTUP_KEY is also required to indicate code coverage. For more information, see "EQA_STARTUP_KEY" on page 467 and the "Specifying code coverage options in the startup key" topic in IBM Documentation.
- Code coverage is not supported in IBM Z and Cloud Modernization Stack (Wazi Code).
- Headless code coverage is not supported in IBM Debug for z/OS.

Full-screen debugging

If you want to use full-screen debugging, the following examples apply:

Table 23. TEST runtime options for full-screen debugging

Scenario	TEST runtime option usage
CICS full screen mode	`TEST(ALL,,,MFI%F000:)` When running under CICS®, z/OS Debugger displays its screens on terminal ID F000.

Full-screen mode with a dedicated terminal	`TEST(ALL,,,MFI%TRMLU001:)`
	For use with full-screen mode using a dedicated terminal without Terminal Interface Manager. The VTAM LU TRMLU001 is used for display. This terminal must be known to VTAM and not in session when z/OS Debugger is started.
	`TEST(ALL,,,MFI%SYSTEM01.TRMLU001:)`
	For use in the following situations:
	• You are using full-screen mode using a dedicated terminal without Terminal Interface Manager.
	• You must specify a network identifier.
	The VTAM LU TRMLU001 on network node SYSTEM01 is used for display. This terminal must be known to VTAM and not in session when z/OS Debugger is started.

Full-screen mode using Terminal Interface Manager	`TEST(ALL,,,VTAM%USERABCD:)` For use with full-screen mode using the Terminal Interface Manager. The user accessed the z/OS Debugger Terminal Interface Manager with user id USERABCD.
TSO full-screen mode	`TEST(,,,MFI:*)` Indicates that you want the debugger to start a debug session in TSO full-screen mode.

Note: Full-screen debugging is supported only in IBM Developer for z/OS Enterprise Edition and IBM Debug for z/OS.

NOTEST

z/OS Debugger is not started at program initialization. Note that a call to CEETEST, PLITEST, or `__ctest()` causes z/OS Debugger to be started during the program's execution.

NOTEST(ALL,MYCMDS,*,*)

z/OS Debugger is not started at program initialization. Note that a call to CEETEST, PLITEST, or `__ctest()` causes z/OS Debugger to be started during the program's execution. After z/OS Debugger is started, the suboptions specified become effective and the commands in the file allocated to DD name of MYCMDS are processed.

If you specify NOTEST and control has returned from the program in which z/OS Debugger first became active, you can no longer debug non-Language Environment programs or detect non-Language Environment events.

TEST
Specifying TEST with no suboptions causes a check for other possible definitions of the suboption. For example, C and C++ allow default suboptions to be selected at compile time using `#pragma runopts`. Similarly, PL/I offers the PLIXOPT string. Language Environment provides the macro CEEXOPT. Using this macro, you can specify installation and program-specific defaults.

If no other definitions for the suboptions exist, the IBM-supplied default suboptions (`ALL,*,PROMPT,INSPPREF`) are in effect. In an environment that is not a foreground TSO task, z/OS Debugger operates in delay debug mode when the Debug Profile Service API is active.

TEST(ALL,*,*,*)
z/OS Debugger is not started initially; however, any condition or an attention in your program causes z/OS Debugger to be started, as does a call to CEETEST, PLITEST, or `__ctest()`. Neither a primary commands file nor preferences file is used.

TEST(NONE,,*,*)
z/OS Debugger is not started initially and begins by running in a "production mode", that is, with minimal effect on the processing of the program. However, z/OS Debugger can be started using CEETEST, PLITEST, or `__ctest()`.

TEST(ALL,test.scenario,PROMPT,prefer)
z/OS Debugger is started at the end of environment initialization, but before the main program prolog has completed. The ddname `prefer` is processed as the preferences file, and subsequent commands are found in data set `test.scenario`. If all commands in the commands file are processed and you issue a STEP command when prompted, or a STEP command is run in the commands file, the main block completes initialization (that is, its AUTOMATIC storage is obtained and initial values are set). If z/OS Debugger is reentered later for any reason, it continues to obtain commands from `test.scenario` repeating this process until end-of-file is reached. At this point, commands are obtained from your terminal.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS Language Environment Programming Guide

Specifying additional run-time options with VS COBOL II and PL/I programs

There are two additional run-time options that you might need to specify to debug COBOL and PL/I programs: STORAGE and `TRAP(ON)`.

Specifying the STORAGE run-time option

The STORAGE run-time option controls the initial content of storage when allocated and freed, and the amount of storage that is reserved for the "out-of-storage" condition. When you specify one of the parameters in the STORAGE run-time option, all allocated storage processed by the parameter is initialized to that value. If your program does not have self-initialized variables, you must specify the STORAGE run-time option.

Specifying the TRAP(ON) run-time option

The `TRAP(ON)` run-time option is used to fully enable the Language Environment condition handler that passes exceptions to the z/OS Debugger. Along with the TEST option, it **must** be used if you want the z/OS Debugger to take control automatically when an exception occurs. You must also use the `TRAP(ON)` run-time option if you want to use the `GO BYPASS` command and to debug handlers you have written. Using `TRAP(OFF)` with the z/OS Debugger causes unpredictable results to occur, including the operating system cancelling your application and z/OS Debugger when a condition, abend, or interrupt is encountered.

Note: This option replaces the OS PL/I and VS COBOL II `STAE/NOSTAE` options.

Specifying TEST run-time option with #pragma runopts in C and C++

The TEST run-time option can be specified either when you start your program, or directly in your source by using this #pragma:

```
#pragma runopts (test(suboption,suboption...))
```

This #pragma must appear before the first statement in your source file. For example, if you specified the following in the source:

```
#pragma runopts (notest(all,*,prompt))
```

then entered TEST on the command line, the result would be

```
TEST(ALL,*,PROMPT).
```

TEST overrides the NOTEST option specified in the #pragma and, because TEST does not contain any suboptions of its own, the suboptions ALL, *, and PROMPT remain in effect.

If you link together two or more compile units with differing #pragmas, the options specified with the first compile are honored. With multiple enclaves, the options specified with the first enclave (or compile unit) started *in each new process* are honored.

If you specify options on the command line and in a #pragma, any options entered on the command line override those specified in the #pragma unless you specify NOEXECOPS. Specifying NOEXECOPS, either in a #pragma or with the EXECOPS compiler option, prevents any command line options from taking effect.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
z/OS XL C/C++ User's Guide

Chapter 14. Starting z/OS Debugger from the IBM z/OS Debugger Utilities

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The **z/OS Debugger Setup File** option (starts z/OS Debugger Setup Utilities or DTSU) in IBM z/OS Debugger Utilities helps you manage setup files which store the following information:

- file allocation statements
- run-time options
- program parameters
- the name of your program

Then you use the setup files to run your program in foreground or batch. The z/OS Debugger Setup Utility (DTSU) RUN command performs the file allocations and then starts the program with the specified options and parameters in the foreground. The DTSU SUBMIT command submits a batch job to start the program.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Creating the setup file" on page 115
"Editing an existing setup file" on page 115
"Saving your setup file" on page 117
"Starting your program" on page 117

Creating the setup file

You can have several setup files, but you must create them one at a time. To create a setup file, do the following steps:

1. From the **IBM z/OS Debugger Utilities** panel, select the **z/OS Debugger Setup File** option.
2. In the `z/OS Debugger Foreground – Edit Setup File` panel, type the name of the new setup file in the **Setup File Library** or **Other Data Set Name** field. Do not specify a member name if you are creating a sequential data set. If you are creating a setup file for a Db2 program, select the **Initialize New setup file for Db2** field. Press Enter.
3. A panel similar to the ISPF 3.2 "Allocate New Data Set" panel appears when you enter the name of the new set up file in the **Other Data Set Name** field. You can modify the default allocation parameters. Enter the END command or press PF3 to continue.
4. The `Edit – Edit Setup File` panel appears. You can enter file allocation statements, run-time options, and program parameters.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Entering file allocation statements, runtime options, and program parameters" on page 116

Editing an existing setup file

You can have several setup files, but you can edit only one file at a time. To edit an existing setup file, do the following steps:

1. From the `IBM z/OS Debugger Utilities` panel, select the **z/OS Debugger Setup File** option.
2. In the `z/OS Debugger Foreground – Edit Setup File` panel, type the name of the existing setup file in the **Setup File Library** or **Other Data Set Name** field. Press Enter to continue.

© Copyright IBM Corp. 1992, 2023

3. The `Edit – Edit Setup File` panel appears. You can modify file allocation statements, run-time options, and program parameters.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Entering file allocation statements, runtime options, and program parameters" on page 116

Copying information into a setup file from an existing JCL

You can enter the COPY command to copy an EXEC statement and its associated DD statements from another data set containing JCL.

You can use option A to select a step of a job, and convert it to the setup file format.

Entering file allocation statements, runtime options, and program parameters

The top part of the Edit–Setup File panel contains the name of the program (load module) that you want to run and the runtime parameter string. If the setup file is for a Db2 program, the panel also contains fields for the Db2 System identifier and the Db2 plan. The bottom part of the Edit–Setup File panel contains the file allocation statements. This part of the panel is similar to an ISPF edit panel. You can insert new lines, copy (repeat) a line, delete a line, and type over information on a line.

To modify the name of the load module, type the new name in the **Load Module Name** field.

To modify the parameter string:

1. Select the format of the parameter string and whether the program is to start in the Language Environment. Non-Language Environment COBOL programs do not run in Language Environment. If you are debugging a non-Language Environment COBOL program, select the non-Language Environment option.
2. Enter the parameter string in one of the following ways:
   - Type the parameter string in the **Enter / to modify parameters** field.
   - Type a slash ("/") before the **Enter / to modify parameters** field and press Enter. The z/OS Debugger Foreground - Modify Parameter String panel appears. Define your runtime options and suboptions by doing the following steps:
     a. Define the TEST run-time option and its suboptions.
     b. Enter any Language Environment or z/OS Debugger runtime options and other program parameters.
     c. Press PF3. DTSU creates the parameter string from the options that you specified and puts it in the **Enter / to modify parameters** field.

In the file allocation section of the panel, each line represents an element of a DD name allocation or concatenation. The statements can be modified, copied, deleted, and reordered.

To modify a statement, do one of the following steps:

- Modify the statement directly on the Edit – Edit Setup File panel:
  1. Move your cursor to the statement you want to modify.
  2. Type the new information over the existing information.
  3. Press Enter.
- Modify the statement by using a select command:
  1. Move your cursor to the statement you want to modify.
  2. Type one of the following select commands:
     – SA - Specify allocation information

- SD - Specify DCB information
- SS - Specify SMS information
- SP - Specify protection information
- SO - Specify sysout information
- SX - Specify all DD information by column display
- SZ - Specify all DD information by section display

3. Press Enter.

To copy a statement, do the following steps:

1. Move your cursor to the **Cmd** field of the statement you want to copy.
2. Type R and press Enter. The statement is copied into a new line immediately following the current line.

To delete a statement, do the following steps:

1. Move your cursor to the **Cmd** field of the statement you want to delete.
2. Type D and press Enter. The statement is deleted.

IBM z/OS Debugger Utilities does not support reordering the DD names, only the data sets within each concatenation. The DD names are automatically sorted in alphabetical order. To reorder statements in a concatenation, do the following steps:

1. Move your cursor to the sequence number field of a statement you want to move and enter the new sequence number.

To insert a new line, do the following steps:

1. Move your cursor to the **Cmd** field of the line right above the line you want a new statement inserted.
2. Type I and press Enter.
3. Move your cursor to the new line and type in the new information or use one of the Select commands.

The Edit and Browse line commands allow you to modify or view the contents of the data set name specified for DD and SYSIN DD types.

You can use the DDNAME STEPLIB to specify the load module search order.

For additional help, move the cursor to any field and enter the HELP command or press PF1.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Saving your setup file" on page 117

Saving your setup file

To save your information, enter the SAVE command. To save your information in a second data set and continue editing in the second data set, enter the SAVE AS command.

To save your setup file and exit the Edit–Edit Setup File panel, enter the END command or press PF3.

To exit the Edit–Edit Setup File panel without saving any changes to your setup file, enter the CANCEL command or press PF12.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Starting your program" on page 117

Starting your program

To perform the allocations and run the program with the specified parameter string, enter the RUN command or press PF4.

To generate JCL from the information in the setup file and then submit to the batch job, enter the SUBMIT command or press PF10.

Chapter 15. Starting z/OS Debugger from a program

The instructions in this section apply to programs that run in Language Environment. For programs that do not run in Language Environment, refer to the instructions in “Starting z/OS Debugger for programs that start outside of Language Environment” on page 134.

z/OS Debugger can also be started directly from within your program using one of the following methods:

- Language Environment provides the callable service CEETEST that is started from Language Environment-enabled languages.
- For C or C++ programs, you can use a `__ctest()` function call or include a `#pragma runopts` specification in your program.

  Note: The `__ctest()` function is not supported in CICS.
- For PL/I programs, you can use a call to PLITEST or by including a PLIXOPT string that specifies the correct TEST run-time suboptions to start z/OS Debugger.

However, you cannot use these methods in Db2 stored procedures with the PROGRAM TYPE of SUB.

If you use these methods to start z/OS Debugger, you can debug non-Language Environment programs and detect non-Language Environment events only in the enclave in which z/OS Debugger first appeared and in subsequent enclaves. You cannot debug non-Language Environment programs or detect non-Language Environment events in higher-level enclaves.

To start z/OS Debugger using these alternatives, you still need to be aware of the TEST suboptions specified using NOTEST, CEEUOPT, or other "indirect" settings.

“Example: using CEETEST to start z/OS Debugger from C/C++” on page 122
“Example: using CEETEST to start z/OS Debugger from COBOL” on page 123
“Example: using CEETEST to start z/OS Debugger from PL/I” on page 124

Related tasks

“Starting z/OS Debugger with CEETEST” on page 119
“Starting z/OS Debugger with PLITEST” on page 125
“Starting z/OS Debugger with the __ctest() function” on page 126
“Starting z/OS Debugger under CICS by using CEEUOPT” on page 141

Refer to the following topics for more information related to the material discussed in this topic.

Related references

“Special considerations while using the TEST run-time option” on page 107

Starting z/OS Debugger with CEETEST

Using CEETEST, you can start z/OS Debugger from within your program and send it a string of commands. If no command string is specified, or the command string is insufficient, z/OS Debugger prompts you for commands from your terminal or reads them from the commands file. In addition, you have the option of receiving a feedback code that tells you whether the invocation procedure was successful.

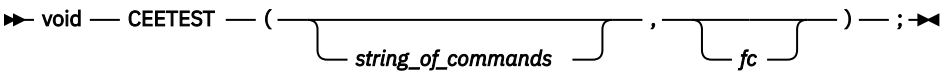
If you don't want to compile your program with hooks, you can use CEETEST calls to start z/OS Debugger at strategic points in your program. If you decide to use this method, you still need to compile your application so that symbolic information is created.

Using CEETEST when z/OS Debugger is already initialized results in a reentry that is similar to a breakpoint.

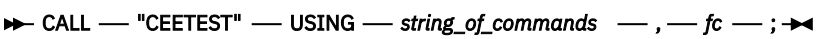
The following diagrams describe the syntax for CEETEST:

For C and C++

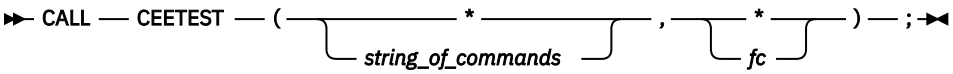
© Copyright IBM Corp. 1992, 2023



For COBOL



For PL/I



***string_of_commands* (input)**

Halfword-length prefixed string containing a z/OS Debugger command list. The command string *string_of_commands* is optional.

If z/OS Debugger is available, the commands in the list are passed to the debugger and carried out.

If *string_of_commands* is omitted, z/OS Debugger prompts for commands in interactive mode.

For z/OS Debugger, remember to use the continuation character if your command exceeds 72 characters.

The first command in the command string can indicate that you want to start z/OS Debugger in one of the following debug modes:

- full-screen mode using the Terminal Interface Manager
- remote debug mode

To indicate that you want to start z/OS Debugger in full-screen mode using a dedicated terminal without Terminal Interface Manager, specify the MFI suboption of the TEST runtime option with the LU name of the dedicated terminal. For example, you can code the following call in your PL/I program:

```
Call CEETEST('MFI%TRMLU001:*;Query Location;Describe CUS;',*);
```

For a COBOL program, you can code the following call:

```
01 PARMS.
05 LEN PIC S9(4) BINARY Value 43.
05 PARM PIC X(43) Value 'MFI%TRMLU001:*;Query Location;Describe CUS;'.

CALL "CEETEST" USING PARMS FC.
```

To indicate that you want to start z/OS Debugger in full-screen mode using the Terminal Interface Manager, specify the VTAM suboption of the TEST runtime option with the User ID that you supplied to the Terminal Interface Manager. For example, you can code the following call in your PL/I program:

```
Call CEETEST(VTAM%USERABCD:*;Query Location;Describe CUS;,*);
```

In these examples, the suboption `:*` can be replaced with the name of a preferences file. If you started z/OS Debugger the TEST runtime option and specified a preferences file and you specify another preferences file in the CEETEST call, the preferences file in the CEETEST call replaces the preferences file specified with the TEST runtime option.

To indicate that you want to start z/OS Debugger in remote debug mode, specify the DBMDT or TCPIP suboptions of the TEST runtime option with the userid you logged on RSE with (DBMDT) or the IP address and port number that the remote debugger is listening to (TCPIP).

Note: You cannot use CEETEST to start z/OS Debugger in standard mode.

To start z/OS Debugger in Debug Tool compatibility mode during remote debug by using Debug Manager and specify the user ID you logged on RSE with, code the following call:

```
Call CEETEST('DBMDT%userid:*;',*);
```

To start z/OS Debugger in Debug Tool compatibility mode and specify the TCP/IP address of your workstation, code the following call:

```
Call CEETEST('TCPIP&your.company.com%8001:*;',*);
```

These calls must include the trailing semicolon (;).

***fc* (output)**
A 12-byte *feedback* code, optional in some languages, that indicates the result of this service.

CEE000

Severity = 0
Msg_No = Not Applicable
Message = Service completed successfully

CEE2F2

Severity = 3
Msg_No = 2530
Message = A debugger was not available

Note: The CEE2F2 feedback code can also be obtained by MVS/JES batch applications. For example, either the z/OS Debugger environment was corrupted or the debug event handler could not be loaded.

Language Environment provides a callable service called CEEDCOD to help you decode the fields in the feedback code. Requesting the return of the feedback code is recommended.

For C and C++ and COBOL, if z/OS Debugger was started through CALL CEETEST, the GOTO command is only allowed after z/OS Debugger has returned control to your program via STEP or GO.

Additional notes about starting z/OS Debugger with CEETEST

C and C++
Include leawi.h header file.

COBOL
Include CEEIGZCT. CEEIGZCT is in the Language Environment SCEESAMP data set.

PL/I
Include CEEIBMAW and CEEIBMCT. CEEIBMAW is in the Language Environment SCEESAMP data set.

Batch and CICS nonterminal processes
We strongly recommend that you use feedback codes (fc) when using CEETEST to initiate z/OS Debugger from a batch process or a CICS nonterminal task; otherwise, results are unpredictable.

QUIT DEBUG
After you use QUIT DEBUG to stop your debug session, you can restart z/OS Debugger with CEETEST. To start z/OS Debugger when a CEETEST call is encountered, set the EQAOPTS CEEREACTAFTERQDBG command to YES.

Note: You cannot use CEETEST to start z/OS Debugger in standard mode for remote debugging.

"Example: using CEETEST to start z/OS Debugger from C/C++" on page 122
"Example: using CEETEST to start z/OS Debugger from COBOL" on page 123
"Example: using CEETEST to start z/OS Debugger from PL/I" on page 124

Related tasks
"Entering multiline commands in full-screen" on page 263

Related references

z/OS Language Environment Programming Guide
IBM z/OS Debugger Reference and Messages

Example: using CEETEST to start z/OS Debugger from C/C++

The following examples show how to use the Language Environment callable service CEETEST to start z/OS Debugger from C or C++ programs.

Example 1

In this example, an empty command string is passed to z/OS Debugger and a pointer to the Language Environment feedback code is returned. If no other TEST run-time options have been compiled into the program, the call to CEETEST starts z/OS Debugger with all defaults in effect. After it gains control, z/OS Debugger prompts you for commands.

```
#include <leawi.h>
#include <string.h>
#include <stdio.h>

int main(void) {
  _VSTRING  commands;
  _FEEDBACK fc;

  strcpy(commands.string, "");
  commands.length = strlen(commands.string);

  CEETEST(&commands, &fc);

}
```

Example 2

In this example, a string of valid z/OS Debugger commands is passed to z/OS Debugger and a pointer to Language Environment feedback code is returned. The call to CEETEST starts z/OS Debugger and the command string is processed. At statement 23, the values of x and y are displayed in the Log, and execution of the program resumes. Barring further interrupts, the behavior at program termination depends on whether you have set AT TERMINATION:

- If you have set AT TERMINATION, z/OS Debugger regains control and prompts you for commands.
- If you have not set AT TERMINATION, the program terminates.

The command LIST(z) is discarded when the command GO is executed.

Note: If you include a STEP or GO in your command string, all commands after that are not processed. The command string operates like a commands file.

```
#include <leawi.h>
#include <string.h>
#include <stdio.h>

int main(void) {
  _VSTRING  commands;
  _FEEDBACK fc;

  strcpy(commands.string, "AT LINE 23; {LIST(x); LIST(y);} GO; LIST(z)");
  commands.length = strlen(commands.string);
⋮
  CEETEST(&commands, &fc);
⋮
}
```

Example 3

In this example, a string of valid z/OS Debugger commands is passed to z/OS Debugger and a pointer to the feedback code is returned. If the call to CEETEST fails, an informational message is printed.

If the call to CEETEST succeeds, z/OS Debugger is started and the command string is processed. At statement 30, the values of x and y are displayed in the Log, and execution of the program resumes. Barring further interrupts, the behavior at program termination depends on whether you have set AT TERMINATION:

- If you have set AT TERMINATION, z/OS Debugger regains control and prompts you for commands.
- If you have not set AT TERMINATION, the program terminates.

```
#include <leawi.h>
#include <string.h>
#include <stdio.h>

#define SUCCESS "\0\0\0\0"

int main (void) {

   int x,y,z;
   _VSTRING commands;
   _FEEDBACK fc;

   strcpy(commands.string,"AT LINE 30 { LIST(x); LIST(y); } GO;");
   commands.length = strlen(commands.string);
.
   CEETEST(&commands,&fc);
.
   if (memcmp(&fc,SUCCESS,4) != 0) {
      printf("CEETEST failed with message number %d\n",fc.tok_msgno);
      return(2999);
   }
}
```

Example: using CEETEST to start z/OS Debugger from COBOL

The following examples show how to use the Language Environment callable service CEETEST to start z/OS Debugger from COBOL programs.

Example 1

A command string is passed to z/OS Debugger at its invocation and the feedback code is returned. After it gains control, z/OS Debugger becomes active and prompts you for commands or reads them from a commands file.

```
 01  FC.
    02  CONDITION-TOKEN-VALUE.
    COPY CEEIGZCT.
        03  CASE-1-CONDITION-ID.
            04  SEVERITY    PIC S9(4) BINARY.
            04  MSG-NO      PIC S9(4) BINARY.
        03  CASE-2-CONDITION-ID
                  REDEFINES CASE-1-CONDITION-ID.
            04  CLASS-CODE  PIC S9(4) BINARY.
            04  CAUSE-CODE  PIC S9(4) BINARY.
        03  CASE-SEV-CTL    PIC X.
        03  FACILITY-ID     PIC XXX.
    02  I-S-INFO            PIC S9(9) BINARY.
77  Debugger                PIC x(7)  Value 'CEETEST'.

01  Parms.
    05  AA                  PIC S9(4) BINARY   Value 14.
    05  BB                  PIC x(14)   Value 'SET SCREEN ON;'.

CALL Debugger USING Parms FC.
```

Example 2

A string of commands is passed to z/OS Debugger when it is started. After it gains control, z/OS Debugger sets a breakpoint at statement 23, runs the LIST commands and returns control to the program by running the GO command. The command string is already defined and assigned to the variable COMMAND-STRING by the following declaration in the DATA DIVISION of your program:

```
01 COMMAND-STRING.
   05 AA     PIC 99    Value 60 USAGE IS COMPUTATIONAL.
   05 BB     PIC x(60) Value 'AT STATEMENT 23; LIST (x); LIST (y); GO;'.
```

The result of the call is returned in the feedback code, using a variable defined as:

```
01  FC.
    02  CONDITION-TOKEN-VALUE.
    COPY CEEIGZCT.
```

```
        03  CASE-1-CONDITION-ID.
            04  SEVERITY    PIC S9(4) BINARY.
            04  MSG-NO      PIC S9(4) BINARY.
        03  CASE-2-CONDITION-ID
                  REDEFINES CASE-1-CONDITION-ID.
            04  CLASS-CODE  PIC S9(4) BINARY.
            04  CAUSE-CODE  PIC S9(4) BINARY.
        03  CASE-SEV-CTL    PIC X.
        03  FACILITY-ID     PIC XXX.
    02  I-S-INFO            PIC S9(9) BINARY.
```

in the DATA DIVISION of your program. You are not prompted for commands.

```
CALL "CEETEST" USING COMMAND-STRING FC.
```

Example: using CEETEST to start z/OS Debugger from PL/I

The following examples show how to use the Language Environment callable service CEETEST to start z/OS Debugger from PL/I programs.

Example 1

No command string is passed to z/OS Debugger at its invocation and no feedback code is returned. After it gains control, z/OS Debugger becomes active and prompts you for commands or reads them from a commands file.

```
CALL  CEETEST(*,*);    /*  omit arguments   */
```

Example 2

A command string is passed to z/OS Debugger at its invocation and the feedback code is returned. After it gains control, z/OS Debugger becomes active and executes the command string. Barring any further interruptions, the program runs to completion, where z/OS Debugger prompts for further commands.

```
DCL   ch  char(50)
      init('AT STATEMENT 10 DO; LIST(x); LIST(y); END; GO;');

DCL   1  fb,
         5  Severity  Fixed bin(15),
         5  MsgNo     Fixed bin(15),
         5  flags,
            8  Case    bit(2),
            8  Sev     bit(3),
            8  Ctrl    bit(3),
         5  FacID     Char(3),
         5  I_S_info  Fixed bin(31);


DCL   CEETEST  ENTRY  ( CHAR(*)  VAR  OPTIONAL,
         1   optional ,
           254 real fixed bin(15),     /* MsgSev     */
           254 real fixed bin(15),     /* MSGNUM     */
           254                         /* Flags      */,
             255 bit(2),               /* Flags_Case     */
             255 bit(3),               /* Flags_Severity */
             255 bit(3),               /* Flags_Control  */
           254 char(3),                /* Facility_ID    */
           254 fixed bin(31) )         /* I_S_Info    */
                 options(assembler) ;

CALL  CEETEST(ch, fb);
```

Example 3

This example assumes that you use predefined function prototypes and macros by including CEEIBMAW, and predefined feedback code constants and macros by including CEEIBMCT.

A command string is passed to z/OS Debugger that sets a breakpoint on every tenth executed statement. Once a breakpoint is reached, z/OS Debugger displays the current location information and continues the execution. After the CEETEST call, the feedback code is checked for proper execution.

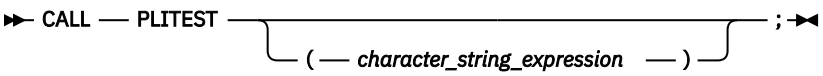
Note: The feedback code returned is either CEE000 or CEE2F2. There is no way to check the result of the execution of the command passed.

```
%INCLUDE CEEIBMAW;
%INCLUDE CEEIBMCT;
DCL 01 FC FEEDBACK;

/* if CEEIBMCT is NOT included, the following DECLARES need to be
   provided:          ---------- comment start -------------

Declare CEEIBMCT  Character(8) Based;
Declare ADDR      Builtin;
%DCL FBCHECK ENTRY;
%FBCHECK:  PROC( fbtoken, condition ) RETURNS( CHAR );
   DECLARE
      fbtoken    CHAR;
      condition  CHAR;
RETURN('(ADDR('||fbtoken||')->CEEIBMCT = '||condition||')');
%END FBCHECK;
%ACT FBCHECK;
                       ---------- comment end --------------- */

Call CEETEST('AT Every 10 STATEMENT * Do; Q Loc; Go; End;'||
             'List AT;', FC);

If ¬FBCHECK(FC, CEE000)
   Then Put Skip List('----> ERROR! in CEETEST call', FC.MsgNo);
```

Starting z/OS Debugger with PLITEST

For PL/I programs, the preferred method of Starting z/OS Debugger is to use the built-in subroutine PLITEST. It can be used in exactly the same way as CEETEST, except that you do not need to include CEEIBMAW or CEEIBMCT, or perform declarations.

The syntax is:



character_string_expression
Specifies a list of z/OS Debugger commands. If necessary, this is converted to a fixed-length string.

Note:

1. If z/OS Debugger executes a command in a CALL PLITEST command string that causes control to return to the program (GO for example), any commands remaining to be executed in the command string are discarded.
2. If you don't want to compile your program with hooks, you can use CALL PLITEST statements as hooks and insert them at strategic points in your program. If you decide to use this method, you still need to compile your application so that symbolic information is created.

The following examples show how to use PLITEST to start z/OS Debugger for PL/I.

Example 1
No argument is passed to z/OS Debugger when it is started. After gaining control, z/OS Debugger prompts you for commands.

```
CALL PLITEST;
```

Example 2
A string of commands is passed to z/OS Debugger when it is started. After gaining control, z/OS Debugger sets a breakpoint at statement 23, and returns control to the program. You are not prompted for commands. In addition, the `List Y;` command is discarded because of the execution of the GO command.

```
CALL PLITEST('At statement 23 Do; List X; End; Go; List Y;');
```

Example 3

Variable *ch* is declared as a character string and initialized as a string of commands. The string of commands is passed to z/OS Debugger when it is started. After it runs the commands, z/OS Debugger prompts you for more commands.

```
DCL  ch   Char(45) Init('At Statement 23 Do; List x; End;');
CALL PLITEST(ch);
```

Starting z/OS Debugger with the __ctest() function

You can also use the C and C++ library routine `__ctest()` or `ctest()` to start z/OS Debugger. Add:

```
#include <ctest.h>
```

to your program to use the `ctest()` function.

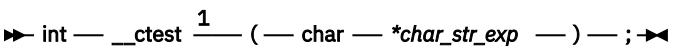
Note: If you do not include `ctest.h` in your source or if you compile using the option `LANGLVL(ANSI)`, you **must** use `__ctest()` function. The `__ctest()` function is not supported in CICS.

When a list of commands is specified with `__ctest()`, z/OS Debugger runs the commands in that list. If you specify a null argument, z/OS Debugger gets commands by reading from the supplied commands file or by prompting you. If control returns to your application before all commands in the command list are run, the remainder of the command list is ignored. z/OS Debugger will continue reading from the specified commands file or prompt for more input.

If you do not want to compile your program with hooks, you can use `__ctest()` function calls to start z/OS Debugger at strategic points in your program. If you decide to use this method, you still need to compile your application so that symbolic information is created.

Using `__ctest()` when z/OS Debugger is already initialized results in a reentry that is similar to a breakpoint.

The syntax for this option is:



Notes:

[1] The syntax for `ctest()` and `__ctest()` is the same.

char_str_exp

Specifies a list of z/OS Debugger commands.

The following examples show how to use the __ctest() function for C and C++.

Example 1

A null argument is passed to z/OS Debugger when it is started. After it gains control, z/OS Debugger prompts you for commands (or reads commands from the primary commands file, if specified).

```
__ctest(NULL);
```

Example 2

A string of commands is passed to z/OS Debugger when it is started. At statement 23, z/OS Debugger lists x and y, then returns control to the program. You are not prompted for commands. In this case, the command `list z;` is never executed because of the execution of the command `GO`.

```
__ctest("at line 23 {"
        "  list x;"
        "  list y;"
        "}"
        "go;"
        "list z;");
```

Example 3

Variable *ch* is declared as a pointer to character string and initialized as a string of commands. The string of commands is passed to z/OS Debugger when it is started. After it runs the string of commands, z/OS Debugger prompts you for more commands.

```
char *ch = "at line 23 list x;";
⋮
__ctest(ch);
```

Example 4

A string of commands is passed to z/OS Debugger when it is started. After z/OS Debugger gains control, you are not prompted for commands. z/OS Debugger runs the commands in the command string and returns control to the program by way of the GO command.

```
#include <stdio.h>
#include <string.h>

char *ch = "at line 23 printf(\"x.y is %d\n\", x.y); go;";
char buffer[35.132];

strcpy(buffer, "at change x.y;");

__ctest(strcat(buffer, ch));
```

Chapter 16. Starting z/OS Debugger in batch mode

Choose one of the following options to start z/OS Debugger in batch mode:

- Follow the instructions outlined in this section. This includes modifying your JCL to include the appropriate z/OS Debugger data sets and TEST runtime options.
- Use the z/OS Debugger Setup Utility (DTSU). DTSU can generate JCL that includes the appropriate z/OS Debugger data sets and TEST runtime options, and can submit your batch job. For instructions on how to use DTSU, refer to Chapter 14, “Starting z/OS Debugger from the IBM z/OS Debugger Utilities,” on page 115.

To start z/OS Debugger in batch mode without using DTSU, do the following steps:

1. Ensure that you have compiled your program with the TEST compiler option.
2. Modify the JCL that runs your batch program to include the appropriate z/OS Debugger data sets and to specify the TEST run-time option.
3. Run the modified JCL.

You can interactively debug an MVS batch job by choosing one of the following options:

- In full-screen mode using the Terminal Interface Manager. Follow the instructions in “Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 131.
- In remote debug mode. Follow the instructions in the topic "Preparing to debug" of the online help for the remote IDE.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Appendix F, “Notes on debugging in batch mode,” on page 493
Chapter 29, “Entering z/OS Debugger commands,” on page 261

Example: JCL that runs z/OS Debugger in batch mode

Sample JCL for a batch debug session for the COBOL program, EMPLRUN, is provided below. The job card and data set names need to be modified to suit your installation.

```
//DEBUGJCL JOB <appropriate JOB card information>
//* ************************************************************
//*  JCL to run a batch z/OS Debugger session
//*    Program EMPLRUN was previously compiled with the COBOL
//*    compiler TEST option
//* ************************************************************
//STEP1  EXEC PGM=EMPLRUN,
//         PARM='/TEST(,INSPIN,,)'      1
//*
//*  Include the z/OS Debugger SEQAMOD data set
//*
//STEPLIB     DD DISP=SHR,DSN=userid.TEST.LOAD
//            DD DISP=SHR,DSN=hlq.SEQAMOD
//*
//*  Specify a commands file with DDNAME matching the one
//*    specified in the /TEST runtime option above
//*  This example shows inline data but a data set could be
//*    specified like: //INSPIN  DD DISP=SHR,DSN=userid.TEST.INSPIN
//*
//INSPIN      DD *
           STEP;
           AT *
             PERFORM
                QUERY LOCATION;
                GO;
             END-PERFORM;
           GO;
           QUIT;
```

© Copyright IBM Corp. 1992, 2023

```
/*
//*
//*  Specify a log file for the debug session
//*    Log file can be a data set with LRECL >= 42 and <= 256
//*    For COBOL only, use LRECL <= 72 if you are planning to
//*    use the log file as a commands file in subsequent Debug
//*    Tool sessions.  You can specify the log file like:
//*    //INSPLOG   DD DISP=SHR,DSN=userid.TEST.INSPLOG
//*
//INSPLOG     DD SYSOUT=*,DCB=(LRECL=72,RECFM=FB,BLKSIZE=0)
//SYSPRINT    DD SYSOUT=*
//SYSUDUMP    DD DUMMY
//SYSOUT      DD SYSOUT=*
/*
//
```

Modifying the example to debug in full-screen mode

The example in “Example: JCL that runs z/OS Debugger in batch mode” on page 129 can be modified so that the batch program can be debugged in full-screen mode. Change line 1 to one of the following examples:

- To use full-screen mode using a dedicated terminal without Terminal Interface Manager, use the following statement:

```
//        PARM='/TEST(,INSPIN,,MFI%TRMLU001:)'
```

- To use full-screen mode using the Terminal Interface Manager, use the following statement:

```
//        PARM='/TEST(,INSPIN,,VTAM%USERABCD:)'
```

Chapter 17. Starting z/OS Debugger for batch or TSO programs

This section describes how to start z/OS Debugger to debug programs that run in the following situations:

- Programs that start in Language Environment
- Programs that start outside of Language Environment

Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can debug batch programs interactively by using full-screen mode using the Terminal Interface Manager or full-screen mode using a dedicated terminal without Terminal Interface Manager. Before you start this debugging session, contact your system administrator to verify that your system was customized to support this type of debugging session, and for instructions on how to access a terminal that supports this mode.

You need to decide whether you will use the z/OS Debugger Terminal Interface Manager. The z/OS Debugger Terminal Interface Manager enables you to associate a user ID with a specific dedicated terminal, which removes the need to update your runtime parameter string whenever the dedicated terminal LU name changes. This is the recommended method for most users.

To start a debugging session in full-screen mode using the Terminal Interface Manager, do the following steps:

1. Start two terminal emulator sessions in either of the following ways:
   - Two separate emulator windows.
   - If you use IBM Session Manager, you can select two sessions from the IBM Session Manager menu.

   In either case, connect the second emulator session to a terminal that can handle a full-screen mode using the Terminal Interface Manager and that also starts the Terminal Interface Manager.
2. On the first terminal emulator session, log on to TSO.
3. On the second terminal emulator session, provide your login credentials to the Terminal Interface Manager and press Enter. The login credentials can be your TSO user ID and password, PassTicket, password phrase, or MFA token.

   Notes:

   a. You are not logging on TSO. You are indicating that you want your user ID associated with this terminal LU.

   b. When the number of characters entered into the password field, including blanks, exceeds 8, the input is passed to the security system as a password phrase.

   c. To use PassTickets with Terminal Interface Manager, ensure that the PTKTDATA profile is defined following the rules for MVS batch jobs by your system programmer.

   A panel similar to the following panel is then displayed on the second terminal emulator session:

© Copyright IBM Corp. 1992, 2023

```
                    z/OS DEBUGGER TERMINAL INTERFACE MANAGER

EQAY001I Terminal TRMLU001 connected for user USER1
EQAY001I Ready for z/OS Debugger









               PF3=EXIT  PF10=Edit LE options data set
PF12=LOGOFF
```

The terminal is now ready to receive a z/OS Debugger full-screen mode using the Terminal Interface Manager session.

4. Edit the PARM string of your batch job so that you specify the TEST runtime parameter as follows:

   ```
   TEST(,,,VTAM%userid:*)
   ```

   Place a slash (/) before or after the parameter, depending on our programming language. *userid* is the TSO user ID that you provided to the Terminal Interface Manager.
5. Submit the batch job.
6. On the second terminal emulator session, a full-screen mode debugging session is displayed. Interact with it the same way you would with any other full-screen mode debugging session.
7. After you exit z/OS Debugger, the second terminal emulator session displays the panel and messages you saw in step 3. This indicates that z/OS Debugger can use this session again. (this will happen each time you exit from z/OS Debugger).
8. If you want to start another debugging session, return to step 5. If you are finished debugging, you can do one of the following tasks:
   - Close the second terminal emulator session.
   - Exit the Terminal Interface Manager by choosing one of the following options:
     - Press PF12 to display the Terminal Interface Manager logon panel. You can log in with the same ID or a different user ID.
     - Press PF3 to exit the Terminal Interface Manager.

To start a debugging session using a dedicated terminal without the z/OS Debugger Terminal Interface Manager, do the following steps:

1. Ask your system programmer if you need to specify a VTAM network identifier to communicate with the terminal LU you will use for display. If so, make a note of the network identifier.
2. Start two terminal emulator sessions. Connect the second emulator session to a terminal that can handle a full-screen mode debugging session through a dedicated terminal.
3. On the first terminal emulator session, log on to TSO.
4. On the second terminal emulator session, note the LU name of the terminal. If a session manager is displayed, exit from it.
5. Edit the PARM string of your batch job so that you specify the TEST runtime parameter in one of the following ways:

- `TEST(,,,MFI%luname:*)`
- `TEST(,,,MFI%network_identifier.luname:*)`

  Place a slash (/) before or after the parameter, depending on your programming language. *luname* is the VTAM LU name of the second terminal emulator. *network_identifier* is the name of the VTAM network node that contains *luname*.

6. Submit the batch job.
7. On the second terminal emulator session, a full-screen mode debugging session is displayed. Interact with it the same way you would with any other full-screen mode debugging session.
8. After you exit z/OS Debugger, a USSMSG10 or Telnet Solicitor Logon panel is displayed on the second terminal emulator session.
9. Go back to step 6 if you need to restart the debugging session.

Starting z/OS Debugger for programs that start in Language Environment

Choose one of the following options to start z/OS Debugger under MVS in TSO:

- You can follow the instructions outlined in this section. The instructions describe how to allocate all the files you need to start your debug session and how to start your program with the proper parameters.
- Use the z/OS Debugger Setup Utility (DTSU). DTSU helps you allocate all the files you need to start your debug session, and can start your program or submit your batch job. For instructions on using DTSU, refer to Chapter 14, “Starting z/OS Debugger from the IBM z/OS Debugger Utilities,” on page 115.

To start z/OS Debugger under MVS in TSO without using DTSU, do the following steps:

1. Ensure your program has been compiled with the TEST compiler option.
2. Ensure that the z/OS Debugger SEQAMOD library is in the load module search path.`SEQAMOD` must be placed before any other library in the load module search path that contains `CEEEVDBG` for z/OS Debugger to get control of a debug session.

   Note: High-level qualifiers and load library names are specific to your installation. Ask the person who installed z/OS Debugger the name of the data set. By default, the name of the data set ends in SEQAMOD. This data set might already be in the linklist or included in your TSO logon procedure, in which case you don't need to do anything to access it.
3. Allocate all other data sets containing files your program needs.
4. Allocate any z/OS Debugger files that you want to use. For example, if you want a session log file, allocate a data set for the session log file. Do not allocate the session log file to a terminal. For example, do not use `ALLOC FI(INSPLOG) DA(*)`.
5. Start your program with the TEST run-time option, specifying the appropriate suboptions, or include a call to `CEETEST`, `PLITEST`, or `__ctest()` in the program's source.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 13, “Writing the TEST runtime option string,” on page 107
“Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 131
“Recording your debug session in a log file” on page 172
Chapter 15, “Starting z/OS Debugger from a program,” on page 119

Related references

IBM z/OS Debugger Reference and Messages
z/OS Language Environment Programming Guide

Example: Allocating z/OS Debugger load library data set

The following example CLIST fragments show how you might allocate the z/OS Debugger load library data set (SEQAMOD) if it is not in the linklist or TSO logon procedure:

Example 1:

```
PROC 0 TEST
TSOLIB ACTIVATE DA('hlq.SEQAMOD')
END
```

Example 2:

```
PROC 0 TEST
TSOLIB DEACTIVATE
FREE FILE(SEQAMOD)
ALLOCATE DA('hlq.SEQAMOD') FILE(SEQAMOD) SHR REUSE
TSOLIB ACTIVATE FILE(SEQAMOD)
END
```

If you store either example CLIST in `MYID.CLIST(DTSETUP)`, you can run the CLIST by entering the following command at the TSO READY prompt:

```
EXEC 'MYID.CLIST(DTSETUP)'
```

The CLIST runs and the appropriate z/OS Debugger data set is allocated.

Example: Allocating z/OS Debugger files

The following example illustrate how you can use the command line to allocate the preferences and log files, then start the COBOL program `tstscrpt` with the TEST run-time option:

```
ALLOCATE FILE(inspprefe) data set(setup.pref) REUSE
ALLOCATE FILE(insplog) data set(session.log) REUSE
CALL 'USERID1.MYLIB(TSTSCRPT)' '/TEST'
```

The example illustrates that the default z/OS Debugger run-time suboptions and the default Language Environment run-time options were assumed.

The following example illustrates how you can use a CLIST to define the preferences file (`debug.preferen`) and the log file (`debug.log`), then start the C program `prog1` with the TEST run-time option:

```
ALLOC FI(insplog)  DA(debug.log)              REUSE
ALLOC FI(inspprefe) DA(debug.preferen)        REUSE

CALL 'MYID.MYQUAL.LOAD(PROG1)' +
     ' TRAP(ON) TEST(,*,;,inspprefe)/'
```

All the data sets must exist before starting this CLIST.

Starting z/OS Debugger for programs that start outside of Language Environment

To debug an MVS batch or TSO program that has an initial program that does not run under the control of Language Environment, including non-Language Environment COBOL programs, use the z/OS Debugger program EQANMDBG to start z/OS Debugger.

If you need to debug a non-Language Environment program where EQANMDBG is used to start z/OS Debugger, and your program frees SUBPOOL 1 (which z/OS Debugger uses itself by default), you need to specify a new parm to EQANMDBG.

The parameter is *NONLESP(nnn)* where *nnn* is a SUBPOOL number from 2 - 127, that specifies the SUBPOOL for z/OS Debugger to use for its storage.

If the initial program does run under the control of Language Environment and subsequent programs run outside the control of Language Environment, you can use the methods described in "Starting z/OS Debugger for programs that start in Language Environment" on page 133 to debug all the programs.

To start z/OS Debugger by using EQANMDBG, do one of the following options:

- By using the IBM z/OS Debugger Utilities option 2, **z/OS Debugger Setup File** to run the programs either under TSO or in MVS batch.
- By modifying the MVS JCL, TSO CLIST or REXX EXEC that you use to start your program, making the following changes:
  - Change the name of the program to be started to EQANMDBG.
  - Make one of the following updates:
    - Change the parameters by adding the name of the program to be debugged and any required z/OS Debugger run-time parameters. See "Passing parameters to EQANMDBG by using only the PARM string" on page 136 for instructions.
    - Add a EQANMDBG DD statement that provides the name of the program to be debugged and any required z/OS Debugger run-time parameters. See "Passing parameters to EQANMDBG using only the EQANMDBG DD statement" on page 136 for instructions.
    - Change the parameters by adding the name of the program to be debugged, and add an EQANMDBG DD statement that provides any required z/OS Debugger run-time parameters. See "Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement" on page 136 for instructions.
    - Verify that the z/OS Debugger SEQAMOD and SEQABMOD libraries are in the load module search path. SEQAMOD must be placed before any other library in the load module search path that contains CEEEVDBG for z/OS Debugger to get control of a debug session.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 14, "Starting z/OS Debugger from the IBM z/OS Debugger Utilities," on page 115

Passing parameters to EQANMDBG

When you modify your JCL, CLIST, or REXX EXEC to start EQANMDBG, you pass the following parameters to EQANMDBG:

- The name of the user program to be debugged (required)
- Any of the following run-time options (optional):
  - COUNTRY to specify a country code for z/OS Debugger
  - NATLANG to specify the national language used to communicate with z/OS Debugger
  - NONLESP to specify the SUBPOOL for z/OS Debugger to use for its storage
  - TEST to specify z/OS Debugger options. For example, you can use suboptions of the TEST run-time option to specify the data sets that contain z/OS Debugger commands and preferences. You can use suboptions to specify whether to use a remote debug mode session or a full-screen mode using the Terminal Interface Manager session.
  - TRAP to specify whether z/OS Debugger is to intercept abends.

You can specify these parameters in one of following ways:

- "Passing parameters to EQANMDBG by using only the PARM string" on page 136
- "Passing parameters to EQANMDBG using only the EQANMDBG DD statement" on page 136
- "Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement" on page 136

Refer to the following topics for more information related to the material discussed in this topic.

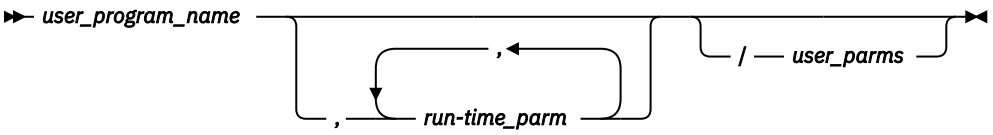
Related references

z/OS Debugger run-time options (*IBM z/OS Debugger Reference and Messages*)

Passing parameters to EQANMDBG by using only the PARM string

The easiest way to pass parameters to EQANMDBG is to modify the PARM string to contain the name of the program to be debugged, optionally followed by any of the z/OS Debugger run-time options and the parameters required by your program.

The syntax for this string is:



The following table compares how a sample JCL statement might look like after you modify the PARM string:

Original sample JCL	Modified sample JCL

`//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'`
`  ...`
`//`
	`//STEP1 EXEC PGM=EQANMDBG,`
	`// PARM='MYPROG,NATLANG(UEN)/ABC,X(12)'`
	`  ...`
	`//`

Passing parameters to EQANMDBG using only the EQANMDBG DD statement

If the user parameter string that you are passing to your program is too long to add the necessary z/OS Debugger parameters to the PARM string, you can leave the PARM string unchanged and pass all required parameters to z/OS Debugger by using the EQANMDBG DD statement.

When you add an EQANMDBG DD statement to your JCL or allocate the EQANMDBG file in your TSO session, it can point to a data set with any RECFM (F, V, or U) and any LRECL. The data set must contain one or more lines. If it contains more than one line, all trailing blanks are removed from each line. However, each line is assumed to start in column 1 with any leading blanks considered to be part of the parameter data. Sequence numbers are not supported in this file.

The following table compares original JCL and modified JCL:

Original JCL	Modified JCL

`//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'`
`  ...`
`//`
	`//STEP1 EXEC PGM=EQANMDBG,`
	`// PARM='ABC,X(12)'`
	`//EQANMDBG DD *`
	`MYPROG,`
	`TEST(ALL,INSPIN,,MFI:*),`
	`NATLANG(ENU)`
	`/*`
	`  ...`
	`//`

Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement

With this method you can put the name of the user program to be debugged as part of the PARM string, and then specify all other z/OS Debugger run-time options by using the EQANMDBG DD statement.

This can be desirable if you need to pass the same run-time parameters to several programs, you have room in the PARM string to add the name of the program to be debugged, but you do not have room to add all of the run-time parameters to the PARM string.

When you use this method, you must do the following:

- Include an EQANMDBG DD statement that includes, at a minimum, an asterisk as the first positional parameter to indicate that the user-program name is to be taken from the PARM string.
- Modify the PARM string to include the user-program name followed by a slash at the beginning of the PARM string.

The following table compares original JCL and modified JCL:

Original JCL	Modified JCL

`//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'`
`  ...`
`//`
	`//STEP1 EXEC PGM=EQANMDBG,`
	`// PARM='MYPROG/ABC,X(12)'`
	`//EQANMDBG DD *`
	`*,TEST(ALL,INSPIN,,MFI:*),NATLANG(ENU)`
	`/*`
	`  ...`
	`//`

Example: Modifying JCL that invokes an assembler Db2 program running in a batch TSO environment

The following example shows a portion of JCL that invokes an assembler Db2 program and the modifications you make to this portion of the JCL to start z/OS Debugger.

Original sample JCL	Modified sample JCL

`//RUN EXEC PGM=IKJEFT01,DYNAMNBR=20`
`//SYSTSIN DD *`
`  DSN SYSTEM(`*`Db2_subsystem_id`*`)`
`  RUN  PROGRAM(MYPGM) PLAN(MYPGM) -`
`       PARM('`*`program-parameters`*`')`
`  END`
`/*`
`// ... other DD statements as needed ...`
`// ... for TSO and the application ...`
	`//RUN EXEC PGM=IKJEFT01,DYNAMNBR=20`
	`//SYSTSIN DD *`
	`  DSN SYSTEM(`*`Db2_subsystem_id`*`)`
	`  RUN  PROGRAM(EQANMDBG) PLAN(MYPGM) -`
	`       PARM('`*`program-parameters`*`')`
	`  END`
	`/*`
	`//EQANMDBG DD *`
	`MYPGM,TEST(,,,VTAM%`*`user-id`*`:)`
	`/*`
	`// ... other DD statements as needed ...`
	`// ... for TSO and the application ...`

Chapter 18. Starting z/OS Debugger under CICS

This topic compares the different methods you can use to start z/OS Debugger and gives instructions on each method. This topic assumes you have completed the following tasks:

- Ensured that all of the required installation and configuration steps for CICS Transaction Server, Language Environment, and z/OS Debugger have been completed. For more information, refer to the installation and customization guides for each product.
- Completed all the tasks in the following topics:
  - Chapter 4, “Planning your debug session,” on page 25
  - Chapter 5, “Updating your processes so you can debug programs with z/OS Debugger,” on page 59
  - Chapter 10, “Preparing a CICS program,” on page 81

Comparison of methods for starting z/OS Debugger under CICS

There are several different mechanisms available to start z/OS Debugger under CICS. Each mechanism has a different advantage and are listed below:

- DTCN is a full-screen CICS transaction that z/OS Debugger provides. By using DTCN, you can create a profile that contains a pattern of CICS resource names that identify a task that you want to debug. You can dynamically change any Language Environment TEST or NOTEST runtime option that your application was originally link-edited with. You can also use DTCN to dynamically change any other Language Environment runtime options that are not specific to z/OS Debugger which are defined in your CICS installation except the STACK option.

  DTCN has the following advantages and differences compared to CADP:

  - Provides a view to create CICS profiles (DTCN profiles) for remote users. For more information, see the "Working with the z/OS Debugger Profiles view" topic in IBM Documentation.
  - Provides two mechanisms for managing debug profiles:
    1. In a Temporary Storage Queue (TSQ) - debug profiles are owned by the terminal that created them. The debug profiles are deleted if the terminal that created the profile is disconnected or the CICS region is terminated. Also, a single terminal can have only one debug profile.
    2. In a VSAM file - debug profiles are owned by the user ID that created them. The debug profiles persist through disconnections or CICS region restarts. Also, a single terminal can have multiple debug profiles that are created by using different users.
  - Provides general and field sensitive help.
  - Provides a service that deletes ownerless profiles from the DTCN repository. See "Deleting DTCN profiles with the DTCN LINK service" in the *IBM z/OS Debugger Customization Guide*.
  - Displays both the generated and saved repository runtime strings.
  - Provides the following additional CICS resources for identifying a task that you want to debug:
    - Eight pairs of Load Module and CU Names (including wildcards)
    - IP Name/Address
    - Commarea Offset
    - Commarea Data
    - Container Name
    - Container Offset
    - Container Data
    - URM Debugging

© Copyright IBM Corp. 1992, 2023

  - Provides a **EQAOPTS File** field. You can use this field to specify a file that contains a set of z/OS Debugger EQA0PTS commands for the debug session.

  To learn how to set up profiles by using DTCN, see Chapter 10, “Preparing a CICS program,” on page 81.

- CADP is a CICS transaction for you to manage debugging profiles. This transaction is available with CICS Transaction Server for z/OS Version 2 Release 3.

  CADP has the following advantages and differences compared to DTCN:

  - With CADP, you can add multiple profiles from the same display device by using a single program name. There is no limit to the number of supported profiles. You can specify the program names by using a wildcard.
  - CADP provides the same abilities as DTCN for managing debug profiles for Language Environment applications. CADP can also manage debug profiles for Java applications, Enterprise Java Beans (EJBs), and CORBA stateless objects.
  - CADP profiles are persistent, and are kept in VSAM files. Persistence means that if a CADP profile is present before a CICS region is restarted, the CADP profile is present after the CICS region is restarted.
  - CADP profiles can be shared across a CICSPLEX.
- Language Environment CEEUOPT module link-edited into your application, containing an appropriate TEST option, which tells Language Environment to start z/OS Debugger every time the application is run.

  This mechanism can be useful during initial testing of new code when you will want to run z/OS Debugger frequently.
- A compiler directive within the application, such as `#pragma runopts(test)` (for C and C++) or `CALL CEETEST`.

  These directives can be useful when you need to run multiple debug sessions for a piece of code that is deep inside a multiple enclave or multiple CU application. The application runs without z/OS Debugger until it encounters the directive, at which time z/OS Debugger is started at the precise point that you specify. With `CALL CEETEST`, you can even make the invocation of z/OS Debugger conditional, depending on variables that the application can test.

If your program uses several of these methods, the order of precedence is determined by Language Environment. For more information about the order of precedence for Language Environment run-time options, see *z/OS Language Environment Programming Guide*.

Starting z/OS Debugger under CICS by using DTCN

If a DTCN profile exists, when a CICS program starts, z/OS Debugger analyzes the program's resources to see if they match a profile. If z/OS Debugger finds a match, z/OS Debugger starts a debugging session for that program. If multiple profiles exist, z/OS Debugger selects the profile with the greatest number of resources that match the program. If two programs have an equal number of matching resources, z/OS Debugger selects the older profile.

Before you begin, verify that you prepared your CICS program as instructed in Chapter 10, “Preparing a CICS program,” on page 81.

To start z/OS Debugger under CICS by using DTCN, do the following steps:

1. If you chose screen control mode, start the DTSC transaction on the terminal you specified in the **Display Id** field.
2. Run your CICS programs. If z/OS Debugger identifies a task that matches a DTCN profile, z/OS Debugger starts. If you chose screen control mode, press Enter on the terminal running the DTSC transaction to connect to z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27

Ending a CICS debugging session that was started by DTCN

After you have finished debugging your program, use DTCN again to turn off your debug profile by pressing PF6 to delete your debug profile and then pressing PF3 to exit. You do not need to remove EQADCCXT from the load module; in fact, it's a good idea to leave it there for the next time you want to start z/OS Debugger.

Example: How z/OS Debugger chooses a CICS program for debugging

For example, consider the following two profiles:

- First, profile A is saved, specifying resource CU PROG1
- Later, profile B is saved, specifying resource User Id USER1

When PROG1 is run by USER1, profile A is used.

If this situation occurs, an error message is displayed on the system console, suggesting that you should specify additional resources. In the above example, each profile should specify both a User Id and a CU resource.

Starting z/OS Debugger for CICS programs by using CADP

Before you begin, verify that you prepared your CICS program as instructed in Chapter 10, "Preparing a CICS program," on page 81.

To start z/OS Debugger under CICS by using CADP, do the following steps:

1. If you chose screen control mode, start the DTSC transaction on the terminal you specified in the **Display Id** field.
2. Run your CICS programs. If z/OS Debugger identifies a task that matches a CADP profile, z/OS Debugger starts. If you chose screen control mode, press Enter on the terminal running the DTSC transaction to connect to z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Creating and storing debugging profiles with CADP" on page 92

Related references

CICS Supplied Transactions

Starting z/OS Debugger under CICS by using CEEUOPT

To request that Language Environment start z/OS Debugger every time the application is run, assemble a CEEUOPT module with an appropriate TEST run-time option. It is a good idea to link-edit the CEEUOPT module into a library and just add an `INCLUDE LibraryDDname(CEEUOPT-MemberName)` statement to the link-edit options when you link your application. Once the application program has been placed in the load library (and NEWCOPY'd if required), whenever it is run z/OS Debugger will be started.

z/OS Debugger runs in the mode defined in the TEST run-time option you supplied, normally Single Terminal mode, although you could provide a primary commands file and a log file and not use a terminal at all.

To start z/OS Debugger, simply run the application. Don't forget to remove the CEEUOPT containing your TEST run-time option when you have finished debugging your program.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 13, "Writing the TEST runtime option string," on page 107

Starting z/OS Debugger under CICS by using compiler directives

When compile-directives are processed by your program, z/OS Debugger will be started in single terminal mode (this method supports only single terminal mode).

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Starting z/OS Debugger with CEETEST" on page 119

Chapter 19. Starting a debug session

You can start z/OS Debugger by using the Language Environment TEST run-time option in one of the following ways:

- Using the z/OS Debugger Setup Utility (DTSU). DTSU helps you allocate files and can start your program. The methods listed below describe how you manually perform the same tasks.

  Note: DTSU is not available in IBM Developer for z/OS (non-Enterprise Edition), IBM Z and Cloud Modernization Stack (Wazi Code).
- For TSO programs that start in Language Environment, start your program with the TEST run-time option as described in "Starting z/OS Debugger for programs that start in Language Environment" on page 133.
- For MVS batch programs that start in Language Environment, start your Language Environment program with the TEST runtime option and specify the appropriate suboptions, as described in Chapter 16, "Starting z/OS Debugger in batch mode," on page 129.
- For MVS batch programs that do not start in Language Environment, start the non-Language Environment z/OS Debugger (EQANMDBG), and pass your program name and the TEST runtime option. Specify the appropriate suboptions, as described in "Starting z/OS Debugger for programs that start outside of Language Environment" on page 134.
- For CICS, make sure z/OS Debugger is installed in your CICS region. Enter DTCN or CADP (in CICS Transaction Server for z/OS Version 2 Release 3 and later) to start the z/OS Debugger control transaction. Enter the name of the transaction and program that you want to debug and any other criteria, such as terminal id or user id. If you are using DTCN, press PF4 to save the default debugging profile, then press PF3 to exit the DTCN transaction. You are now setup to start your transaction and begin a debugging session.

  If you are using CADP to manage your debugging profiles, make sure that the DEBUGTOOL system initialization parameter is set to YES.
- For CICS transactions that run non-Language Environment assembler programs or non-Language Environment COBOL programs, verify with your system administrator that the z/OS Debugger CICS global user exits are installed and active. If exits are active and the non-Language Environment assembler or non-Language Environment COBOL programs are defined in a DTCN or CADP debugging profile, z/OS Debugger will debug the non-Language Environment assembler or non-Language Environment COBOL programs. These programs must be the first program to run at a CICS Link Level (for example, at the start of a task or through a CICS LINK or XCTL request).

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 14, "Starting z/OS Debugger from the IBM z/OS Debugger Utilities," on page 115
"Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
"Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44
"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27
"Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34
"Ending a full-screen debug session" on page 195
"Entering commands on the session panel" on page 158
"Passing parameters to EQANMDBG" on page 135

Related references

"z/OS Debugger session panel" on page 149

© Copyright IBM Corp. 1992, 2023

Chapter 20. Starting z/OS Debugger in other environments

You can start z/OS Debugger to debug batch programs from Db2 stored procedures.

Starting z/OS Debugger from Db2 stored procedures

Before you run the stored procedure, verify that you have completed all the instructions in Chapter 9, "Preparing a Db2 stored procedures program," on page 79.

To verify that the stored procedure has started, enter the following Db2 Display command, where *xxxx* is the name of the stored procedure:

```
Display Procedure(xxxx)
```

If the stored procedure is not started, enter the following Db2 command:

```
Start procedure(xxxx)
```

If z/OS Debugger or the remote debugger do not start when the stored procedure calls them, verify that you have correctly specified connection information (for example, the TCP/IP address and port number) in the Language Environment EQAD3CXT exit routine or the Db2 catalog.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 4, "Planning your debug session," on page 25

© Copyright IBM Corp. 1992, 2023

Part 4. Debugging your programs in full-screen mode

Note: This part is not applicable to IBM Developer for z/OS (non-Enterprise Edition), IBM Z and Cloud Modernization Stack (Wazi Code).

© Copyright IBM Corp. 1992, 2023

Chapter 21. Using full-screen mode: overview

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The topics below describe the z/OS Debugger full-screen interface, and how to use this interface to perform common debugging tasks.

Debugging your programs in full-screen mode is the easiest way to learn how to use z/OS Debugger, even if you plan to use batch or line modes later.

The following list describes the maximum screen size supported by z/OS Debugger for a particular type of terminal:

- In full screen mode, you can use any screen size supported by ISPF.
- In full-screen mode using the Terminal Interface Manager or a CICS terminal, you can use a maximum screen size (number of rows times number of columns) of 10922. If the number of rows times the number of columns is not less than 10923, z/OS Debugger displays a WTO error message and abends.

Note: The PF key definitions used in these topics are the default settings.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 19, “Starting a debug session,” on page 143
“Ending a full-screen debug session” on page 195
“Entering commands on the session panel” on page 158
“Navigating through z/OS Debugger windows” on page 164
“Recording your debug session in a log file” on page 172
“Setting breakpoints to halt your program at a line” on page 174
“Setting breakpoints in a load module that is not loaded or in a program that is not active” on page 174
“Stepping through or running your program” on page 175
“Displaying and monitoring the value of a variable” on page 182
“Displaying error numbers for messages in the Log window” on page 193
“Displaying a list of compile units known to z/OS Debugger” on page 194
“Requesting an attention interrupt during interactive sessions” on page 194
Chapter 25, “Debugging a C program in full-screen mode,” on page 221
Chapter 26, “Debugging a C++ program in full-screen mode,” on page 231
Chapter 22, “Debugging a COBOL program in full-screen mode,” on page 197
Chapter 24, “Debugging a PL/I program in full-screen mode,” on page 213

z/OS Debugger session panel

The z/OS Debugger session panel contains a header with information about the program you are debugging, a command line, and up to three physical windows. A physical window is the space on the screen dedicated to the display of a specific type of debugging information. The debugging information is organized into the following types, called logical windows:

Monitor window
Variables and their values, which you can display by entering the `SET AUTOMONITOR ON` and `MONITOR` commands.

Source window
The source or listing file, which z/OS Debugger finds or you can specify where to find it.

Log window
The record of your interactions with z/OS Debugger and the results of those interactions.

© Copyright IBM Corp. 1992, 2023

Memory window
Section of memory, which you can select by entering the MEMORY command.

Each physical window can be assigned only one logical window. The physical window assumes the name of the logical window, so when you enter commands that affect the physical window (for example, the WINDOW SIZE command), you identify the physical window by providing the name of its assigned logical window. Physical windows can be closed (not displayed), but at least one physical window must remain open at any time.

The z/OS Debugger session panel below shows the default layout which contains three physical windows: one for the Monitor window 1, a second for the Source window 2, and the third for the Log window 3.

```
COBOL    LOCATION: DTAM01 :> 109.1
Command ===>                                                 Scroll ===> PAGE
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 7
*************************** TOP OF MONITOR ********************************
                              ----+----1----+----2----+----3----+----4----
0001  1 NUM1                   0000000005
0002  2 NUM4                  '1111'          1
0003  3 WK-LONG-FIELD-2       '123456790 223456790 323456790 423456790 5234
0004                          56790 623456790 723456790 8234567890 9234567
0005                          90 023456790 123456790 223456790 323456790 4
0006                          23456790 5234567890 623456790 723456790 8234
SOURCE: DTAM01 ---1----+----2----+----3----+----4----+----5--- LINE: 107 OF 196
     107       *    SINGLE DATAITEM IN A STRUCTURE                          .
     108       *------------------------------------------------------------ .
     109            ADD 1 TO AA-NUM1      2                                  .
     110                                                                     .
     111       *------------------------------------------------------------ .
     112       *    SINGLE DATAITEM IN A STRUCTURE - QUALIFIED               .
LOG 0----+----1----+----2----+----3----+----4----+----5----+---- LINE: 40 OF 43
0040  MONITOR
0041    LIST NUM4 ;
0042  MONITOR                                  3
0043    LIST WK-LONG-FIELD-2 ;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Customizing the layout of physical windows on the session panel” on page 252

Related references
“Session panel header” on page 150
“Monitor window” on page 152
“Source window” on page 152
“Log window” on page 153
“Memory window” on page 154

Session panel header

The first few lines of the z/OS Debugger session panel contain a command line and header fields that display information about the program that you are debugging.

Below is an example header for a C program.

```
 C  1        LOCATION: MYID.SOURCE(TSTPGM1):>248           2
Command ===>  3                                 SCROLL ===> PAGE    4
                          5
```

Below is an example header for a COBOL program.

```
COBOL  1             LOCATION:  XYZPROG::>SUBR:>118   2
Command ===>  3                                 SCROLL ===> PAGE    4
                          5
⋮
```

The header fields are described below.

1 Assemble, C, COBOL, LX COBOL, Disassem, or PL/I
The name of the current programming language. This language is not necessarily the programming language of the code in the Source window. The language that is displayed in this field determines the syntax rules that you must follow for entering commands.

Note:

1. z/OS Debugger does not differentiate between C and C++ programs. If there is a C++ program in the Source window, only C is displayed in this field.
2. LX COBOL is used to indicate LangX COBOL.

2 LOCATION
The program unit name and statement where execution is suspended, usually in the form *compile unit:>nnnnnn*.

In the C example above, execution in `MYID.SOURCE(TSTPGM1)` is suspended at line 248.

In the COBOL example above, execution in XYZPROG is suspended at XYZPROG`::`>SUBR`:`>118, or line 118 of subroutine SUBR.

If you are replaying recorded statements, the word "LOCATION" is replaced by PBK<LOC or PBK>LOC. The < and > symbols indicate whether the recorded statements are being replayed in the backward (<) or forward (>) direction.

If you are using the Enterprise PL/I compiler or the C/C++ compiler, the compile unit name is the entire data set name of the source. If the setting for `LONGCUNAME` is `ON` (the default) to display the CU name in long form, the name might be truncated. If your PL/I program was compiled with the following compiler and running in the following environment, the package statement or the name of the main procedure is displayed.

- Enterprise PL/I for z/OS, Version 3.5, compiler with the PTFs for APARs PK35230 and PK35489 applied, or Enterprise PL/I for z/OS, Version 3.6 or later

3 COMMAND
The input area for the next z/OS Debugger command. You can enter any valid z/OS Debugger command here.

4 SCROLL
The number of lines or columns that you want to scroll when you enter a `SCROLL` command without an amount specified. To hide this field, enter the `SET SCROLL DISPLAY OFF` command. To modify the scroll amount, use the `SET DEFAULT SCROLL` command.

The value in this field is the operand applied to the `SCROLL UP`, `SCROLL DOWN`, `SCROLL LEFT`, and `SCROLL RIGHT` scrolling commands. Table 24 on page 151 lists all the scrolling commands.

Table 24. Scrolling commands

Command	Description
n	Scroll by *n* number of lines.
`HALF`	Scroll by half a page.
`PAGE`	Scroll by a full page.
`TOP`	Scroll to the top of the data.
`BOTTOM`	Scroll to the bottom of the data.
`MAX`	Scroll to the limit of the data.
`LEFT` *x*	Scroll to the left by *x* number of characters.
`RIGHT` *x*	Scroll to the right by *x* number of characters.
`CURSOR`	Position of the cursor.
`TO` *x*	Scroll to line *x*, where *x* is an integer.

5 Message areas
Information and error messages are displayed in the space immediately below the command line.

Source window

```
1 SOURCE: MULTCU ---1----+----2----+----3----+----4----+----5----+ LINE: 70 OF 85
      70        PROCEDURE DIVISION.                                        .
      71       ************************************************************ .
      72       * THIS IS THE MAIN PROGRAM AREA.  This program only displays   .
      73       *         text. 3                                            .
      74       ************************************************************ .

  2   75            DISPLAY "MULTCU COBOL SOURCE STARTED." UPON CONSOLE.      .
      76            MOVE 25 TO PROGRAM-USHORT-BIN.                            .
      77            MOVE -25 TO PROGRAM-SSHORT-BIN.                           . 4
      78            PERFORM TEST-900.                                         .
      79            PERFORM TEST-1000.                                        .
      80            DISPLAY "MULTCU COBOL SOURCE ENDED." UPON CONSOLE.        .
```

The Source window displays the source file or listing. The Source window has four parts, described below.

1 Header area
Identifies the window, shows the compile unit name, and shows the current position in the source or listing.

2 Prefix area
Occupies the left-most eight columns of the Source window. Contains statement numbers or line numbers you can use when referring to the statements in your program. You can use the prefix area to set, display, and remove breakpoints with the prefix commands AT, CLEAR, ENABLE, DISABLE, QUERY, and SHOW.

3 Source display area
Shows the source code (for a C and C++ program), the source listing (for a COBOL, LangX COBOL, or PL/I program), a pseudo assembler listing (for an assembler program), or the disassembly view (for programs without debug information) for the currently qualified program unit. If the current executable statement is in the source display area, it is highlighted.

4 Suffix area
A narrow, variable-width column at the right of the screen that z/OS Debugger uses to display frequency counts. It is only as wide as the largest count it must display.

The suffix area is optional. To show the suffix area, enter `SET SUFFIX ON`. To hide the suffix area, enter `SET SUFFIX OFF`. You can also set it on or off with the *Source Listing Suffix* field in the Profile Settings panel.

The labeled header line for each window contains a scale and a line counter. If you scroll a window horizontally, the scale also scrolls to indicate the columns displayed in the window. The line counter indicates the line number at the top of a window and the total number of lines in that window. If you scroll a window vertically, the line counter reflects the top line number currently displayed in that window.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Entering prefix commands on specific lines or statements” on page 161
“Customizing profile settings” on page 254

Monitor window

The Monitor window displays the names and values of variables selected by the `SET AUTOMONITOR` or `MONITOR` commands.

The following diagram shows the default Monitor window and highlights the parts of the Monitor window:

```
COBOL    LOCATION: DTAM01 :> 109.1
Command ===>                                                 Scroll ===> PAGE
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 7
******************************** TOP OF MONITOR *******************************
                                  -----+----1----+----2----+- 1 --3----+----4--
0001  1 NUM1                       0000000005
0002  2 NUM4                      '1111'          2
0003  3 WK-LONG-FIELD-2           '123456790 223456790 323456790 423456790 5234
0004      3                        56790 623456790 723456790 8234567890 9234567
0005                               90 023456790 123456790 223456790 323456790 4
0006  4                            23456790 5234567890 623456790 723456790 8234
0007  4 HEX-NUM1                 X'ABCD 1234'
```

1
Monitor value scale, which provides a reference to help you measure the column position in the Monitor value area.

2
Monitor value area, where z/OS Debugger displays the values of the variables. z/OS Debugger extends the display to the right up to the full width of the displayable area of the Monitor window.

3
Monitor name area, where z/OS Debugger displays the names of the variables.

4
Monitor reference number area, where z/OS Debugger displays the reference number it assigned to a variable.

When you enter the `MONITOR LIST`, `MONITOR QUERY`, `MONITOR DESCRIBE`, and `SET AUTOMONITOR` commands, z/OS Debugger displays the output in the Monitor window. If this window is not open, z/OS Debugger opens it when you enter a `MONITOR` or `SET AUTOMONITOR` command.

By default, the Monitor window displays a maximum of 1000 lines. You can change this maximum by using the `SET MONITOR LIMIT` command. However, monitoring large amounts of data can use large amounts of storage, which might create problems. Verify that there is enough storage available to monitor large data items or data items that contain a large number of elements. To find out the current maximum, enter the `QUERY MONITOR LIMIT` command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Adding variables to the Monitor window” on page 184
“Replacing a variable in the Monitor window with another variable” on page 185
“Adding variables to the Monitor window automatically” on page 186
“Scrolling through the physical windows” on page 165

Related references
"SET MONITOR command" in *IBM z/OS Debugger Reference and Messages*
"QUERY command" in *IBM z/OS Debugger Reference and Messages*

Log window

```
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6 LINE: 6 OF 14
0007  MONITOR
0008    LIST PROGRAM-USHORT-BIN ;
0009  MONITOR
0010    LIST PROGRAM-SSHORT-BIN ;
0011  AT 75 ;
0012  AT 77 ;
0013  AT 79 ;
0014  GO ;
```

The Log window records and displays your interactions with z/OS Debugger.

At the beginning of a debug session, if you have specified any of the following files, the Log window displays messages indicating the beginning and end of any commands issued from these files:

- global preferences file
- preferences file
- commands file

If a global preferences file exists, the data set name of the global preferences file is displayed.

The following commands are not recorded in the Log window.

PANEL
FIND
CURSOR
RETRIEVE
SCROLL
WINDOW
IMMEDIATE
QUERY prefix command
SHOW prefix command

If SET INTERCEPT ON is in effect for a file, that file's output also appears in the Log window.

You can optionally exclude STEP and GO commands from the log by specifying SET ECHO OFF.

Commands that can be used with IMMEDIATE, such as the SCROLL and WINDOW commands, are excluded from the Log window.

By default, the Log window keeps 1000 lines for display. The default value can be changed by one of the following methods:

- The system administrator changes it through a global preferences file.
- You can change it through a preferences file.
- You can change it by entering SET LOG KEEP *n*, where *n* is the number of lines you want kept for display

The maximum number of lines is determined by the amount of storage available.

The labeled header line for each window contains a scale and a line counter. If you scroll a window horizontally, the scale also scrolls to indicate the columns displayed in the window. The line counter indicates the line number at the top of a window and the total number of lines in that window. If you scroll a window vertically, the line counter reflects the top line number currently displayed in that window.

Memory window

The Memory window displays the contents of memory. The following figure highlights the parts of the Memory window.

```
MEMORY---1----+----2----+----3----+----4----+----5----+----6----+----7----+- 1
History:  24702630         2505A000
 2
Base address:  265B1018  Amode: 31
+00000 265B1018   11C3D6C2 D6D34040 4011D3D6 C3C1E3C9   | .COBOL   .LOCATI |
+00010 265B1028   D6D57A12 D7D9D6C7 F1407A6E 40F4F44B   | ON:.PROG1 :> 44. |
+00020 265B1038   F1404040 40404040 40404040 40404040   | 1                |
+00030 265B1048   40404040 40404040 40404040 40404040   |        6         |
+00040 265B1058   40404040 40404040 40404040 40404040   |                  |
+00050 265B1068   11C39694 94819584 117E7E7E 6E009389   | .Command.===>.li |
+00060 265B1078   A2A340A2 A3969981 87854DA2 A399F16B   | st storage(str1, |
+00070 265B1088   F3F25D40 40404040 40404040 40404040   | 32)              |
 3          4                     5
```

1 Header area
The header area identifies the window and contains a scale.

2 Information area
The information area displays a memory history of up to 8 base addresses. The information area also displays the address mode and up to 8 unique base addresses.

The following sections are collectively known as the memory dump area.

3 Offset column
The offset column displays the offset from the base address of the line of data in memory.

4 Address column
The address column displays the low-order 32 bits of the starting address of the line of data in memory.

5 Hexadecimal data column
The hexadecimal data area displays data in hexadecimal format. Each line displays 16 bytes of memory in four 4 byte groups.

6 Character data column
The character data area displays data in character format. Each line displays 16 bytes of memory.

The maximum number of lines that the Memory window can display is limited to the size of the window. You can use the SCROLL DOWN and SCROLL UP commands to display additional memory.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Navigating through the Memory window using the history area" on page 170

Command pop-up window

z/OS Debugger displays the Command pop-up window as a pop-up window over the Source, Log, and Monitor windows so that you to can more easily enter long or multiline commands. z/OS Debugger displays the Command pop-up window when any of the following situations occur:

- You enter the POPUP command
- You enter an incomplete command on the command line
- You enter a continuation character on the command line
- You type over long text in the Source or Log window

You can control the size of the window by doing any of the following actions:

- When you enter the POPUP command, specify the number of lines you want for that particular instance of a Command pop-up window
- If you want the Command pop-up window to display the same number of lines every time you enter the POPUP command, specify the number of lines you want with the SET POPUP command
- Resize the window by moving the cursor below the last line in the Command pop-up window and then press Enter

After you finish entering commands, press Enter to run the commands and close the window.

List pop-up window

When the Log window is not visible, z/OS Debugger displays the results of a LIST *expression* command in the List pop-up window and writes the results to the log. If the expression evaluation fails, z/OS Debugger displays the List pop-up window with the error message. While the List pop-up window is open, you can not alter the value of a variable. You can scroll up and down in the List pop-up window by entering the SCROLL UP and SCROLL DOWN commands in the Command line or using the appropriate PF key. The maximum lines of data for the List pop-up window can not exceed 1000 lines. If the result of the expression evaluation exceeds 1000 lines, z/OS Debugger displays a warning message below the Command line. To close the List pop-up window, do either of the following:

- Press Enter.

- Enter any command except `SCROLL UP` or `SCROLL DOWN` in the Command line. z/OS Debugger closes the window and runs the command.

Creating a preferences file

If you have a preference as to the appearance or behavior of z/OS Debugger, you can set these options in a preferences file. You can modify the layout of the windows of the session panel, set PF keys to specific actions, or change the colors use in the session panel. "Saving customized settings in a preferences file" on page 256 describes what you can specify in a preferences file and how to make z/OS Debugger use your preferences file.

If your site has preferences for all users to use, the system administrator can set these preferences in a global preferences file. When z/OS Debugger starts, it does the following steps:

1. Checks for a global preferences file specified through the EQAOPTS `GPFDSN` command and runs any commands specified in that file.
2. If you specify a preferences file, z/OS Debugger looks for that preferences file and runs any commands in that preferences file. A preferences file can be specified through one of the following methods:
   - directly; for example, through the TEST runtime option
   - through the EQAOPTS `PREFERENCESDSN` command
3. If you specify a commands file, z/OS Debugger looks for that commands file and runs any commands in that commands file. A commands file can be specified through one of the following methods:
   - Directly, for example, through the TEST runtime option.
   - Through the EQAOPTS `COMMANDSDSN` command. If that file has a member in it that matches the name of the initial load module in the first enclave, z/OS Debugger reads that member as a commands file.

Because of the order in which z/OS Debugger processes these files, any settings that you specify in your preferences and commands files can override settings in the global preferences file. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or *IBM z/OS Debugger Customization Guide*. To learn about what format to use for the global preferences file, preferences file, and commands file, see Appendix A, "Data sets used by z/OS Debugger," on page 401.

Displaying the source

z/OS Debugger displays your source in the Source Window using a source, listing, or separate debug file, depending on how you prepared your program.

When you start z/OS Debugger, if your source is not displayed, see "Changing which file appears in the Source window" on page 157 for instructions on how find and display the source.

If there is no debug data, you can display the disassembled code by entering the `SET DISASSEMBLY` command.

If your programs contain Db2 or CICS code, you might need to use a different file. See Chapter 8, "Preparing a Db2 program," on page 75 or Chapter 10, "Preparing a CICS program," on page 81 for more information.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27
Chapter 6, "Preparing a LangX COBOL program," on page 67
"Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34
"Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
"Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44
Chapter 7, "Preparing an assembler program," on page 71

Chapter 8, “Preparing a Db2 program,” on page 75
Chapter 9, “Preparing a Db2 stored procedures program,” on page 79
Chapter 10, “Preparing a CICS program,” on page 81
Chapter 11, “Preparing an IMS program,” on page 95

Related references
Appendix B, “How does z/OS Debugger locate source, listing, or separate debug files?,” on page 407
IBM z/OS Debugger Reference and Messages

Changing which file appears in the Source window

This topic describes several different ways of changing which file appears in the Source window. This topic assumes you already know the name of the source, listing, or separate debug file that you want to display. If you don't know the name of the file, see “Displaying a list of compile units known to z/OS Debugger” on page 194 for suggestions on how to find the name of a file.

Before you change the file that appears in the Source window, make sure you understand how z/OS Debugger locates source, listing, and separate debug files by reading Appendix B, “How does z/OS Debugger locate source, listing, or separate debug files?,” on page 407.

To change which file appears in the Source window, choose one of the following options:

- Type over the name after SOURCE:, which is in the Header area of the Source window, with the desired name. The new name must be the name of a compile unit that is known to z/OS Debugger.
- Use the Source Identification panel to direct z/OS Debugger to the new files:
  1. With the cursor on the command line, press PF4 (LIST).

     In the Source Identification panel, you can associate the source, listing, or separate debug file that show in the Source window with their compile unit.
  2. Type over the **Listing/Source File** field with the new name.
- Use the SET SOURCE command. With the cursor on the command line, type SET SOURCE ON (*cuname*) *new_file_name*, where *new_file_name* is the new source file. Press Enter.

  If you need to do this repeatedly, you can use the SET SOURCE ON commands generated in the Log window. You can save these commands in a file and reissue them with the USE command for future invocations of z/OS Debugger.
- Enter the PANEL PROFILE command, which displays the Profile Settings panel. Enter the new file name in the Default Listing PDS name field.
- Use the SET DEFAULT LISTINGS command. With the cursor on the command line, type SET DEFAULT LISTINGS *new_file_name*, where *new_file_name* is the renamed listing or separate debug file. Press Enter.

  To point z/OS Debugger to several renamed files, you can use the SET DEFAULT LISTINGS command and specify the renamed files, separated by commas and enclosed in parenthesis. For example, to point z/OS Debugger to the files SVTRSAMP.TS99992.MYPROG, PGRSAMP.LLTEST.PROGA, and RRSAMP.CRTEST.PROGR, enter the following command:

  ```
  SET DEFAULT LISTINGS (SVTRSAMP.TS99992.MYPROG, PGRSAMP.LLTEST.PROGA,
   RRSAMP.CRTEST.PROGR) ;
  ```
- Use the EQADEBUG DD statement to define the location of the files.
- Code the EQAUEDAT user exit with the location of the files.

For C and C++ programs compiled with the FORMAT(DWARF) and FILE suboptions of the DEBUG compiler option, the information in this topic describes how to specify the location of the *source* file. If you or your site specified YES for the EQAOPTS MDBG command (which requires z/OS Debugger to search for the .dbg and the source file in a .mdbg file)[9], you cannot specify another location for the source file.

Entering commands on the session panel

You can enter a command or modify what is on the session panel in several areas, as shown in Figure 1 on page 158 and Figure 2 on page 159.

Figure 1. z/OS Debugger session panel displaying the Log window.

[9] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

```
COBOL    LOCATION: PROG1 :> 44
Command ===>     1                                          Scroll ===> CSR  2
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 2
****************************** TOP OF MONITOR *******************************
                                ----+----1----+----2----+----3----+----4----
0001  1 STR1                    'ONE  '
0002  2 STR3                    'THREE'
***************************** BOTTOM OF MONITOR *****************************
SOURCE: PROG1 - 3 -1----+----2----+----3----+----4----+----5----+ LINE: 43 OF 53
      43            MOVE "ONE" TO STR1. MOVE "TWO" TO STR2. MOVE "THREE" TO S .
      44            MOVE "FOUR" TO STR4. MOVE "FIVE" TO STR5.                 .
      45            PERFORM UNTIL R = 9                                       .
 4    46              MOVE "TOP" TO STR1 MOVE "BEG" TO STR2 MOVE "UP" TO STR3 .
      47              ADD 1 TO VARBL1                                         .
      48              SUBTRACT 2 FROM VARBL2              5                   .
      49              ADD 1 TO R                                              .
      50              MOVE "BOT" TO STR1 MOVE "END" TO STR2 MOVE "DOW" TO STR .
      51            END-PERFORM.                                              .
      52            MOVE "DONE" TO STR1. MOVE "END" TO STR2. MOVE "FIN" TO ST .
      53            STOP RUN.                                                 .
***************************** BOTTOM OF SOURCE ******************************
MEMOR 6 -+----2----+----3----+----4----+----5----+----6----+----7----+----8----+
History: 329D47DA         329D65CC         329D88AB         329D8000
         329D90E8                  8
Base address: 329D90E8 Amode: 31
+00000 329D90E8   D6D5C540 40000000 E3E6D640 40000000    | ONE  ...TWO  ... |
+00010 329D90F8   E3C8D9C5 C5000000 00000000 00000000    | THREE........... |
+00020 329D9108   00000000 00000000 00000000 00000000    | ................ |
+00030 329D9118   00000000 00000000 00000000 00000000    | ................ |
+00040 329D9128   00000000 00000000 00000000 00000000    | ................ |
+00050 329D9138   00000000 00000000 00000000 00000000    | ................ |
+00060 329D9148   00000000 00000000 00000000 00000000    | ................ |
+00070 329D9158   00000000 00000000 00000000 00000000    | ................ |
PF  1:ZOOM MEM   2:STEP       3:QUIT       4:SWAP       5:MEMORY     6:BREAK
PF  7:UP         8:DOWN       9:GO        10:ZOOM SRC  11:ZOOM LOG  12:RETRIEVE
```

Figure 2. z/OS Debugger session panel displaying the Memory window.

Note: Figure 2 on page 159 shows PF keys that were redefined. If you want to redefine your PF keys, see "Defining PF keys" on page 251.

1 Command line
You can enter any valid z/OS Debugger command on the command line.

2 Scroll area
You can redefine the default amount you want to scroll by typing the desired value over the value currently displayed.

3 Compile unit name area
You can change the qualification by typing the desired qualification over the value currently displayed. For example, to change the current qualification from `ICFSSCU1`, as shown in the Source window header, to `ICFSSCU2`, type `ICFSSCU2` over `ICFSSCU1` and press Enter.

4 Prefix area
You can enter only z/OS Debugger prefix commands in the prefix area, located in the left margin of the Source window.

5 Source window
You can modify any lines in the Source window and place them on the command line.

6 Window id area
You can change your window configuration by typing the name of the window you want to display over the name of the window that is currently being displayed.

7 Log window
You can modify any lines in the log and have z/OS Debugger place them on the command line.

8 Memory window
You can modify memory or specify a new memory base address. This window is not displayed by default. You must enter the `WINDOW SWAP MEMORY LOG` command, `WINDOW OPEN MEMORY` command, or `WINDOW ZOOM MEMORY` command to display this window.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using the session panel command line" on page 160
"Issuing system commands" on page 160
"Entering prefix commands on specific lines or statements" on page 161
"Entering multiple commands in the Memory window" on page 162
"Using commands that are sensitive to the cursor position" on page 162
"Using Program Function (PF) keys to enter commands" on page 162
"Retrieving previous commands" on page 163
"Composing commands from lines in the Log and Source windows" on page 164

Related references

"Order in which z/OS Debugger accepts commands from the session panel" on page 160
"Initial PF key settings" on page 163

Order in which z/OS Debugger accepts commands from the session panel

If you enter commands in more than one valid input area on the session panel and press Enter, the input areas are processed in the following order of precedence.

1. Prefix area
2. Command line
3. Compile unit name area
4. Scroll area
5. Window id area
6. Source/Log window
7. Memory window

Using the session panel command line

You can enter any z/OS Debugger command in the command field. You can also enter any TSO command by prefixing them with SYSTEM or TS0. Commands can be up to 48 SBCS characters or 23 DBCS characters in length.

If you need to enter a lengthy command, z/OS Debugger provides a command continuation character, the SBCS hyphen (-). When the current programming language is C and C++, you can also use the backslash (****) as a continuation character. You can continue requesting additional command lines by entering the continuation characters until you complete your command.

z/OS Debugger also provides automatic continuation if your command is not complete; for example, if you enter a left brace (**{**) without the matching right brace (**}**). If you need to continue your command, z/OS Debugger displays the Command pop-up window. You type in the rest of your command and any other commands. Press Enter to run the commands and close the Command pop-up window.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 29, "Entering z/OS Debugger commands," on page 261

Issuing system commands

During your z/OS Debugger session, you can still access your base operating system using the SYSTEM command. The string following the SYSTEM command is passed on to your operating system. You can communicate with TSO in a TSO environment. For example, if you want to see a TSO catalog listing while in a debugging session, enter `SYSTEM LISTC;`.

When you are entering system commands, you must comply with the following:

- A command is required after the SYSTEM keyword. Do not enter any required parameters. z/OS Debugger prompts you.
- If you are debugging in batch and need system services, you can include commands and their requisite parameters in a CLIST and substitute the CLIST name in place of the command.
- If you want to enter several TSO commands, you can include them in a USE file, a procedure, or other commands list. Or you can enter:

  ```
  SYSTEM ISPF;
  ```

  This starts ISPF and displays an ISPF panel on your host emulator screen that you can use to issue commands.

For CICS only: The SYSTEM command is not supported.

TS0 is a synonym for the SYSTEM command. Truncation of the TS0 command is not allowed.

Entering prefix commands on specific lines or statements

You can type certain commands, known as *prefix commands*, in the prefix area of specific lines in the Source or Monitor window so that those commands affect only those lines. For example, you can type the AT command in the prefix area of line 8 in the Source window, press Enter, then z/OS Debugger sets a statement breakpoint only on line 8.

The following prefix commands can be entered in the prefix area of the Source window:

- AT
- CLEAR
- DISABLE
- ENABLE
- L
- M
- QUERY
- RUNTO
- SHOW

The following prefix commands can be entered in the prefix area of the Monitor window, including the automonitor section:

- HEX
- DEF
- CL
- LIST
- CC...code coverage(to clear a range of lines)

To enter a prefix command into the Source window, do the following steps:

1. Scroll through the Source window until you see the line or lines of code you want to change.
2. Move your cursor to the prefix area of the line you want to change.
3. Type in the appropriate prefix command.
4. If there are multiple statements or verbs on the line, you can indicate which statement or verb you want to change by typing in a number indicating the relative position of the statement or verb. For example, if there are three statements on the line and you want to set a breakpoint on the third statement, type in a 3 following the AT prefix command. The resulting prefix command is AT 3.
5. If there are more lines you want to change, return to step 3.
6. Press Enter. z/OS Debugger runs the commands you typed on the lines you typed them on.

To enter a prefix command into the Monitor window, do the following steps:

1. Scroll through the Monitor window until you see the line or lines you want to change.
2. Move your cursor to the prefix area of the line you want to change.
3. Type in the appropriate prefix command.
4. If there are more lines you want to change, return to step 3.
5. Press Enter. z/OS Debugger runs the commands you typed on the lines you typed them on.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
SET MONITOR command in *IBM z/OS Debugger Reference and Messages*
Prefix commands in *IBM z/OS Debugger Reference and Messages*

Entering multiple commands in the Memory window

You can enter multiple commands and changes into the Memory window. z/OS Debugger processes the user input line by line, starting at the top of the Memory window, as described in the following list:

1. History entry area. Processing stops at an invalid input, which displays an error message, or after the first "G" or "R" command. The Memory window is refreshed and the remaining commands and changes you typed into the Memory window are ignored.
2. Base address. Processing stops at an invalid input, which displays an error message; after valid input; or after the first "G" command. The Memory window is refreshed and the remaining commands and changes you typed into the Memory window are ignored.
3. Address column. Processing stops at an invalid input, which displays an error message; after valid input; or after the first "G" command. The Memory window is refreshed and the remaining commands and changes you typed into the Memory window are ignored.
4. Hexadecimal data area. Processing stops at an invalid input, which displays an error message; after valid input; or after the first "G" command. Valid changes that z/OS Debugger encounters before invalid changes or the "G" command are processed. The Memory window is refreshed and the remaining commands or changes you typed into the Memory window are ignored.

Using commands that are sensitive to the cursor position

Certain commands are sensitive to the position of the cursor. These commands, called *cursor-sensitive* commands, include all those that contain the keyword `CURSOR` (`AT CURSOR`, `DESCRIBE CURSOR`, `FIND CURSOR`, `LIST CURSOR`, `SCROLL...CURSOR`, `TRIGGER AT CURSOR`, `WINDOW...CURSOR`).

To enter a cursor-sensitive command, type it on the command line, position the cursor at the location in your Source window where you want the command to take effect (for example, at the beginning of a statement or at a verb), and press Enter.

You can also issue cursor-sensitive commands by assigning them to PF keys.

Note: Do not confuse cursor-sensitive commands with the `CURSOR` command, which returns the cursor to its last saved position.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Defining PF keys” on page 251

Using Program Function (PF) keys to enter commands

The cursor-sensitive commands, as well as other full-screen tasks, can be issued more quickly by assigning the commands to PF keys. You can issue the `WINDOW CLOSE`, `LIST`, `CURSOR`, `SCROLL TO`, `DESCRIBE ATTRIBUTES`, `RETRIEVE`, `FIND`, `WINDOW SIZE`, and the scrolling commands (`SCROLL UP`, `DOWN`, `LEFT`, and `RIGHT`) this way. Using PF keys makes tasks convenient and easy.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Defining PF keys" on page 251
"Using commands that are sensitive to the cursor position" on page 162

Related references

"Initial PF key settings" on page 163

Initial PF key settings

The table below shows the initial PF key settings.

PF key	Label	Definition	Use
PF1	?	?	"Getting online help for z/OS Debugger command syntax" on page 265
PF2	STEP	STEP	"Stepping through or running your program" on page 175
PF3	QUIT	QUIT	"Ending a full-screen debug session" on page 195
PF4	LIST	LIST	"Displaying a list of compile units known to z/OS Debugger" on page 194
PF4	LIST	LIST *variable_name*	"Displaying and monitoring the value of a variable" on page 182
PF5	FIND	IMMEDIATE FIND	"Finding a string in a window" on page 167
PF6	AT/CLEAR	AT TOGGLE CURSOR	"Setting breakpoints to halt your program at a line" on page 174
PF7	UP	IMMEDIATE UP	"Scrolling through the physical windows" on page 165
PF8	DOWN	IMMEDIATE DOWN	"Scrolling through the physical windows" on page 165
PF9	GO	GO	"Stepping through or running your program" on page 175
PF10	ZOOM	IMMEDIATE ZOOM	"Zooming a window to occupy the whole screen" on page 253
PF11	ZOOM LOG	IMMEDIATE ZOOM LOG	"Zooming a window to occupy the whole screen" on page 253
PF12	RETRIEVE	IMMEDIATE RETRIEVE	"Retrieving previous commands" on page 163

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Defining PF keys" on page 251

Retrieving previous commands

To retrieve the last command you entered, press PF12 (RETRIEVE). The retrieved command is displayed on the command line. You can make changes to the command, then press Enter to issue it.

To step backwards through previous commands, press PF12 to retrieve each command in sequence. If a retrieved command is too long to fit in the command line, only its last line is displayed.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Composing commands from lines in the Log and Source windows" on page 164

Composing commands from lines in the Log and Source windows

You can use lines in the Log and Source windows to compose new commands.

To compose a command from lines in the Log or Source window, do the following steps:

1. Move the cursor to the desired line.
2. Modify one or more lines that you want to include in the command. For example, delete any comment characters.
3. Press Enter. z/OS Debugger displays the input line or lines on the command line. If the line or lines do not fit on the command line, z/OS Debugger displays the Command pop-up window with the command as typed in so far. Any trailing blanks on the last line are removed. If you want to expand the Command pop-up window, place the cursor below it and press Enter.
4. If the command is incomplete, modify the command.
5. Press Enter to run the command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Retrieving previous commands" on page 163
Chapter 29, "Entering z/OS Debugger commands," on page 261
Related references
"COBOL command format" on page 267
"z/OS Debugger subset of PL/I commands" on page 283
"PL/I language statements" on page 283
"z/OS Debugger commands that resemble C and C++ commands" on page 293

Opening the Command pop-up window to enter long z/OS Debugger commands

If you need to enter a command that is longer than the length of the command line, enter the POPUP command to open the Command pop-up window and then enter your z/OS Debugger command.

z/OS Debugger automatically displays the Command pop-up window in the following situations:

- You enter an incomplete command on the command line.
- You enter a continuation character on the command line.

You can enter the rest of your command in the Command pop-up window.

Navigating through z/OS Debugger windows

You can navigate in any of the windows using the CURSOR command and the scrolling commands: SCROLL UP, DOWN, LEFT, RIGHT, TO, NEXT, TOP, and BOTTOM. You can also search for character strings using the FIND command, which scrolls you automatically to the specified string.

The window acted upon by any of these commands is determined by one of several factors. If you specify a window name (LOG, MEMORY, MONITOR, or SOURCE) when entering the command, that window is acted upon. If the command is cursor-oriented, the window containing the cursor is acted upon. If you do not specify a window name and the cursor is not in any of the windows, the window acted upon is determined by the settings of **Default window** and *Default scroll amount* under the Profile Settings panel.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Moving the cursor between windows" on page 165
"Scrolling through the physical windows" on page 165
"Scrolling to a particular line number" on page 166
"Finding a string in a window" on page 167
"Changing which file appears in the Source window" on page 157
"Displaying the line at which execution halted" on page 169
"Customizing profile settings" on page 254

Moving the cursor between windows

To move the cursor back and forth quickly from the Monitor, Source, or Log window to the command line, use the CURSOR command. This command, and several other cursor-oriented commands, are highly effective when assigned to PF keys. After assigning the CURSOR command to a PF key, move the cursor by pressing that PF key. If the cursor is not on the command line when you issue the CURSOR command, it goes there. To return it to its previous position, press the CURSOR PF key again.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Defining PF keys" on page 251

Switching between the Memory window and Log window

z/OS Debugger has four logical windows, but can only display up to three physical windows at a time. You can alternate between the Memory window and the Log window by entering the `WINDOW SWAP MEMORY LOG` command on the command line. You can navigate through the physical windows by entering scroll commands.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Scrolling to a particular line number" on page 166
"Scrolling through the physical windows" on page 165

Scrolling through the physical windows

You can scroll through the physical windows by using commands or PF keys. Either way, the placement of the cursor plays a key role in determining which physical window is affected by the command.

To scroll through a physical window by using commands, do the following steps:

1. If you are going to scroll left or right through the Monitor value area of the Monitor window, enter the `SET MONITOR WRAP OFF` command.
2. Type in the scroll command in the command line, but do not press the Enter key. You can enter any of the following scroll commands: `SCROLL LEFT`, `SCROLL RIGHT`, `SCROLL UP`, `SCROLL DOWN` . You cannot scroll left or right in the Memory window.
3. Move the cursor to the physical window or area of the physical window you want to scroll through. In the Memory window, move the cursor to any section of the memory dump area. In the Monitor window, move the cursor to the Monitor value area to scroll left or right through that area. If you did not enter the `SET MONITOR WRAP OFF` command, then the scroll command will scroll the entire window.
4. Press Enter.

If you scroll a window or area to the right or left, z/OS Debugger adjusts the scale in the window or area to indicate the columns displayed in the window. If you scroll a window up or down, the line counter reflects the top line number currently displayed in that window. In the Memory window, if you scroll up or down, all the sections of the memory dump area adjust to display the new information.

You can combine steps 2 and 3 above by using the command to indicate which physical window you want to scroll through. For example, if you want to scroll up 5 lines in the physical window that is displaying the Monitor window, you enter the command `SCROLL UP 5 MONITOR`.

To scroll through a physical window using PF keys, do the following steps:

1. Move the cursor to the physical window or scrollable area you want to scroll through. A scrollable area includes the memory dump area of the Memory window.
2. Press the PF7 (UP) key to scroll up or the PF8 (DOWN) key to scroll down. The number of lines that you scroll through is determined by the value of the Default scroll amount setting.

If you do not move the cursor to a specific physical window, the default logical window is scrolled. To find out which logical window is the default logical window, enter the `QUERY DEFAULT WINDOW` command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Customizing the layout of physical windows on the session panel" on page 252
"Scrolling to a particular line number" on page 166
"Customizing profile settings" on page 254
"Enlarging a physical window" on page 166
"Navigating through the Memory window using the history area" on page 170

Related references

QUERY command in *IBM z/OS Debugger Reference and Messages*
SCROLL command in *IBM z/OS Debugger Reference and Messages*
SET DEFAULT WINDOW command in *IBM z/OS Debugger Reference and Messages*

Enlarging a physical window

You can enlarge a physical window to full screen by using the `WINDOW ZOOM` command or a PF key. To enlarge a physical window by using the `WINDOW ZOOM` command, type in `WINDOW ZOOM`, followed by the name of the physical window you want to enlarge, then press Enter. To reduce the physical window back to its original size, enter the `WINDOW ZOOM` command again. For example, if you want to enlarge the physical window that is displaying the Monitor window, enter the command `WINDOW ZOOM`. To reduce the size of that physical window back to its original size, enter the command `WINDOW ZOOM`.

To enlarge a physical window by using a PF key, move the cursor into the physical window that you want to enlarge, then press the PF10 (ZOOM) key. For example, if you want to enlarge the physical window that is displaying the Source window, move your cursor somewhere into the Source window, then press the PF10 (ZOOM) key. To reduce the size of that physical window back to its original size, press the PF10 (ZOOM) key.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Customizing the layout of physical windows on the session panel" on page 252

Related references

WINDOW command in *IBM z/OS Debugger Reference and Messages*

Scrolling to a particular line number

To display a particular line at the top of a window, use the `POSITION` or `SCROLL TO` command with the line or statement numbers shown in the window prefix areas. Enter `POSITION` *n* or `SCROLL TO` *n* (where *n* is a line number) on the command line and press Enter.

For example, to bring line 345 to the top of the window, enter `POSITION 345` OR `SCROLL TO 345` on the command line. z/OS Debugger scrolls the selected window vertically so that it displays line 345 at the top of that window.

If you used the `LIST AT LINE` or `LIST AT STATEMENT` command to get a list of line or statement breakpoints, then use the `POSITION` or `SCROLL TO` command to display one of those breakpoints at the top of the Source window. As an alternate to using the combination of the `LIST AT LINE` or `LIST AT STATEMENT` command with the `POSITION` or `SCROLL TO` command, you can use the `FINDBP`

command. The FINDBP command works in a manner similar to the FIND command for strings, except that it searches for line, statement, and offset breakpoints.

Finding a string in a window

You can search for strings in the Source, Monitor, or Log window. You can specify where to start the search, to search either forward or backward, and, for the Source window, the columns that are searched. The default window that is searched is the window specified by the SET DEFAULT WINDOW command or the *Default window* entry in your Profile Settings panel. The default direction for searches is forward. For the Source window, the default boundaries for columns are 1 to *, unless you specify a different set of boundaries with the SET FIND BOUNDS command.

To find a string within the default window using the default search direction, do the following steps:

1. Type in the FIND command, specifying the string you want to find. Ensure that the string complies with the rules described "Syntax of a search string" on page 167.
2. Press Enter.

If you want to repeat the previous search, hit the PF5 key.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts
"How does z/OS Debugger search for strings?" on page 167

Related references
"Syntax of a search string" on page 167

How does z/OS Debugger search for strings?

The z/OS Debugger FIND command uses many of the same rules for beginning a search that the ISPF FIND command uses to begin its searches. z/OS Debugger begins a search in the first position after the cursor location.

If you reach the end, z/OS Debugger displays a message indicating you have reached the end. Repeat the FIND command by pressing the PF5 key and then the search starts from the top.

If you were searching backwards and you reach the beginning, z/OS Debugger displays a message indicating you have reached the beginning. Repeat the FIND command by pressing the PF5 key and the search begins from the end.

Syntax of a search string

The string can contains any combination of characters, numbers, and symbols. However, if the string contains any of the following characters, it must be enclosed in quotation marks (") or apostrophes ('):

- spaces
- an asterisk ("*")
- a question mark ("?")
- a semicolon (";")

Use the following rules to determine whether to use quotation marks (") or apostrophes ('):

- If you are debugging a C or C++ program, the string must be enclosed in quotation marks (").
- If you are debugging an assembler, COBOL, LangX COBOL, disassembly, or PL/I program, the string can be enclosed in quotation marks (") or apostrophes (').

Finding the same string in a different window

To find the same string in a different window, type in the command: FIND * *window_name*.

Finding a string in the Monitor value area when SET MONITOR WRAP OFF is in effect

Type the FIND command with the string, then place the cursor in the Monitor window. z/OS Debugger searches the entire Monitor window, including the scrolled data in the Monitor value area, until the string is found or until the end of data is reached.

Finding the same string in a different direction

To find the same string in a different direction, enter the FIND * command with the string and the PREV or NEXT keyword. For example, the following command searches for the string "RecordDate" in the backwards direction:

```
FIND RecordDate PREV ;
```

Specifying the boundaries of a search in the Source window

You can specify that z/OS Debugger search through a limited number of columns in the Source window, which can be useful when you are searching through a very large source file and some text is organized in specific columns. You can specify the boundaries to use for the current search or for all searches. The column alignment of the source might not match the original source code. The column specifications for the FIND command are related to the scale shown in the Source window, not the original source code.

To specify the boundaries for the current search, enter the FIND command and specify the search string and the boundaries. For example, to search for "ABC" in columns 7 through 12, enter the following command:

```
FIND "ABC" 7 12;
```

To search for "VAR1" that begins in column 8 or any column after that, enter the following command:

```
FIND "VAR1" 8 *;
```

To search for "VAR1" beginning in column 1, enter the following command:

```
FIND "VAR1" 1;
```

To specify the default boundaries to use for all searches, enter the SET FIND BOUNDS command, specifying the left and right boundaries. After you enter the SET FIND BOUNDS command, every time you enter the FIND command without specifying boundaries, z/OS Debugger searches for the string you specified only within those boundaries. For example, to specify that you want z/OS Debugger to always search for text within columns 7 through 52, enter the following command:

```
SET FIND BOUNDS 7 52;
```

Afterward, every time you enter the FIND command without specifying boundaries, z/OS Debugger searches only within columns 7 through 52. To reset the boundaries to the default setting, which is 1 through *, enter the following command:

```
SET FIND BOUNDS;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related references

“Example: Searching for COBOL paragraph names” on page 169

FIND command in *IBM z/OS Debugger Reference and Messages*

SET FIND BOUNDS command in *IBM z/OS Debugger Reference and Messages*

QUERY command in *IBM z/OS Debugger Reference and Messages*

Example: Complex searches

To find a string in the backwards direction in a different window, enter the FIND command with the string, the PREV keyword, and the name of the window. For example, the following command searches for the string "EmployeeName" in the Log window:

```
FIND EmployeeName PREV LOG;
```

Example: Searching for COBOL paragraph names

To find a COBOL paragraph name that begins in column 8, enter the following command:

```
FIND paraa 8;
```

z/OS Debugger will find only the string that starts in column 8.

To find a reference to a COBOL paragraph name in COBOL's Area B within columns 12 through 72, enter the following command:

```
FIND paraa 12 72;
```

z/OS Debugger will find only the string that starts and ends within columns 12 to 72.

Displaying the line at which execution halted

After displaying different source files and scrolling, you can go back to the halted execution point by entering the `SET QUALIFY RESET` command.

Navigating through the Memory window

This topic describes the navigational aids available through the Memory window that are not available through other windows.

Displaying the Memory window

You can display the Memory window by doing one of the following options:

- Entering the `WINDOW SWAP MEMORY LOG` command. z/OS Debugger replaces the contents of the physical window that is displaying the Log window with the Memory window. The Memory window is empty if you did not specify a base address (by using the `MEMORY` command) or the history area is empty.
- After assigning the Memory window to a physical window, entering the `WINDOW OPEN MEMORY` command. z/OS Debugger opens the physical window and displays the contents of the Memory window.
- Customizing the session panel so that the Memory window is displayed in a default physical window instead of the Log window. Use this option if you want the Memory window to display continuously and in place of the Log window.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Scrolling through the physical windows" on page 165
"Switching between the Memory window and Log window" on page 165
"Displaying memory through the Memory window" on page 16
"Customizing the layout of physical windows on the session panel" on page 252

Related references

"Memory window" on page 154
"Order in which z/OS Debugger accepts commands from the session panel" on page 160
MEMORY command in *IBM z/OS Debugger Reference and Messages*

Navigating through the Memory window using the history area

Every time you enter a new MEMORY command or use the G command, the current base address is moved to the right and down in the history area. The history area can hold up to eight base addresses. When the history area is full and you enter a new base address, z/OS Debugger removes the oldest base address (located at the bottom and right-most part of the history area) from the history area and puts the new base address on the top left. The history area is persistent in a debug session.

To use the history area to navigate through the Memory window, enter the G or g command over an address in the history area, then press Enter. z/OS Debugger displays the memory dump data starting with the new address. You can clear the history area by entering the CLEAR MEMORY command. You can remove an entry in the history area by typing over the entry with the R or r command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Scrolling through the physical windows" on page 165
"Specifying a new base address" on page 170

Specifying a new base address

You can use any of the following methods to specify a new base address:

- Enter the MEMORY command on the command line
- If you defined a PF key as the MEMORY command, place the cursor in the Source window under a variable name and press that PF key.
- Type over an existing address in the Memory window in one of the following locations:
  - Information area: Type over the current base address.
  - Memory dump area: Type over an address in the address column.
- Use the G command in the Memory window in one of the following locations:
  - Information area: Enter the G command over an entry in the history area.
  - Memory dump area: Enter the G command over an address in the address column or hexadecimal data columns.

    If you enter the G command in the hexadecimal data columns, verify that the address is completely in one column and does not span across columns. For example, in the following screen, the hexadecimal addresses X'329E6470' appears in two locations:

    - In the second row, it spans the first and second column.
    - In the fifth row, it is contained in the third column.

```
MEMORY---1----+----2----+----3----+----4----+----5----+----6----+----7----+-
History:  24702630         2505A000

Base address:  265B1018  Amode: 31
+00000 265B1018   40404040 40404040 40404040 40404040  |                   |
+00010 265B1028   4040329E 64704040 40404040 40404040  |  .. ..            |
+00020 265B1038   40404040 40404040 40404040 40404040  |                   |
+00030 265B1048   40404040 40404040 40404040 40404040  |                   |
+00040 265B1058   40404040 40404040 329E6470 40404040  |          ....     |
+00050 265B1068   40404040 40404040 40404040 40404040  |                   |
+00060 265B1078   40404040 40404040 40404040 40404040  |                   |
+00070 265B1088   40404040 40404040 40404040 40404040  |                   |
```

    If you enter the G command over the second row, first column, z/OS Debugger tries to set the base address to X'4040329E'. If you enter the G command over the second row, second column, z/OS Debugger tries to set the base address to X'64704040'. If you want to set the base address to X'329E6470', do one of the following options:

    - Type the G command over the address in the fifth row, third column.
    - Enter X'329E6470' in the Base address field.

- Type in X'329E6470' in an address column, without spanning two columns, and then press Enter.

Creating a commands file

A commands file is a convenient method of reproducing debug sessions or resuming interrupted sessions. Use one of the following methods to create a commands file:

- Record your debug session in a log file and then use the log file as a commands file. This is the fastest way to create a valid commands file.
- Create a commands file manually. Appendix A, “Data sets used by z/OS Debugger,” on page 401 describes the requirements for this file and when z/OS Debugger processes it.

When you create a commands file that might be used in an application program that was created with several different programming languages, you might want to use z/OS Debugger commands that are *programming language neutral*. The following guidelines can help you write commands that are programming language neutral:

- Write conditions with the `%IF` command.
- Delimit strings and long compile unit names with quotation marks (").
- Prefix a hexadecimal constant with an X or x, followed by an apostrophe ('), then suffix the constant with an apostrophe ('). For example, you can write the hexadecimal constant C1C2C3C4 as `x'C1C2C3C4'`.
- Group commands together with the `BEGIN` and `END` commands.
- Check the *IBM z/OS Debugger Reference and Messages* to determine if a command works with only specific programming languages.
- Type in comments beginning at column 2 and not extending beyond column 72. Begin comments with "/*" and end them with "*/".

For PL/I programs, if your commands file has sequence numbers in columns 73 through 80, you must enter the `SET SEQUENCE ON` command as the first command in the commands file or before you use the commands file. After you enter this command, z/OS Debugger does not interpret the data in columns 73 through 80 as a command. Later, if you want z/OS Debugger to interpret the data in columns 73 through 80 as a command, enter the command `SET SEQUENCE OFF`.

For C and C++ programs, if you use commands that reference blocks, the block names can differ if the same program is compiled with either the ISD or DWARF compiler option. If your program is compiled with the ISD compiler option, z/OS Debugger assigns block names in a sequential manner. If your program is compiled with the DWARF compiler option, z/OS Debugger assigns block names in a non-sequential manner. Therefore, the names might differ. If you switch compiler options, check the block names in commands you use in your commands file.

At runtime, a commands file can be specified through one of the following methods:

- Directly, for example, through the TEST runtime option.
- Through the EQAOPTS `COMMANDSDSN` command. If that file has a member in it that matches the name of the initial load module in the first enclave, z/OS Debugger reads that member as a commands file.

To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or *IBM z/OS Debugger Customization Guide*. To learn about what format to use for the commands file, see Appendix A, “Data sets used by z/OS Debugger,” on page 401.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Entering comments in z/OS Debugger commands” on page 264

Related references

BEGIN command in *IBM z/OS Debugger Reference and Messages*

%IF command in *IBM z/OS Debugger Reference and Messages*

Recording your debug session in a log file

z/OS Debugger can record your commands and their generated output in a session log file. This allows you to record your session and use the file as a reference to help you analyze your session strategy. You can also use the log file as a command input file in a later session by specifying it as your primary commands file. This is a convenient method of reproducing debug sessions or resuming interrupted sessions.

The following appear as comments (preceded by an asterisk {*} in column 7 for COBOL programs, and enclosed in/* */for C, C++, PL/I and assembler programs):

- All command output
- Commands from USE files
- Commands specified on a `__ctest()` function call
- Commands specified on a `CALL CEETEST` statement
- Commands specified on a `CALL PLITEST` statement
- Commands specified in the run-time TEST command string suboption
- `QUIT` commands
- z/OS Debugger messages about the program execution (for example, intercepted console messages and exceptions)

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Creating the log file" on page 172
"Saving and restoring settings, breakpoints, and monitor specifications" on page 178

Creating the log file

For debugging sessions in full-screen mode, you can create a log file in one of the following ways:

- Automatically by using the EQAOPTS `LOGDSN` and `LOGDSNALLOC` commands. This method helps new z/OS Debugger users automatically create a log file. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or *IBM z/OS Debugger Customization Guide*.

  If you are an existing user that saves settings in a SAVESETS data set, z/OS Debugger does not create a new log file for you because the SAVESETS data set contains a `SET LOG` command. z/OS Debugger uses the log file specified in that `SET LOG` command.
- Manually as described in this topic.

For debugging sessions in batch mode, manually create the log file as described in this topic.

To create a permanent log of your debug session, first create a file with the following specifications:

- RECFM(F) or RECFM(FB) and 32<=LRECL<=256
- RECFM(V) or RECFM(VB) and 40<=LRECL<=264

Then, allocate the file to the DD name INSPLOG in the CLIST, JCL, or EXEC you use to run your program.

For COBOL and LangX COBOL only, if you want to subsequently use the session log file as a commands file, make the RECFM FB and the LRECL equal to 72. z/OS Debugger ignores everything after column 72 for file input during a COBOL debug session.

For CICS only, `SET LOG OFF` is the default. To start the log, you must use the `SET LOG ON file` command. For example, to have the log written to a data set named `TSTPINE.DT.LOG` , issue: `SET LOG ON FILE TSTPINE.DT.LOG;`.

Make sure the default of `SET LOG ON` is still in effect. If you have issued `SET LOG OFF`, output to the log file is suppressed. If z/OS Debugger is never given control, the log file is not used.

When the default log file (INSPLOG) is accessed during initialization, any existing file with the same name is overwritten. On MVS, if the log file is allocated with disposition of MOD, the log output is appended to the existing file. Entering the SET LOG ON FILE xxx command also appends the log output to the existing file.

If a log file was not allocated for your session, you can allocate one with the SET LOG command by entering:

```
SET LOG ON FILE logddn;
```

This causes z/OS Debugger to write the log to the file which is allocated to the DD name LOGDDN.

Note: A sequential file is recommended for a session log since z/OS Debugger writes to the log file.

At any time during your session, you can stop information from being sent to a log file by entering:

```
SET LOG OFF;
```

To resume use of the log file, enter:

```
SET LOG ON;
```

The log file is active for the entire z/OS Debugger session.

z/OS Debugger keeps a log file in the following modes of operation: line mode, full-screen mode, and batch mode.

Recording how many times each source line runs

To record of how many times each line of your code was executed:

1. Use a log file if you want to keep a permanent record of the results. To learn how to create a log file, see “Creating the log file” on page 172.
2. Issue the command:

   ```
   SET FREQUENCY ON;
   ```

   After you have entered the SET FREQUENCY ON command, your Source window is updated to show the current frequency count. Remember that this command starts the statistic gathering to display the actual count, so if your application has already executed a section of code, the data for these executed statements will not be available.

   If you want statement counts for the entire program, issue:

   ```
   GO ;
   LIST FREQUENCY * ;
   ```

   which lists the number of times each statement is run. When you quit, the results are written to the Log file. You can issue the LIST FREQUENCY * at any time, but it will only display the frequency count for the currently active compile unit.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Creating the log file” on page 172

Recording the breakpoints encountered

If you are debugging a compile unit that does not support automonitoring, you can use the SET AUTOMONITOR command to record the breakpoints encountered in that compile unit. After you enter the SET AUTOMONITOR ON command, z/OS Debugger records the location of each breakpoint that is encountered, as if you entered the QUERY LOCATION command.

Setting breakpoints to halt your program at a line

To set or clear a line breakpoint, move the cursor over an executable line in the Source window and press PF6 (AT/CLEAR). You can temporarily turn off the breakpoint with DISABLE and turn it back on with ENABLE.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Halting on a line in C only if a condition is true" on page 225
"Halting on a line in C++ only if a condition is true" on page 236
"Halting on a COBOL line only if a condition is true" on page 201
"Halting on a PL/I line only if a condition is true" on page 217

Setting breakpoints in a load module that is not loaded or in a program that is not active

You can browse the source or set breakpoints in a load module that has not yet been loaded or in a program that is not yet active by using the following command:

```
SET QUALIFY CU load_spec ::> cu_spec ;
```

In this command, specify the name of the load module and CU in which you wish to set breakpoints. The load module is then implicitly loaded, if necessary, and a CU is created for the specified CU. The source for the specified CU is then displayed in the SOURCE window. You can then set statement breakpoints as desired.

When program execution is resumed because of a command such as GO or STEP, any implicitly loaded modules are deleted, all breakpoints in implicitly created CUs are suspended, and any implicitly created CUs are destroyed. If the CU is later created during normal program execution, the suspended breakpoints are reactivated.

If you use the SET SAVE BPS function to save and restore breakpoints, the breakpoints are saved and restored under the name of the first load module in the active enclave. Therefore, if you use the command SET QUALIFY CU to set breakpoints in programs that execute as part of different enclaves, the breakpoints that you set by using this command are not restored when run in a different enclave.

Controlling how z/OS Debugger handles warnings about invalid data in comparisons

When z/OS Debugger processes (evaluates) a conditional expression and the data in one of the operands is invalid, the conditional expression becomes invalid. In this situation, z/OS Debugger stops and prompts you for a command. You have to enter the GO command to continue running your program. If you want to prevent z/OS Debugger from prompting you in this situation, enter the SET WARNING OFF command.

A conditional expression can become invalid for several reasons, including the following situations:

- A variable is not initialized and the data in the variable is not valid for the variable's attributes.
- A field has multiple definitions, with each definition having different attributes. While the program is running, the type of data in the field changes. When z/OS Debugger evaluates the conditional expression, the data in the variable used in the comparison is not valid for the variable's attributes.

If an exception is raised during the evaluation of a conditional expression and SET WARNING is OFF, z/OS Debugger still stops, displays a message about the exception, and prompts you to enter a command.

The following example describes what happens when you use a field that has multiple definitions, with each definition having different attributes, as part of a conditional expression:

1. You enter the following command to check the value of WK-TEST-NUM, which is a field with two definitions, one is numeric, the other is string:

```
AT CHANGE WK-TEST-NUM
   BEGIN;
   IF WK-TEST-NUM = 10;
      LIST 'WK-TEST-NUM IS 10';
   ELSE;
      GO;
   END-IF;
   End;
```

2. When z/OS Debugger evaluates the conditional expression `WK-TEST-NUM = 10`, the type of data in the field `WK-TEST-NUM` is string. Because the data in the field `WK-TEST-NUM` is a string and it cannot be compared to 10, the comparison becomes invalid. z/OS Debugger stops and prompts you to enter a command.
3. You decide you want z/OS Debugger to continue running the program and stop only when the type of data in the field is numeric and matches the 10.
4. You enter the following command, which adds calls to the `SET WARNING OFF` and `SET WARNING ON` commands:

```
AT CHANGE WK-TEST-NUM
   BEGIN;
   SET WARNING OFF;
   IF WK-TEST-NUM = 10;
      LIST 'WK-TEST-NUM IS 10';
   ELSE;
      BEGIN;
      SET WARNING ON;
      GO;
      END;
   END-IF;
   SET WARNING ON;
   END;
```

   Now, when the value of the field `WK-TEST-NUM` is not 10 or it is not a numeric type, z/OS Debugger evaluates the conditional expression `WK-TEST-NUM = 10` as false and runs the `GO` command. z/OS Debugger does not stop and prompt you for a command.

In this example, the display of warning messages about the conditional expression (`WK-TEST-NUM = 10`) was suppressed by entering the `SET WARNING OFF` command before the conditional expression was evaluated. After the conditional expression was evaluated, the display of warning messages was allowed by entering the `SET WARNING ON` command.

Carefully consider when you enter the `SET WARNING OFF` command because you might suppress the display of warning messages that might help you detect other problems in your program.

Stepping through or running your program

By default, when z/OS Debugger starts, none of your program has run yet (including C++ constructors and static object initialization).

z/OS Debugger defines a line as one line on the screen, commonly identified by a line number. A statement is a language construct that represents a step in a sequence of actions or a set of declarations. A statement can equal one line, it can span several lines, or there can be several statements on one line. The number of statements that z/OS Debugger runs when you step through your program depends on where hooks are placed.

To run your program up to the next hook, press PF2 (STEP). If you compiled your program with a combination of any of the following TEST or DEBUG compiler suboptions, STEP performs one statement:

- For C, compile with `TEST(ALL)` or `DEBUG(HOOK(LINE,NOBLOCK,PATH))`.
- For C++, compile with TEST or `DEBUG(HOOK(LINE,NOBLOCK,PATH))`.
- For any release of Enterprise COBOL for z/OS, Version 3, or Enterprise COBOL for z/OS and OS/390, Version 2, compile with one of the following suboptions:

  - TEST(ALL)
  - TEST(NONE) and use the Dynamic Debug facility
- For Enterprise COBOL for z/OS, Version 4, compile with one of the following suboptions:
  - TEST(HOOK)
  - TEST(NOHOOK) and use the Dynamic Debug facility
- For any release of Enterprise PL/I for z/OS, compile with TEST(ALL).
- For Enterprise PL/I for z/OS, Version 3.4 or later, compile with TEST(ALL,NOHOOK) and use the Dynamic Debug facility.

To run your program until a breakpoint is reached, the program ends, or a condition is raised, press PF9 (GO).

Note: A condition being raised is determined by the setting of the TEST run-time suboption *test_level*.

The command STEP OVER runs the called function without stepping into it. If you accidentally step into a function when you meant to step over it, issue the STEP RETURN command that steps to the return point (just after the call point).

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 4, “Planning your debug session,” on page 25
Chapter 13, “Writing the TEST runtime option string,” on page 107

Recording and replaying statements

z/OS Debugger provides a set of commands (the PLAYBACK commands) that helps you record and replay the statements that you run while you debug your program. To record and replay statements, you need to do the following:

1. Record the statements that you run (PLAYBACK ENABLE command). If you specify the DATA parameter or the DATA parameter is defaulted, additional information about your program is recorded.
2. Prepare to replay statements (PLAYBACK START command).
3. Replay the statements that you recorded (STEP or RUNTO command).
4. Change the direction that the statements are replayed (PLAYBACK FORWARD command).
5. Stop replaying statements (PLAYBACK STOP command).
6. Stop recording the statements that you run (PLAYBACK DISABLE command). All data for the compile units specified or implied on the PLAYBACK DISABLE command is discarded.

Each of these steps are described in more detail in the sections that follow.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

Recording the statements that you run

The PLAYBACK ENABLE command includes a set of parameters to specify:

- Which compile units to record
- The maximum amount of storage to use to record the statements that you run
- Whether to record the following additional information about your program:
  - The value of variables.
  - The value of registers.
  - Information about the files you use: open, close, last operation performed on the files, how the files were opened.

The PLAYBACK ENABLE command can be used to record the statements that you run for all compile units or for specific compile units. For example, you can record the statements that you run for compile units A, B, and C, where A, B, and C are existing compile units. Later, you can enter the PLAYBACK ENABLE command and specify that you want to record the statements that you run for all compile units. You can use an asterisk (*) to specify all current and future compile units.

The number of statements that z/OS Debugger can record depends on the following:

- The amount of storage specified or defaulted.
- The number of changes made to the variables.
- The number of changes made to files.

You cannot change the storage value after you have started recording. The more storage that you specify, the more statements that z/OS Debugger can record. After z/OS Debugger has filled all the available storage, z/OS Debugger puts information about the most recent statements over the oldest information. When the DATA parameter is in effect, the available storage fills more quickly.

You can use the DATA parameter with programs compiled with the SYM suboption of the TEST compiler option only if they are compiled with the following compilers:

- Enterprise COBOL for z/OS, Version 6
- Enterprise COBOL for z/OS, Version 5
- Enterprise COBOL for z/OS, Version 4[10]
- Enterprise COBOL for z/OS and OS/390, Version 3 Release 2 or later
- Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 with APAR PQ63235
- COBOL for OS/390 & VM, Version 2 with APAR PQ63234

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Stop the recording" on page 178

Preparing to replay the statements that you recorded

The PLAYBACK START command notifies z/OS Debugger that you want to replay the statements that you recorded. This command suspends normal debugging; all breakpoints are suspended and you cannot use many z/OS Debugger commands. *IBM z/OS Debugger Reference and Messages* provides a complete list of which commands you cannot use while you replay statements.

The initial direction is backward.

Replaying the statements that you recorded

To replay the statements that you recorded, enter the STEP or RUNTO command. You can replay the statements you recorded until one of the following conditions is reached:

- If you are replaying in the backward direction, you reach the point where you entered the PLAYBACK ENABLEcommand. If you are replaying in the forward direction, you reach the point where you entered the PLAYBACK START command. command.
- You reach the point where there are no more statements to replay, because you have run out of storage.

You can replay as far forward as the point where you entered the PLAYBACK START command. As you replay statements, you see only the statements that you recorded for those compile units you indicated you wanted to record. While you are replaying steps, you cannot modify variables. If the DATA parameter is in effect, you can access the contents of variables and expressions.

[10] With Enterprise COBOL for z/OS, Version 4, and the TEST compiler option the symbol tables are always generated.

Changing the direction that statements are replayed

To change the direction that statements are replayed, enter the PLAYBACK FOWARD or PLAYBACK BACKWARD command. The initial direction is backward.

Stop the replaying

To stop replaying the statements that you recorded and resume normal debugging, enter the PLAYBACK STOP command. This command resumes normal debugging at the point where you entered the PLAYBACK START command. z/OS Debugger continues to record the statements that you run.

Stop the recording

To stop recording the statements that you run and collecting additional information about your program, enter the PLAYBACK DISABLE command. This command can be used to stop recording the statements that you run in all or specific compile units. If you stop recording for one or more compile units, the data collected for those compile units is discarded. If you stop recording for all compile units, the PLAYBACK START command is no longer available.

Restrictions on recording and replaying statements

You cannot modify the value of variables or storage while you are replaying statements.

When you replay statements, many z/OS Debugger commands are unavailable. *IBM z/OS Debugger Reference and Messages* contains a complete list of all the commands that are not available.

Restrictions on accessing COBOL data

If the DATA parameter is specified or defaulted for a COBOL compile unit that supports this parameter, you can access data defined in the following section of the DATA DIVISION:

- FILE SECTION
- WORKING-STORAGE SECTION
- LOCAL-STORAGE SECTION
- LINKAGE SECTION

You can also access special registers, except for the ADDRESS OF, LENGTH OF, and WHEN-COMPILED special registers. You can also access all the special registers supported by z/OS Debugger commands.

When you are replaying statements, many z/OS Debugger commands are available only if the following conditions are met:

- The DATA parameter must be specified or defaulted for the compile unit.
- The compile unit must be compiled with a compiler that supports the DATA parameter.

You can use the QUERY PLAYBACK command to determine the compile units for which the DATA option is in effect.

IBM z/OS Debugger Reference and Messages contains a complete list of all the commands that can be used when you specify the DATA parameter.

Saving and restoring settings, breakpoints, and monitor specifications

You can save settings, breakpoints, and monitor specifications from one debugging session and then restore them in a subsequent debugging session. You can save the following information:

Settings
The settings for the WINDOW SIZE, WINDOW CLOSE, and SET command, except for the following settings for the SET command:

- DBCS
- FREQUENCY
- NATIONAL LANGUAGE
- PROGRAMMING LANGUAGE
- FILE operand of the RESTORE SETTINGS switch
- QUALIFY
- SOURCE
- TEST

Breakpoints
All of the breakpoints currently set or suspended in the current debugging session as well as all LOADDEBUGDATA (LDD) specifications. The following breakpoints are saved:

- APPEARANCE breakpoints
- CALL breakpoints
- DELETE breakpoints
- ENTRY breakpoints
- EXIT breakpoints
- GLOBAL APPEARANCE breakpoints
- GLOBAL CALL breakpoints
- GLOBAL DELETE breakpoints
- GLOBAL ENTRY breakpoints
- GLOBAL EXIT breakpoints
- GLOBAL LABEL breakpoints
- GLOBAL LOAD breakpoints
- GLOBAL STATEMENT breakpoints
- GLOBAL LINE breakpoints
- LABEL breakpoints
- LOAD breakpoints
- OCCURRENCE breakpoints
- STATEMENT breakpoints
- LINE breakpoints
- TERMINATION breakpoints

If a deferred AT ENTRY breakpoint has not been encountered, it is not saved nor restored.

Monitor specifications
All of the monitor and LOADDEBUGDATA (LDD) specifications that are currently in effect.

In most environments, z/OS Debugger uses specific default data set names to save these items so that it can automatically save and restore these items for you. In these environments, you must automatically restore the settings so that the `SET RESTORE BPS AUTO` and `SET RESTORE MONITORS AUTO` commands are in effect during z/OS Debugger initialization. There are some environments where you have to use the RESTORE command to restore these items manually.

In TSO, CICS (when you log on with your own ID), and UNIX System Services, the following default data set names are used:

- *userid*.DBGTOOL.SAVESETS (a sequential data set) is used to save the settings.
- *userid*.DBGTOOL.SAVEBPS (a PDS or PDSE data set) is used to save the breakpoints, monitor specifications, and LDD specifications.

In non-interactive mode (MVS batch mode without using full-screen mode using the Terminal Interface Manager), you must include an INSPSAFE DD statement to indicate the data set that you want z/OS Debugger to use to save and restore the settings and an INSPBPM DD statement to indicate the data set that you want z/OS Debugger to use to save and restore the breakpoints and monitor and LDD specifications.

Use a sequential data set to save and restore the settings. Use a PDS or PDSE to save and restore the breakpoints and monitor and LDD specifications. We recommend that you use a PDSE to avoid having to compress the data set. z/OS Debugger uses a separate member to store the breakpoints, LDD data, and monitor specifications for each enclave. z/OS Debugger names the member the name of the initial load module in the enclave. If you want to discard all of the saved breakpoints, LDD data, and monitor specifications for an enclave, you can delete the corresponding member. However, do not alter the contents of the member.

Saving and restoring automatically

Saving and restoring automatically means that every time you finish a debugging session, z/OS Debugger saves information about your debugging session. The next time you start a debugging session, z/OS Debugger restores that information. Setting up automatic saving and restoring requires that you allocate files and enter the appropriate commands that enable this feature. You can do this in one of the following ways:

- You or your site can specify the EQAOPTS SAVESETDSNALLOC and SAVEBPDSNALLOC commands. These commands can create the files and enter the appropriate commands for you, your group, or your entire site. If you choose this method, you can skip the rest of this topic and follow the instructions in the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or *IBM z/OS Debugger Customization Guide*.
- Run the EQAWSVST job in *hlq*.SEQASAMP to create the data set and run the appropriate commands. The disadvantage to this method is that you have to determine if the values for the EQAOPTS SAVESETDSN and SAVEBPDSN commands have been altered, and then make a similar change to the job.
- You can do the steps described in this topic.

To enable automatic saving and restoring, you must do the following steps:

1. Pre-allocate a sequential data set with the default name where settings will be saved. If you are running in non-interactive mode (MVS batch mode without using full-screen mode using the Terminal Interface Manager), you must include an INSPSAFE DD statement that references this data set.
2. Pre-allocate a PDSE or PDS with the default name where breakpoints, monitor, and LDD specifications will be saved. If you are running in non-interactive mode (MVS batch mode without using full-screen mode using the Terminal Interface Manager), you must include an INSPBPM DD statement that references this data set.
3. Start z/OS Debugger.
   - If you are running in CICS, you must log on as a user other than the default user and the CICS region must have update authorization to the SAVE SETTINGS and SAVE BPS data sets.
   - If you are running in non-interactive mode (MVS batch mode without using full-screen mode using the Terminal Interface Manager), you must add INSPSAFE and INSPBPM DD statements that reference the data sets you allocated in step 1 and 2.
4. Enable automatic saving and restoring of settings by using the following commands:

   ```
   SET SAVE SETTINGS AUTO;
   SET RESTORE SETTINGS AUTO;
   ```

5. If you want to enable automatic saving and restoring of breakpoints and LDD specifications or monitor and LDD specifications, use the following commands:

   ```
   SET SAVE BPS AUTO;
   SET RESTORE BPS AUTO;
   ```

```
SET SAVE MONITORS AUTO;
SET RESTORE MONITORS AUTO;
```

You must do step 4 (enabling automatic saving and restoring of settings) if you want to enable automatic restoring of breakpoints or monitor specifications.

6. Shutdown z/OS Debugger. Your settings are saved in the corresponding data set.

The next time you start z/OS Debugger, the settings are automatically restored. If you are debugging the same program, the breakpoints and monitor specifications are also automatically restored.

Disabling the automatic saving and restoring of breakpoints, monitors, and settings

To disable automatic saving of breakpoints and monitors, you must ensure that the following settings are in effect:

- `SET SAVE BPS NOAUTO;`
- `SET SAVE MONITORS NOAUTO;`

To disable automatic saving of settings, you must ensure that the `SET SAVE SETTINGS NOAUTO;` setting is in effect.

To disable automatic restoring of breakpoints and monitors, you must ensure that the following settings are in effect:

- `SET RESTORE BPS NOAUTO;`
- `SET RESTORE MONITORS NOAUTO;`

To disable automatic restoring of settings, you must ensure that the `SET RESTORE SETTINGS NOAUTO;` setting is in effect.

If you disable the automatic saving of any of these values, the last saved data is still present in the appropriate data sets. Therefore, you can restore from these data sets. Be aware that this means you will restore values from the last time the data was *saved* which might not be from the last time you ran z/OS Debugger.

Restoring manually

Automatic restoring is not supported in the following environments:

- Debugging in CICS without logging-on
- Debugging Db2 stored procedures

You can save and restore breakpoints, monitor, and LDD specifications by doing the following steps:

1. Pre-allocate a sequential data set for saving and restoring of settings.
2. Pre-allocate a PDSE or PDS for saving and restoring breakpoints and monitor specifications.
3. Start z/OS Debugger.
4. To enable automatic saving of settings, use the following command where *mysetdsn* is the name of the data set that you allocated in step 1:

```
SET SAVE SETTINGS AUTO FILE mysetdsn;
```

5. To enable automatic saving of breakpoints and LDD specifications or monitor and LDD specifications, use the following commands, where *mybpdsn* is the name of the data set that you allocated in step 2:

```
SET SAVE BPS AUTO FILE mybpdsn;
SET SAVE MONITORS AUTO;
```

6. Shutdown z/OS Debugger.

The next time you start z/OS Debugger in one of these environments, you must use the following commands, in the sequence shown, at the beginning of your z/OS Debugger session.

```
SET SAVE SETTINGS AUTO FILE mysetdsn;
RESTORE SETTINGS;
SET SAVE BPS AUTO FILE mybpdsn;
RESTORE BPS MONITORS;
```

You can put these commands into a user preferences file.

Performance considerations in multi-enclave environments

Each time information is saved or restored, the following actions must take place:

1. The data set is allocated.
2. The data set is opened.
3. The data set is written or read.
4. The data set is closed.
5. The data set is deallocated.

Because each of these steps requires operating system services, the overall process can require a significant amount of elapsed time.

For saving and restoring settings, this process is done once when z/OS Debugger is activated and once when z/OS Debugger terminates. Therefore, unless z/OS Debugger is repeatedly activated and terminated, the process is not excessively time-consuming. However, for saving and restoring of breakpoints, monitors, or both, this process occurs once on entry to each enclave and once on termination of each enclave.

If your program consists of multiple enclaves or an enclave that is run repeatedly, this process might occur many times. In this case, if performance is a concern, you might want to consider disabling saving and restoring of breakpoints and monitors. If your program runs under CICS with DTCN and saving and restoring of breakpoints and monitors is not enabled (`SET SAVE BPS NOAUTO;`, `SET SAVE MONITORS NOAUTO;`, `SET RESTORE BPS NOAUTO;`, and `SET RESTORE MONITORS NOAUTO;` are in effect), breakpoints are saved and restored from a CICS Temporary Storage Queue which is less time-consuming than the standard method but does not preserve breakpoints across CICS restarts nor does it provide for saving and restoring of monitors.

Displaying and monitoring the value of a variable

z/OS Debugger can display the value of variables in the following ways:

- One-time display, by using the `LIST` command, the PF4 key, or the `L` prefix command. One-time display displays the value of the variable at the moment you enter the `LIST` command, press the PF4 key, or enter the `L` prefix command. If you step or run through your program, any changes to the value of the variable are not displayed. The `L` and `M` prefix commands are available only when you use the following languages or compilers:
  - Enterprise PL/I for z/OS, Version 3.6 or 3.7 with the PTF for APAR PK70606, or later
  - Enterprise COBOL compiled with the TEST compile option
  - Assembler
  - Disassembly
- Continuous display, called monitoring, by using the `MONITOR LIST` command, the `SET AUTOMONITOR` command, or the `M` prefix command. If you step or run through your program, any changes to the value of the variable are displayed.

Note: Use the command `SET LIST TABULAR` to format the LIST output for arrays and structures in tabular format. See the *IBM z/OS Debugger Reference and Messages* for more information about this command.

If z/OS Debugger cannot display the value of a variable in its declared data type, see “How z/OS Debugger handles characters that cannot be displayed in their declared data type” on page 189.

One-time display of the value of variables

Before you begin, determine if you want to change the format in which information is displayed. Variables that are areas and structures might be easier to read if they are arranged in a tabular format on the screen. To make changes to the format, do one of the following options:

- If you want to change the format of the output for arrays and structures to tabular format when displaying a variable, do the following steps:
  1. Move the cursor to the command line.
  2. Enter the following command: `SET LIST TABULAR ON`
- If you want to change the format of the output for arrays and structures to linear format when displaying a variable, do the following steps:
  1. Move the cursor to the command line.
  2. Enter the following command: `SET LIST TABULAR OFF`
- If you want to format the logged output of arrays and structures when `SET AUTOMONITOR ON LOG` is in effect, do the following steps:
  1. Move the cursor to the command line.
  2. Enter the following command: `SET LIST TABULAR ON`
  3. Enter the following command: `SET AUTOMONITOR ON LOG`

To display the contents of a variable once, do one of the following options:

- By using the PF4 key, do the following steps:
  1. Scroll through the Source window until you find the variable you want to display.
  2. Move your cursor to the variable name.
  3. Press the PF4 (LIST) key. The value of the variable is displayed in the Log window.
- By using the `LIST` command:
  1. Move the cursor to the command line.
  2. Type the following command, substituting your variable name for *variable-name*:

     ```
     LIST variable-name;
     ```

  3. Press Enter. The value of the variable is displayed in the Log window.
- By using the L prefix command, do the following steps:
  1. Scroll through the Source window until you find the operand you want to display.
  2. Move your cursor to the prefix area of the line that contains the operand you want to display.
  3. Type in an "L" in the prefix area, then press Enter to display the value of all of the operands on that line. If you want to display the value of a specific operand on that line, do the following steps:
     a. If you are debugging a high-level language program, beginning from the left and with the number 1, assign a number to the first occurrence of each variable. For example, in the following line, *rightSide* is 1, *leftSide* is 2, and *bottomSide* is 3:

        ```
        rightSide = (leftSide * leftSide) + (bottomSide * bottomSide);
        ```

        If you are debugging an assembler or disassembly program, beginning from the left and beginning with number 1 assign the each operand of the machine instruction a number.

     b. Type in an "L" in the prefix area, followed by the number assigned to the operand that you want to display. If you wanted to display the value of *leftSide* in the previous example, you would enter "L2" in the prefix area.

c. Press Enter. z/OS Debugger displays the value of *leftSide* in the Log window.

Adding variables to the Monitor window

When you add a variable to the Monitor window, you are *monitoring* the value of that variable. To add a variable to the Monitor window, do one of the following options:

- To use the `MONITOR LIST` command, do the following steps:
  1. Move the cursor to the command line.
  2. Type the following command, substituting your variable name for *variable-name*:

     ```
     MONITOR LIST variable-name;
     ```

  3. Press Enter. z/OS Debugger assigns the variable a reference number between 1 and 99, adds the variable to the Monitor window (above the automonitor section, if it is displayed), and displays the current value of the variable.

  Every time z/OS Debugger receives control or every time you enter a z/OS Debugger command that can affect the display, z/OS Debugger updates the value of *variable-name* in the Monitor window so that the Monitor window always displays the current value.
- To use the `M` prefix command, do the following steps:
  1. Scroll through the Source window until you find the operand you want to monitor.
  2. Move your cursor to the prefix area of the line that contains the operand you want to monitor.
  3. Type in an "M" in the prefix area, then press Enter to monitor the value of all of the operands on that line. If you want to monitor the value of a specific operand on that line, do the following steps:
     a. If you are debugging a high-level language program, beginning from the left and with number 1, assign a number to the first occurrence of each variable. For example, in the following line, *rightSide* is 1, *leftSide* is 2, and *bottomSide* is 3:

        ```
        rightSide = (leftSide * leftSide) + (bottomSide * bottomSide);
        ```

        If you are debugging an assembler or disassembly program, beginning from the left and beginning with number 1 assign the each operand of the machine instruction a number.
     b. Type in an "M" in the prefix area, followed by the number assigned to the operand that you want to monitor. If you wanted to monitor the value of *leftSide* in the previous example, you would enter "M2" in the prefix area.
     c. Press Enter.

  Every time z/OS Debugger receives control or every time you enter a z/OS Debugger command that can affect the display, z/OS Debugger updates the value of *leftSide* in the Monitor window so that the Monitor window always displays the current value.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Adding variables to the Monitor window automatically” on page 186

Displaying the Working-Storage Section of a COBOL program in the Monitor window

You can add all of the variables in the Working-Storage Section of a COBOL program to the Monitor window by doing the following steps:

1. Move the cursor to the command line.
2. Type in the following command: `MONITOR LIST TITLED WSS;`

3. Press Enter. z/OS Debugger assigns the WSS entry a reference number between 1 and 99, adds the WSS entry to the Monitor window, and displays the current values of all of the variables in the Working-Storage Section.

Every time z/OS Debugger receives control or you enter a z/OS Debugger command that can effect the display, z/OS Debugger updates the value of each variable in the Monitor window so that z/OS Debugger always displays the current value.

Because the Working-Storage Section can contain many variables, monitoring the Working-Storage Section can add a substantial amount of overhead and use more storage.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Modifying variables or storage by typing over an existing value" on page 191

Displaying the data type of a variable in the Monitor window

The command `SET MONITOR DATATYPE ON` displays the data type of the variables displayed in the Monitor window, including those in the automonitor section. The data type is ordinarily the type which was used in the declaration of the variable. The command `SET MONITOR DATATYPE OFF` disables the display of this information.

To display the value and data type of a variable in the Monitor window:

1. Move the cursor to the command line.
2. Enter the following command:

```
SET MONITOR DATATYPE ON;
```

3. Enter one of the following commands:

   - `MONITOR LIST` *`variable-name`*`;`

     Substitute the name of your variable name for *variable-name*. z/OS Debugger adds the variable to the Monitor window and displays the current value and data type of the variable.

   - `SET AUTOMONITOR ON;`

     z/OS Debugger adds the variable or variables in the current statement to the automonitor section of the Monitor window and displays the current value and data type of the variable or variables.

   - `SET AUTOMONITOR ON LOG;`

     z/OS Debugger adds the variable or variables to the automonitor section of the Monitor window, displays the current value and data type of the variable or variables, and saves that information in the log.

Replacing a variable in the Monitor window with another variable

When you add a variable to the Monitor window, z/OS Debugger assigns the variable a reference number between 1 and 99. You can use the reference numbers to help you replace a variable in the Monitor window with another variable.

To replace a variable in the Monitor window with another variable, do the following steps:

1. Verify that you know the reference number of the variable in the Monitor window that you want to replace.
2. Move the cursor to the command line.

3. Type the following command, substituting *reference_number* with the reference number of the variable you want to replace and *variable-name* with the name of a new variable:

   ```
   MONITOR reference_number LIST variable-name;
   ```

   You can specify only an existing reference number or a reference number that is one greater than the highest existing reference number.
4. Press Enter. z/OS Debugger adds the new variable to the Monitor window on the line that displayed the old variable, and displays the current value of that variable.

If you added an element of an array to the Monitor window, you can replace that element with another element of the same array by doing the following steps:

1. Move your cursor to the Monitor window and place it under the subscript you want to change.
2. Type in the new subscript.
3. Press Enter. z/OS Debugger replaces the old element with the new element, then displays a message confirming the change.

Adding variables to the Monitor window automatically

As you step through a program, you might want to monitor variables that are on each statement as you run each statement. Manually adding variables to the Monitor window (as described in "Adding variables to the Monitor window" on page 184) before you run each statement can be time consuming. z/OS Debugger can automatically add the variables at each statement, before or after it is run; display the values of those variables, before or after the statement is run; then remove the variables from the Monitor window after you run the statement. To do this, use the `SET AUTOMONITOR ON` command.

Before you begin, make sure you understand how the `SET AUTOMONITOR` command works by reading "How z/OS Debugger automatically adds variables to the Monitor window" on page 187.

To add variables to the Monitor window automatically, do the following steps:

1. Move the cursor to the command line.
2. Enter one of the following commands:
   - `SET AUTOMONITOR ON;` if you want to display variables at the current statement, before the statement is run.
   - `SET AUTOMONITOR ON PREVIOUS;` if you want to display variables at the statement z/OS Debugger just ran, after the statement was run.
   - `SET AUTOMONITOR ON BOTH;` if you want to display variables at the statement z/OS Debugger just ran, after the statement was run, and the current statement, before the statement is run.

   As you step through your program, z/OS Debugger displays the names and values of the variables in the automonitor section of the window.
3. To stop adding variables to the Monitor window automatically, enter the `SET AUTOMONITOR OFF` command. z/OS Debugger removes the line `********** AUTOMONITOR **********` and any variables underneath that line.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts
"How z/OS Debugger automatically adds variables to the Monitor window" on page 187

Related tasks
"Saving the information in the automonitor section to the log file" on page 187

Related references
Description of the `SET AUTOMONITOR` command in *IBM z/OS Debugger Reference and Messages*.
"Example: How z/OS Debugger adds variables to the Monitor window automatically" on page 188

Saving the information in the automonitor section to the log file

To save the following information in the log file, enter the SET AUTOMONITOR ON LOG command:

- Breakpoint locations
- The names and values of the variables at the breakpoints

The default option is NOLOG, which would not save the above information.

Each entry in the log file contains the breakpoint location within the program and the names and values of the variables in the statement. To stop saving this information in the log file and continue updating the automonitor section of the Monitor window, enter the SET AUTOMONITOR ON NOLOG command.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts
"How z/OS Debugger automatically adds variables to the Monitor window" on page 187

Related tasks
"Adding variables to the Monitor window automatically" on page 186

Related references
Description of the SET AUTOMONITOR command in *IBM z/OS Debugger Reference and Messages*.
"Example: How z/OS Debugger adds variables to the Monitor window automatically" on page 188

How z/OS Debugger automatically adds variables to the Monitor window

When you enter the SET AUTOMONITOR ON command, z/OS Debugger displays the line ********** AUTOMONITOR ********** at the bottom of the list of any monitored variables in the Monitor window, as shown in the following example:

```
COBOL    LOCATION: DTAM01 :> 109.1
Command ===>                                                Scroll ===> PAGE
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 7
******************************** TOP OF MONITOR ********************************
                                  ----+----1----+----2----+----3----+----4----+

0001  1 NUM1                       0000000005
0002  2 NUM4                      '1111'
0003  3 WK-LONG-FIELD-2           '123456790 223456790 323456790 423456790 523
0004                               456790 623456790 723456790 823456790 9234567
0005                               90 023456790 123456790 223456790 323456790 4
0006                               23456790 523456790 623456790 723456790 82345
0007                               ********** AUTOMONITOR **********
```

The area below this line is called the automonitor section. Each time you enter the STEP command or a breakpoint is encountered, z/OS Debugger does the following tasks:

1. Removes any variable names and values displayed in the automonitor section.
2. Displays the names and values of the variables of the statement that z/OS Debugger runs next. The values displayed are values *before* the statement is run.

This behavior displays the value of the variables before z/OS Debugger runs the statement. If you want to see the value of the variables after z/OS Debugger runs the statement, you can enter the SET AUTOMONITOR ON PREVIOUS command. z/OS Debugger displays the line ********** AUTOMONITOR – PREVIOUS *load-name* ::> *cu-name* :> *statement-id* ********** at the bottom of the list of any monitored variables in the Monitor window. Each time you enter the STEP command or a breakpoint is encountered, z/OS Debugger does the following tasks:

1. Removes any variable names and values displayed in the automonitor section.
2. Displays the names and the values of the variables of the most recent statement that z/OS Debugger ran. The values displayed are values *after* that statement was run.

If you want to see the value of the variables before *and* after z/OS Debugger runs the statement, you can enter the SET AUTOMONITOR ON BOTH command. z/OS Debugger displays the line ********** AUTOMONITOR *load-name* ::> *cu-name* :> *statement-id* ********** at the bottom of the list of any monitored variables in the Monitor window. Below this line, z/OS Debugger displays the names and values of the variables on the statement that z/OS Debugger runs next. Then, z/OS Debugger displays the line ***** Previous Statement *load-name* ::> *cu-name* :> *statement-id* ***** . Below this line, z/OS Debugger displays the names and values of the variables of the statement that z/OS Debugger just ran. Each time you enter the STEP command or a breakpoint is encountered, z/OS Debugger does the following tasks:

1. Removes any variable names and values displayed in the automonitor section.
2. Displays the names and values of the variables of the statement that z/OS Debugger runs next. The values displayed are values *before* the statement is run.
3. Displays the names and the values of the variables of the statement that z/OS Debugger just ran. The values displayed are values *after* the statement was run.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Adding variables to the Monitor window automatically" on page 186

Related references

Description of the SET AUTOMONITOR command in *IBM z/OS Debugger Reference and Messages*.

"Example: How z/OS Debugger adds variables to the Monitor window automatically" on page 188

Example: How z/OS Debugger adds variables to the Monitor window automatically

The example in this section assumes that the following two lines of COBOL code are to be run:

```
COMPUTE LOAN-AMOUNT = FUNCTION NUMVAL(LOAN-AMOUNT-IN).  1
COMPUTE INTEREST-RATE = FUNCTION NUMVAL(INTEREST-RATE-IN).
```

Before you run the statement in Line 1, enter the following command:

```
SET AUTOMONITOR ON ;
```

The name and value of the variables LOAN-AMOUNT and LOAN-AMOUNT-IN are displayed in the automonitor section of the Monitor window. These values are the values of the variables before you run the statement.

Enter the STEP command. z/OS Debugger removes LOAN-AMOUNT and LOAN-AMOUNT-IN from the automonitor section of the Monitor window and then displays the name and value of the variables INTEREST-RATE and INTEREST-RATE-IN. These values are the values of the variables before you run the statement.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts

"How z/OS Debugger automatically adds variables to the Monitor window" on page 187

Related tasks

"Adding variables to the Monitor window automatically" on page 186

Related references

Description of the SET AUTOMONITOR command in *IBM z/OS Debugger Reference and Messages*.

How z/OS Debugger handles characters that cannot be displayed in their declared data type

In the Monitor window, z/OS Debugger uses one of the following characters to indicate that a character cannot be displayed in its declared data type:

- For COBOL and PL/I programs, z/OS Debugger displays a dot (X'4B').
- For assembler and LangX COBOL programs, z/OS Debugger displays a quotation mark (").
- For C and C++ programs, z/OS Debugger displays the character as an escape sequence.

Characters that cannot be displayed in their declared data type can vary from code page to code page, but, in general, these are characters that have no corresponding symbol that can be displayed on a screen.

To be able to modify these characters, you can use the HEX and DEF prefix commands to help you verify which character you are modifying.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Modifying characters that cannot be displayed in their declared data type" on page 189

Modifying characters that cannot be displayed in their declared data type

As described in "How z/OS Debugger handles characters that cannot be displayed in their declared data type" on page 189, if you want to modify characters that can't be displayed in their declared data type and ensure that the results are what you expected, do the following steps:

1. Move the cursor to the prefix area of the Monitor window, along the line that contains the character you want to modify.
2. Enter the HEX prefix command. z/OS Debugger changes the character to display in hexadecimal format.
3. Move the cursor to the character.
4. Type in the new *hexadecimal* value and then press Enter. z/OS Debugger modifies the character and displays the new value in hexadecimal format.
5. If you want to view the character in its declared data type, move the cursor to the prefix area and enter the DEF command.

Refer to the following topics for more information related to the material discussed in this topic.

"Displaying and monitoring the value of a variable" on page 182
"Modifying the value of a COBOL variable" on page 200
"Displaying and modifying the value of LangX COBOL variables or storage" on page 210
"Modifying the value of a PL/I variable" on page 216
"Modifying the value of a C variable" on page 224
"Modifying the value of a C++ variable" on page 235
"Displaying and modifying the value of assembler variables or storage" on page 248

Related references

Prefix commands in *IBM z/OS Debugger Reference and Messages*

Formatting values in the Monitor window

To monitor the value of the variable in columnar format, enter the `SET MONITOR COLUMN ON` command. The variable names that are displayed in the Monitor window are aligned to the same column and values are aligned to the same column. z/OS Debugger displays the Monitor value area scale under the header line for the Monitor window.

To display the value of the monitored variables wrapped in the Monitor window, enter the `SET MONITOR WRAP ON` command. To display the value of the monitored variables in a scrollable line, enter the `SET MONITOR WRAP OFF` command after you enter the `SET MONITOR COLUMN ON` command.

Displaying values in hexadecimal format

You can display the value of a variable in hexadecimal format by entering the `LIST %HEX` command or defining a PF key with the `LIST %HEX` command. For PL/I programs, to display the value of a variable in hexadecimal format, use the PL/I built-in function HEX. For more information about the PL/I HEX built-in function, see *Enterprise PL/I for z/OS: Programming Guide*. If you display a PL/I variable in hexadecimal format, you cannot edit the value of the variable by typing over the existing value in the Monitor window.

To display the value of a variable in hexadecimal format, enter one of the following commands, substituting *variable-name* with the name of your variable:

- For PL/I programs: `LIST HEX(`*`variable-name`*`) ;`
- For all other programs: `LIST %HEX(`*`variable-name`*`) ;`

z/OS Debugger displays the value of the variable *variable-name* in hexadecimal format.

If you defined a PF key with the `LIST %HEX` command, do the following steps:

1. If the variable is not displayed in the Source window, scroll through your program until the variable you want is displayed in the Source window.
2. Move your cursor to the variable name.
3. Press the PF key to which you defined `LIST %HEX` command. z/OS Debugger displays the value of the variable *variable-name* in hexadecimal format.

You cannot define a PF key with the PL/I HEX built-in function.

Monitoring the value of variables in hexadecimal format

You can monitor the value of a variable in either the variable's declared data type or in hexadecimal format. To monitor the value of a variable in its declared data type, follow the instructions described in “Adding variables to the Monitor window” on page 184. If you monitor a PL/I variable in hexadecimal format by using the PL/I HEX built-in function, you cannot edit the value of the variable by typing over the existing value in the Monitor window. Instead of using the PL/I HEX built-in function, us the commands described in this topic.

To monitor the value of a variable or expression in hexadecimal format, do one of the following instructions:

- If the variable is already being monitored, enter the following command:

  ```
  MONITOR n HEX ;
  ```

  Substitute *n* with the number in the monitor list that corresponds to the monitored expression that you would like to display in hexadecimal format.
- If the variable is not being monitored, enter the following command:

  ```
  MONITOR LIST (expression) HEX ;
  ```

  Substitute *expression* with the name of the variable or a complex expression that you want to monitor.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Entering prefix commands on specific lines or statements” on page 161

Modifying variables or storage by using a command

You can modify the value of a variable or storage by using one of the following commands:

- assignment command for assembler or disassembly
- assignment command for LangX COBOL
- assignment command for PL/I
- COMPUTE command for COBOL
- Expression command for C and C++
- MOVE command for COBOL
- SET command for COBOL
- STORAGE

Each command is described in *IBM z/OS Debugger Reference and Messages*.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Displaying values of COBOL variables” on page 270
“Displaying values of C and C++ variables or expressions” on page 294
“Accessing PL/I program variables” on page 287
“Displaying and modifying the value of assembler variables or storage” on page 248

Modifying variables or storage by typing over an existing value

To modify the value of a variable by typing over the existing value in the Monitor window, do the following steps:

1. Move the cursor to the existing value. If the part of value you that want to modify is out of screen, use the SCROLL Monitor value area function (available with the `SET MONITOR WRAP OFF` command) and move the cursor to the position of existing value.
2. Type in the new value. Black vertical bars mark the area where you can type in your new value; you cannot type anything before and including the left vertical bar nor can you type anything including and after the right vertical bar.
3. Press Enter.

   z/OS Debugger modifies the variable or storage. The command that z/OS Debugger generated to modify the variable or storage is stored in the log file.

Restrictions for modifying variables in the Monitor window

You can modify the value of a variable by typing over the existing value in the Monitor window, with the following exceptions:

- You cannot type in a value that is larger than the declared type of the variable. For example, if you declare a variable as a string of four character and you try to type in five characters, z/OS Debugger prevents you from typing in the fifth character.
- If z/OS Debugger cannot display the entire value in the Monitor window and the setting of `MONITOR WRAP` is `ON`, you cannot modify the value of that variable.
- If you modify a long value and the setting of `MONITOR WRAP` is `OFF`, z/OS Debugger creates a `STORAGE` command to modify the value. If you are debugging a program that is optimized, the `STORAGE` command might not modify the value.
- You cannot modify the value of z/OS Debugger variables, except value of registers %GPRn, %FPRn, %EPRn, %LPRn.
- You cannot modify the value of a z/OS Debugger built-in function.
- You cannot modify the value of a PL/I built-in function.
- You cannot modify a complex expression.

If you type quotation marks (") or apostrophes (') in the Monitor value area, carefully verify that they comply with any applicable quotation rules.

Opening and closing the Monitor window

If the Monitor window is closed before you enter the `SET AUTOMONITOR ON` command, z/OS Debugger opens the Monitor window and displays the name and value of the variables of statement you run in the automonitor section of the window.

If the Monitor window is open before you enter the `SET AUTOMONITOR OFF` command and you are watching the value of variables not monitored by `SET AUTOMONITOR ON`, the Monitor window remains open.

Displaying and modifying memory through the Memory window

z/OS Debugger can display sections of memory through the Memory window. You can open the Memory window and have it display a specific section of memory by doing one of the following options:

- Entering the `MEMORY` command and specifying a base address. If the Memory window is already displayed through a physical window, the memory dump area displays memory starting at the base address.

  If the Memory window is not displayed through a physical window, the base address is saved for usage later when the Memory window is displayed through a physical window.

  To display the Memory window through a physical window, use the `WINDOW SWAP MEMORY LOG` command or `PANEL LAYOUT` command.
- Assigning the `MEMORY` command to a PF key. After you assign the `MEMORY` command to a PF key, you can move the cursor to a variable, then press the PF key. If the Memory window is already displayed through a physical window, the memory dump area displays memory starting at the base address. If the Memory window is not displayed through a physical window, the base address is saved for usage later when the Memory window is displayed through a physical window.

  To display the Memory window through a physical window, use the `WINDOW SWAP MEMORY LOG` command or `PANEL LAYOUT` command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Scrolling through the physical windows" on page 165
"Switching between the Memory window and Log window" on page 165
"Displaying memory through the Memory window" on page 16
"Customizing the layout of physical windows on the session panel" on page 252

Related references

"Memory window" on page 154
"Order in which z/OS Debugger accepts commands from the session panel" on page 160
MEMORY command in *IBM z/OS Debugger Reference and Messages*

Modifying memory through the hexadecimal data area

You can type over the hexadecimal data area with hexadecimal characters (0-9, A-F, a-f). z/OS Debugger updates the memory with the value you typed in. If you modify the program instruction area of memory, z/OS Debugger does not do any STEP commands or stop at any AT breakpoints near the area where you modified memory. In addition, if you try to run the program, the results are unpredictable.

The character data column is the character representation of the data and is only for viewing purposes.

Managing file allocations

You can manage files while you are debugging by using the `DESCRIBE ALLOCATIONS`, `ALLOCATE`, and `FREE` commands. You cannot manage files while debugging CICS programs.

To view a current list of allocated files, enter the DESCRIBE ALLOCATIONS command. The following screen displays the command and sample output:

```
  DESCRIBE ALLOCATIONS ;
*  Current allocations:
*  VOLUME CAT DISP       OPEN DDNAME   DSNAME
*  ┌1┐--- ┌2┐- ┌3┐------ ┌4┐- ┌5┐----- ┌6┐-----------------------------------------
*  COD008  *  SHR KEEP     *  EQAZSTEP BCARTER.TEST.LOAD
*  SMS004  *  SHR KEEP                 SHARE.CEE210.SCEERUN
*  COD00B  *  OLD KEEP     *  INSPLOG  BCARTER.DTOOL.LOGV
*  VIO        NEW DELETE      ISPCTL0  SYS02190.T085429.RA000.BCARTER.R0100269
*  COD016  *  SHR KEEP        ISPEXEC  BCARTER.MVS.EXEC
*  IPLB13  *  SHR KEEP                 ISPF.SISPEXEC.VB
*  VIO        NEW DELETE      ISPLST1  SYS02190.T085429.RA000.BCARTER.R0100274
*  IPLB13  *  SHR KEEP     *  ISPMLIB  ISPF.SISPMENU
*  SMS278  *  SHR KEEP                 SHARE.ANALYZ21.SIDIMLIB
*  SHR89A  *  SHR KEEP                 SHARE.ISPMLIB
*  SMS25F  *  SHR KEEP     *  ISPPLIB  SHARE.PROD.ISPPLIB
*  SMS891  *  SHR KEEP                 SHARE.ISPPLIB
*  SMS25F  *  SHR KEEP                 SHARE.ANALYZ21.SIDIPLIB
*  IPLB13  *  SHR KEEP                 ISPF.SISPPENU
*  IPLB13  *  SHR KEEP                 SDSF.SISFPLIB
*  IPLB13  *  SHR KEEP                 SYS1.SBPXPENU
*  COD002  *  OLD KEEP     *  ISPPROF  BCARTER.ISPPROF
*             NEW DELETE      SYSIN    TERMINAL
*             NEW DELETE      SYSOUT   TERMINAL
*             NEW DELETE      SYSPRINT TERMINAL
```

The following list describes each column:

1 VOLUME
The volume serial of the DASD volume that contains the data set.

2 CAT
An asterisk in this column indicates that the data set was located by using the system catalog.

3 DISP
The disposition that is assigned to the data set.

4 OPEN
An asterisk in this column indicates that the file is currently open.

5 DDNAME
DD name for the file.

6 DSNAME
Data set name for a DASD data set:

- DUMMY for a DD DUMMY
- SYSOUT(x) for a SYSOUT data set
- TERMINAL for a file allocated to the terminal
- * for a DD * file

You can allocate files to an existing, cataloged data set by using the ALLOCATE command.

You can free an allocated file by using the FREE command.

By default, the DESCRIBE ALLOCATIONS command lists the files allocated by the current user. You can specify other parameters to list other system allocations, such as the data sets currently allocated to LINK list, LPA list, APF list, system catalogs, Parmlib, and Proclib. The *IBM z/OS Debugger Reference and Messages* describes the parameters you must specify to list this information.

Displaying error numbers for messages in the Log window

When an error message shows up in the Log window without a message ID, you can have the message ID show up as in:

```
EQA1807E The command element d is ambiguous.
```

Either modify your profile or use the SET MSGID ON command. To modify your profile, use the PANEL PROFILE command and set **Show message ID numbers** to YES by typing over the NO.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Customizing profile settings" on page 254

Displaying a list of compile units known to z/OS Debugger

This topics describes what to do if you want to know which compile units are known to z/OS Debugger. This is helpful if you have forgotten the name of a compile unit or the load module that a compile unit belongs to.

To determine which compile units are known to z/OS Debugger, do one of the following options:

- Enter the LIST NAMES CUS command.
- If you are debugging an assembler or disassembly program, enter the SET DISASSEMBLY ON or SET ASSEMBLER ON command, then enter the LIST NAMES CUS command.

After you run the LIST NAMES CUS command, z/OS Debugger displays a list of compile units in the Log window. You can use this list to compose a SET QUALIFY CU command by typing in the words "SET QUALIFY CU" over the name of a compile unit. Then press Enter. z/OS Debugger displays the command constructed from the words that you typed in and the name of the compile unit. Press Enter again to run the command.

For example, after you enter the LIST NAMES CUS command, z/OS Debugger displays the following lines in the Log window:

```
USERID.MFISTART.C(CALC)
USERID.MFISTART.C(PUSHPOP)
USERID.MFISTART.C(READTOKN)
```

If you type "SET QUALIFY CU" over the last line, then press Enter, z/OS Debugger composes the following command into the command line: SET QUALIFY CU "USERID.MFISTART.C(READTOKN)". Press Enter and z/OS Debugger runs the command.

This method saves keystrokes and reduces errors in long commands.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Changing which file appears in the Source window" on page 157

Requesting an attention interrupt during interactive sessions

During an interactive z/OS Debugger session, you can request an attention interrupt, if necessary. For example, you can stop what appears to be an unending loop, stop the display of voluminous output at your terminal, or stop the execution of the STEP command.

An attention interrupt should not be confused with the ATTENTION condition. If you set an AT OCCURRENCE or ON ATTENTION, the commands associated with that breakpoint are not run at an attention interrupt.

Language Environment TRAP and INTERRUPT run-time options should both be set to ON in order for attention interrupts that are recognized by the host operating system to be also recognized by Language Environment. The *test_level* suboption of the TEST run-time option should *not* be set to NONE.

An attention interrupt key is not supported in the following environment and debugging modes:

- CICS
- full-screen mode using the Terminal Interface Manager

For MVS only: For C, when using an attention interrupt, use `SET INTERCEPT ON FILE stdout` to intercept messages to the terminal. This is required because messages do not go to the terminal after an attention interrupt.

For the Dynamic Debug facility only: The Dynamic Debug facility supports attention interrupts only for programs that have compiled-in hooks.

The correct key might not be marked ATTN on every keyboard. Often the following keys are used:

- Under TSO: PA1 key
- Under IMS: PA1 key

When you request an *attention interrupt*, control is given to z/OS Debugger:

- At the next hook if z/OS Debugger has previously gained control or if you specified either `TEST(ERROR)` or `TEST(ALL)` or have specifically set breakpoints
- At a `__ctest()` or CEETEST call
- When an HLL condition is raised in the program, such as `SIGINT` in C

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal" on page 131

Related references

z/OS Language Environment Programming Guide

Ending a full-screen debug session

When you have finished debugging your program, you can end your full-screen debug session by using one of the following methods:

Method A

1. Press PF3 (`QUIT`) or enter `QUIT` on the command line.
2. Type Y to confirm your request and press Enter. Your program stops running.

If you are debugging a CICS non-Language Environment assembler or non-Language Environment COBOL program, `QUIT` ends z/OS Debugger and the task ends with an ABEND 4038.

Method B

1. Enter the `QQUIT` command. You are not prompted to confirm your request to end your debug session. Your program stops running.

If you are debugging a CICS non-Language Environment assembler or non-Language Environment COBOL program, `QUIT` ends z/OS Debugger and the task ends with an ABEND 4038.

Method C

1. Enter the `QUIT DEBUG` or the `QUIT DEBUG TASK` (CICS only) command.
2. Type Y to confirm your request and press Enter. z/OS Debugger ends and your program continues running.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

IBM z/OS Debugger Reference and Messages

Chapter 22. Debugging a COBOL program in full-screen mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for COBOL refer to the following COBOL program.

"Example: sample COBOL program for debugging" on page 197

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 30, "Debugging COBOL programs," on page 267
"Halting when certain routines are called in COBOL" on page 200
"Modifying the value of a COBOL variable" on page 200
"Halting on a COBOL line only if a condition is true" on page 201
"Debugging COBOL when only a few parts are compiled with TEST" on page 202
"Capturing COBOL I/O to the system console" on page 202
"Displaying raw storage in COBOL" on page 203
"Getting a COBOL routine traceback" on page 203
"Tracing the run-time path for COBOL code compiled with TEST" on page 203
"Generating a COBOL run-time paragraph trace" on page 204
"Finding unexpected storage overwrite errors in COBOL" on page 205
"Halting before calling an invalid program in COBOL" on page 205

Example: sample COBOL program for debugging

The program below is used in various topics to demonstrate debugging tasks.

This program calls two subprograms to calculate a loan payment amount and the future value of a series of cash flows. It uses several COBOL intrinsic functions.

Main program COBCALC

```
      ************************************************************
      * COBCALC                                                  *
      *                                                          *
      * A simple program that allows financial functions to      *
      * be performed using intrinsic functions.                  *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBCALC.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  PARM-1.
           05  CALL-FEEDBACK     PIC XX.
       01  FIELDS.
           05  INPUT-1           PIC X(10).
       01  INPUT-BUFFER-FIELDS.
           05  BUFFER-PTR        PIC 9.
           05  BUFFER-DATA.
               10  FILLER        PIC X(10)  VALUE "LOAN".
               10  FILLER        PIC X(10)  VALUE "PVALUE".
               10  FILLER        PIC X(10)  VALUE "pvalue".
               10  FILLER        PIC X(10)  VALUE "END".
           05  BUFFER-ARRAY    REDEFINES BUFFER-DATA
                               OCCURS 4 TIMES
                                 PIC X(10).

       PROCEDURE DIVISION.
           DISPLAY "CALC Begins." UPON CONSOLE.
```

© Copyright IBM Corp. 1992, 2023

```
           MOVE 1 TO BUFFER-PTR.
           MOVE SPACES TO INPUT-1.
      * Keep processing data until END requested
           PERFORM ACCEPT-INPUT UNTIL INPUT-1 EQUAL TO "END".
      * END requested
           DISPLAY "CALC Ends." UPON CONSOLE.
           GOBACK.
      * End of program.

      *
      * Accept input data from buffer
      *
       ACCEPT-INPUT.
           MOVE BUFFER-ARRAY (BUFFER-PTR) TO INPUT-1.
           ADD 1 BUFFER-PTR GIVING BUFFER-PTR.
      * Allow input data to be in UPPER or lower case
           EVALUATE FUNCTION UPPER-CASE(INPUT-1)     CALC1
             WHEN "END"
               MOVE "END" TO INPUT-1
             WHEN "LOAN"
               PERFORM CALCULATE-LOAN
             WHEN "PVALUE"
               PERFORM CALCULATE-VALUE
             WHEN OTHER
               DISPLAY "Invalid input: " INPUT-1
           END-EVALUATE.
      *
      * Calculate Loan via CALL to subprogram
      *
       CALCULATE-LOAN.
           CALL "COBLOAN" USING CALL-FEEDBACK.
           IF CALL-FEEDBACK IS NOT EQUAL "OK" THEN
             DISPLAY "Call to COBLOAN Unsuccessful.".
      *
      * Calculate Present Value via CALL to subprogram
      *
       CALCULATE-VALUE.
           CALL "COBVALU" USING CALL-FEEDBACK.
           IF CALL-FEEDBACK IS NOT EQUAL "OK" THEN
             DISPLAY "Call to COBVALU Unsuccessful.".
```

Subroutine COBLOAN

```
      ************************************************************
      * COBLOAN                                                  *
      *                                                          *
      * A simple subprogram that calculates payment amount       *
      * for a loan.                                              *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBLOAN.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  FIELDS.
           05  INPUT-1           PIC X(26).
           05  PAYMENT           PIC S9(9)V99 USAGE COMP.
           05  PAYMENT-OUT       PIC $$$$,$$$,$$9.99 USAGE DISPLAY.
           05  LOAN-AMOUNT       PIC S9(7)V99 USAGE COMP.
           05  LOAN-AMOUNT-IN    PIC X(16).
           05  INTEREST-IN       PIC X(5).
           05  INTEREST          PIC S9(3)V99 USAGE COMP.
           05  NO-OF-PERIODS-IN  PIC X(3).
           05  NO-OF-PERIODS     PIC 99 USAGE COMP.
           05  OUTPUT-LINE       PIC X(79).
       LINKAGE SECTION.
       01  PARM-1.
           05  CALL-FEEDBACK     PIC XX.
       PROCEDURE DIVISION USING PARM-1.
           MOVE "NO" TO CALL-FEEDBACK.
           MOVE "30000 .09 24 " TO INPUT-1.
           UNSTRING INPUT-1 DELIMITED BY ALL " "
             INTO LOAN-AMOUNT-IN INTEREST-IN NO-OF-PERIODS-IN.
      * Convert to numeric values
           COMPUTE LOAN-AMOUNT = FUNCTION NUMVAL(LOAN-AMOUNT-IN).
           COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).
           COMPUTE NO-OF-PERIODS = FUNCTION NUMVAL(NO-OF-PERIODS-IN).
      * Calculate annuity amount required
           COMPUTE PAYMENT = LOAN-AMOUNT *
```

```
           FUNCTION ANNUITY((INTEREST / 12 ) NO-OF-PERIODS).
      * Make it presentable
           MOVE SPACES TO OUTPUT-LINE
           MOVE PAYMENT TO PAYMENT-OUT.
           STRING "COBLOAN:_Repayment_amount_for_a_" NO-OF-PERIODS-IN
                  "_month_loan_of_" LOAN-AMOUNT-IN
                  "_at_" INTEREST-IN "_interest_is:_"
              DELIMITED BY SPACES
              INTO OUTPUT-LINE.
           INSPECT OUTPUT-LINE REPLACING ALL "_" BY SPACES.
           DISPLAY OUTPUT-LINE PAYMENT-OUT.
           MOVE "OK" TO CALL-FEEDBACK.
           GOBACK.
```

Subroutine COBVALU

```
      ************************************************************
      * COBVALU                                                  *
      *                                                          *
      * A simple subprogram that calculates present value        *
      * for a series of cash flows.                              *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBVALU.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHAR-DATA.
           05  INPUT-1           PIC X(10).
           05  PAYMENT-OUT       PIC $$$$,$$$,$$9.99 USAGE DISPLAY.
           05  INTEREST-IN       PIC X(5).
           05  NO-OF-PERIODS-IN  PIC X(3).
           05  INPUT-BUFFER      PIC X(10) VALUE "5069837544".
           05  BUFFER-ARRAY   REDEFINES INPUT-BUFFER
                              OCCURS 5 TIMES
                                 PIC XX.
           05  OUTPUT-LINE       PIC X(79).
       01 NUM-DATA.
           05  PAYMENT           PIC S9(9)V99 USAGE COMP.
           05  INTEREST          PIC S9(3)V99 USAGE COMP.
           05  COUNTER           PIC 99 USAGE COMP.
           05  NO-OF-PERIODS     PIC 99 USAGE COMP.
           05  VALUE-AMOUNT   OCCURS 99 PIC S9(7)V99 COMP.
       LINKAGE SECTION.
       01  PARM-1.
           05  CALL-FEEDBACK  PIC XX.
       PROCEDURE DIVISION USING PARM-1.
           MOVE "NO" TO CALL-FEEDBACK.
           MOVE ".12 5 " TO INPUT-1.
           UNSTRING INPUT-1 DELIMITED BY "," OR ALL " "              VALU1
            INTO INTEREST-IN NO-OF-PERIODS-IN.
      * Convert to numeric values
           COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).          VALU2
           COMPUTE NO-OF-PERIODS = FUNCTION NUMVAL(NO-OF-PERIODS-IN).
      * Get cash flows
           PERFORM GET-AMOUNTS VARYING COUNTER FROM 1 BY 1 UNTIL
             COUNTER IS GREATER THAN NO-OF-PERIODS.
      * Calculate present value
           COMPUTE PAYMENT =
              FUNCTION PRESENT-VALUE(INTEREST VALUE-AMOUNT(ALL) ).   VALU3
      * Make it presentable
           MOVE PAYMENT TO PAYMENT-OUT.
           STRING "COBVALU:_Present_value_for_rate_of_"
                  INTEREST-IN "_given_amounts_"
                  BUFFER-ARRAY (1) ",_"
                  BUFFER-ARRAY (2) ",_"
                  BUFFER-ARRAY (3) ",_"
                  BUFFER-ARRAY (4) ",_"
                  BUFFER-ARRAY (5) "_is:_"
              DELIMITED BY SPACES
              INTO OUTPUT-LINE.
           INSPECT OUTPUT-LINE REPLACING ALL "_" BY SPACES.
           DISPLAY OUTPUT-LINE PAYMENT-OUT.
           MOVE "OK" TO CALL-FEEDBACK.
           GOBACK.
      *
      * Get cash flows for each period
      *
       GET-AMOUNTS.
```

```
           MOVE BUFFER-ARRAY (COUNTER) TO INPUT-1.
           COMPUTE VALUE-AMOUNT (COUNTER) = FUNCTION NUMVAL(INPUT-1).
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 22, “Debugging a COBOL program in full-screen mode,” on page 197

Halting when certain routines are called in COBOL

This topic describes how to halt just before or just after a routine is called by using the AT CALL or AT ENTRY commands. The “Example: sample COBOL program for debugging” on page 197 is used to describe these commands.

To use the AT CALL command, you must compile the calling program with the TEST compiler option.

To halt just before COBLOAN is called, enter the following command:

```
AT CALL COBLOAN ;
```

To use the AT ENTRY command, you must compile the called program with the TEST compiler option.

To halt just after COBVALU is called, enter the following command:

```
AT ENTRY COBVALU ;
```

To halt just after COBVALU is called and only when CALL-FEEDBACK equals OK, enter the following command:

```
AT ENTRY COBVALU WHEN CALL-FEEDBACK = "OK" ;
```

Identifying the statement where your COBOL program has stopped

If you have many breakpoints set in your program, enter the following command to have z/OS Debugger identify your program has been stopped:

```
QUERY LOCATION
```

The z/OS Debugger Log window displays something similar to the following example:

```
QUERY LOCATION ;
You were prompted because STEP ended.
The program is currently entering block COBVALU.
```

Modifying the value of a COBOL variable

“Example: sample COBOL program for debugging” on page 197

To list the contents of a single variable, move the cursor to an occurrence of the variable name in the Source window and press PF4 (LIST). Remember that z/OS Debugger starts **after** program initialization but **before** symbolic COBOL variables are initialized, so you cannot view or modify the contents of variables until you have performed a step or run. The value is displayed in the Log window. This is equivalent to entering LIST TITLED *variable* on the command line. Run the COBCALC program to the statement labeled CALC1, and enter AT 46 ; GO ; on the z/OS Debugger command line. Move the cursor over INPUT-1 and press LIST (PF4). The following appears in the Log window:

```
 LIST ( INPUT-1 ) ;
INPUT-1 = 'LOAN      '
```

To modify the value of INPUT-1, enter on the command line:

```
MOVE 'pvalue' to INPUT-1 ;
```

You can enter most COBOL expressions on the command line.

Now step into the call to COBVALU by pressing PF2 (STEP) and step until the statement labeled VALU2 is reached. To view the attributes of the variable INTEREST, issue the z/OS Debugger command:

```
DESCRIBE ATTRIBUTES INTEREST ;
```

The result in the Log window is:

```
ATTRIBUTES FOR INTEREST
  ITS LENGTH IS 4
  ITS ADDRESS IS 00011DC8
  02 COBVALU:>INTEREST   S999V99 COMP
```

You can use this action as a simple browser for group items and data hierarchies. For example, you can list all the values of the elementary items for the CHAR-DATA group with the command:

```
LIST CHAR-DATA ;
```

with results in the Log window appearing something like this:

```
 LIST CHAR-DATA ;
02 COBVALU:>INPUT-1  of 01 COBVALU:>CHAR-DATA  = '.12 5     '
Invalid data for 02 COBVALU:>PAYMENT-OUT  of 01 COBVALU:>CHAR-DATA   is found.
02 COBVALU:>INTEREST-IN  of 01 COBVALU:>CHAR-DATA  = '.12  '
02 COBVALU:>NO-OF-PERIODS-IN  of 01 COBVALU:>CHAR-DATA  = '5  '
02 COBVALU:>INPUT-BUFFER  of 01 COBVALU:>CHAR-DATA  = '5069837544'
SUB(1) of 02 COBVALU:>BUFFER-ARRAY  of 01 COBVALU:>CHAR-DATA  = '50'
SUB(2) of 02 COBVALU:>BUFFER-ARRAY  of 01 COBVALU:>CHAR-DATA  = '69'
SUB(3) of 02 COBVALU:>BUFFER-ARRAY  of 01 COBVALU:>CHAR-DATA  = '83'
SUB(4) of 02 COBVALU:>BUFFER-ARRAY  of 01 COBVALU:>CHAR-DATA  = '75'
SUB(5) of 02 COBVALU:>BUFFER-ARRAY  of 01 COBVALU:>CHAR-DATA  = '44'
```

Note: If you use the LIST command to list the contents of an uninitialized variable, or a variable that contains invalid data, z/OS Debugger displays INVALID DATA.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using COBOL variables with z/OS Debugger" on page 268

Halting on a COBOL line only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but it fails under certain conditions. You don't want to just set a line breakpoint because you will have to keep entering GO.

"Example: sample COBOL program for debugging" on page 197

For example, in COBVALU you want to stop at the calculation of present value only if the discount rate is less than or equal to -1 (before the exception occurs). First run COBCALC, step into COBVALU, and stop at the statement labeled VALU1. To accomplish this, issue these z/OS Debugger commands at the start of COBCALC:

```
AT 67 ; GO ;
CLEAR AT 67 ; STEP 4 ;
```

Now set the breakpoint like this:

```
AT 44 IF INTEREST > -1 THEN GO ; END-IF ;
```

Line 44 is the statement labeled VALU3. The command causes z/OS Debugger to stop at line 44. If the value of INTEREST is greater than -1, the program continues. The command causes z/OS Debugger to remain on line 44 only if the value of INTEREST is less than or equal to -1.

To force the discount rate to be negative, enter the z/OS Debugger command:

```
MOVE '-2 5' TO INPUT-1 ;
```

Run the program by issuing the GO command. z/OS Debugger halts the program at line 44. Display the contents of INTEREST by issuing the LIST INTEREST command. To view the effect of this breakpoint when the discount rate is positive, begin a new debug session and repeat the z/OS Debugger commands shown in this section. However, do not issue the MOVE '-2 5' TO INPUT-1 command. The program execution does not stop at line 44 and the program runs to completion.

Debugging COBOL when only a few parts are compiled with TEST

"Example: sample COBOL program for debugging" on page 197

Suppose you want to set a breakpoint at entry to COBVALU. COBVALU has been compiled with TEST but the other programs have not. z/OS Debugger comes up with an empty Source window. You can use the LIST NAMES CUS command to determine if the COBVALU compile unit is known to z/OS Debugger and then set the appropriate breakpoint using either the AT APPEARANCE or the AT ENTRY command.

Instead of setting a breakpoint at entry to COBVALU in this example, issue a STEP command when z/OS Debugger initially displays the empty Source window. z/OS Debugger runs the program until it reaches the entry for the first routine compiled with TEST, COBVALU in this case.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Halting when certain routines are called in COBOL" on page 200

Capturing COBOL I/O to the system console

To redirect output normally appearing on the system console to your z/OS Debugger terminal, enter the following command:

```
SET INTERCEPT ON CONSOLE ;
```

"Example: sample COBOL program for debugging" on page 197

For example, if you run COBCALC and issue the z/OS Debugger SET INTERCEPT ON CONSOLE command, followed by the STEP 3 command, you will see the following output displayed in the z/OS Debugger Log window:

```
SET INTERCEPT ON CONSOLE ;
STEP 3 ;
CONSOLE : CALC Begins.
```

The phrase CALC Begins. is displayed by the statement DISPLAY "CALC Begins." UPON CONSOLE in COBCALC.

The SET INTERCEPT ON CONSOLE command not only captures output to the system console, but also allows you to input data from your z/OS Debugger terminal instead of the system console by using the z/OS Debugger INPUT command. For example, if the next COBOL statement executed is ACCEPT INPUT-DATA FROM CONSOLE, the following message appears in the z/OS Debugger Log window:

```
CONSOLE : IGZ0000I AWAITING REPLY.
The program is waiting for input from CONSOLE.
Use the INPUT command to enter 114 characters for the intercepted
fixed-format file.
```

Continue execution by replying to the input request by entering the following z/OS Debugger command:

```
INPUT some data ;
```

Note: Whenever z/OS Debugger intercepts system console I/O, and for the duration of the intercept, the display in the Source window is empty and the Location field in the session panel header at the top of the screen shows *Unknown*.

Displaying raw storage in COBOL

You can display the storage for a variable by using the LIST STORAGE command. For example, to display the storage for the first 12 characters of BUFFER-DATA enter:

```
LIST STORAGE(BUFFER-DATA,12)
```

You can also display only a section of the data. For example, to display the storage that starts at offset 4 for a length of 6 characters, enter:

```
LIST STORAGE(BUFFER-DATA,4,6)
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Displaying and modifying memory through the Memory window” on page 192

Getting a COBOL routine traceback

Often when you get close to a programming error, you want to know how you got into that situation, and especially what the traceback of calling routines is. To get this information, issue the command:

```
LIST CALLS ;
```

“Example: sample COBOL program for debugging” on page 197

For example, if you run the COBCALC example with the commands:

```
AT APPEARANCE COBVALU AT ENTRY COBVALU;
GO;
GO;
LIST CALLS;
```

the Log window contains something like:

```
 AT APPEARANCE COBVALU
   AT ENTRY COBVALU ;
 GO ;
 GO ;
 LIST CALLS ;
At ENTRY in COBOL program COBVALU.
From LINE 67.1 in COBOL program COBCALC.
```

which shows the traceback of callers.

Tracing the run-time path for COBOL code compiled with TEST

To trace a program showing the entry and exit points without requiring any changes to the program, place the following z/OS Debugger commands in a file or data set and USE them when z/OS Debugger initially displays your program. Assuming you have a PDS member, USERID.DT.COMMANDS(COBCALC), that contains the following z/OS Debugger commands:

```
* Commands in a COBOL USE file must be coded in columns 8-72.
* If necessary, commands can be continued by coding a '-' in
* column 7 of the continuation line.
 01 LEVEL PIC 99 USAGE COMP;
 MOVE 1 TO LEVEL;
 AT ENTRY * PERFORM;
   COMPUTE LEVEL = LEVEL + 1;
   LIST ( "Entry:", LEVEL, %CU);
   GO;
   END-PERFORM;
 AT EXIT * PERFORM;
   LIST ( "Exit:", LEVEL);
   COMPUTE LEVEL = LEVEL - 1;
   GO;
   END-PERFORM;
```

You can use this file as the source of commands to z/OS Debugger by entering the following command:

```
USE USERID.DT.COMMANDS(COBCALC)
```

If, after executing the USE file, you run COBCALC, the following trace (or similar) is displayed in the Log window:

```
ENTRY:
LEVEL = 00002
%CU = COBCALC
ENTRY:
LEVEL = 00003
%CU = COBLOAN
EXIT:
LEVEL = 00003
ENTRY:
LEVEL = 00003
%CU = COBVALU
EXIT:
LEVEL = 00003
ENTRY:
LEVEL = 00003
%CU = COBVALU
EXIT:
LEVEL = 00003
EXIT:
LEVEL = 00002
```

If you do not want to create the USE file, you can enter the commands through the command line, and the same effect is achieved.

Generating a COBOL run-time paragraph trace

To generate a trace showing the names of paragraphs through which execution has passed, the z/OS Debugger commands shown in the following example can be used. You can either enter the commands from the z/OS Debugger command line or place the commands in a file or data set.

“Example: sample COBOL program for debugging” on page 197

Assume you have a PDS member, USERID.DT.COMMANDS(COBCALC2), that contains the following z/OS Debugger commands.

```
* COMMANDS IN A COBOL USE FILE MUST BE CODED IN COLUMNS 8-72.
* IF NECESSARY, COMMANDS CAN BE CONTINUED BY CODING A '-' IN
* COLUMN 7 OF THE CONTINUATION LINE.
 AT GLOBAL LABEL PERFORM;
   LIST LINES %LINE;
   GO;
 END-PERFORM;
```

When z/OS Debugger initially displays your program, enter the following command:

```
USE USERID.DT.COMMANDS(COBCALC2)
```

After executing the USE file, you can run COBCALC and the following trace (or similar) is displayed in the Log window:

```
     42       ACCEPT-INPUT.
     59       CALCULATE-LOAN.
     42       ACCEPT-INPUT.
     66       CALCULATE-VALUE.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     42       ACCEPT-INPUT.
     66       CALCULATE-VALUE.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     64       GET-AMOUNTS.
     42       ACCEPT-INPUT.
```

Finding unexpected storage overwrite errors in COBOL

During program run time, some storage might unexpectedly change its value and you want to find out when and where this happened. Consider this example where the program changes more than the caller expects it to change.

```
   05  FIELD-1     OCCURS 2 TIMES
                       PIC X(8).
   05  FIELD-2         PIC X(8).
 PROCEDURE DIVISION.
*                    ( An invalid index value is set )
     MOVE 3 TO PTR.
     MOVE "TOO MUCH" TO FIELD-1( PTR ).
```

Find the address of FIELD-2 with the command:

```
DESCRIBE ATTRIBUTES FIELD-2
```

Suppose the result is X'0000F559'. To set a breakpoint that watches for a change in storage values starting at that address for the next 8 bytes, issue the command:

```
AT CHANGE %STORAGE(H'0000F559',8)
```

When the program runs, z/OS Debugger halts if the value in this storage changes.

Halting before calling an invalid program in COBOL

Calling an undefined program is a severe error. If you have developed a main program that calls a subprogram that doesn't exist, you can cause z/OS Debugger to halt just before such a call. For example, if the subprogram NOTYET doesn't exist, you can set the breakpoint:

```
AT CALL (NOTYET)
```

When z/OS Debugger stops at this breakpoint, you can bypass the CALL by entering the GO BYPASS command. This allows you to continue your debug session without raising a condition.

Chapter 23. Debugging a LangX COBOL program in full-screen mode

Note: This chapter is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for LangX COBOL refer to the following program.

"Example: sample LangX COBOL program for debugging" on page 207

As you read through the information in this document, remember that OS/VS COBOL programs are non-Language Environment programs, even though you might have used Language Environment libraries to link and run your program.

VS COBOL II programs are non-Language Environment programs when you link them with the non-Language Environment library. VS COBOL II programs are Language Environment programs when you link them with the Language Environment library.

Enterprise COBOL programs are always Language Environment programs. Note that COBOL DLL's cannot be debugged as LangX COBOL programs.

Read the information regarding non-Language Environment programs for instructions on how to start z/OS Debugger and debug non-Language Environment COBOL programs, unless information specific to LangX COBOL is provided.

Example: sample LangX COBOL program for debugging

The program below is used in various topics to demonstrate debugging tasks. It is an OS/VS COBOL program which is being used as a representative of LangX COBOL programs.

To run this sample program, do the following steps:

1. Prepare the sample program as described in Chapter 6, "Preparing a LangX COBOL program," on page 67.
2. Verify that the debug information for this program is located in the COB03O and COB03AO members of the *yourid*.EQALANGX data set.
3. Start z/OS Debugger as described in "Starting z/OS Debugger for programs that start outside of Language Environment" on page 134.
4. To load the debug information for this program, enter the following command:

```
LDD (COB03O,COB03AO) ;
```

This program is a small example of an OS/VS COBOL program (COB03O) that calls another OS/VS COBOL program (COB03A0).

Load module: COB03O

COB03O

```
      ********************************************************
      *  PROGRAM NAME: COB03O                                *
      *                                                      *
      *  COMPILED WITH IBM OS/VS COBOL COMPILER              *
      ********************************************************

       IDENTIFICATION DIVISION.
       PROGRAM-ID. COB03O.
      ************************************************************************
      *                                                                      *
      * LICENSED MATERIALS - PROPERTY OF IBM                                 *
      *                                                                      *
      * 5655-P14: Debug Tool                                          *
      * (C) Copyright IBM Corp. 2005 All Rights Reserved                     *
```

© Copyright IBM Corp. 1992, 2023

```
      *                                                               *
      * US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR    *
      * DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM    *
      * CORP.                                                         *
      *                                                               *
      *                                                               *
      *****************************************************************
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01  LOAN          PIC 999999.
       01  INTEREST-RATE PIC 99V99.
       01  INTEREST-DUE  PIC 999999.
       01  INTEREST-SAVE PIC 999999.
       01  INTEREST-AFTER-MULTIPLY PIC 999999.
       01  INTEREST-AFTER-DIVIDE PIC 999999.

      * DATE THAT WILL RECEIVE INCREMENTED JULIAN-DATE
       01  INC-DATE            PIC 9(7).
      * LOOP COUNT TO INCREMENT DATE 1000 TIMES *
       01  LOOPCOUNT           PIC   9999.

      * JULIAN DATE
       01  JULIAN-DATE         PIC 9(7).
       01  J-DATE REDEFINES JULIAN-DATE.
           05 J-YEAR           PIC 9(4).
           05 J-DAY            PIC 9(3).
      * SAVE DATE
       01  SAVE-DATE           PIC 9(7).

       PROCEDURE DIVISION.

       PROG.
           ACCEPT JULIAN-DATE FROM DAY
           DISPLAY 'JULIAN DATE: ' JULIAN-DATE
           MOVE JULIAN-DATE TO SAVE-DATE

           MOVE 10000    TO LOAN

           CALL 'COB03AO' USING LOAN INTEREST-DUE.

           DISPLAY 'LOAN: ' LOAN
           DISPLAY 'INTEREST-DUE: ' INTEREST-DUE

           STOP RUN.
```

COB03AO

```
      ******************************************************
      *  PROGRAM NAME: COB03AO                             *
      *                                                    *
      *  COMPILED WITH IBM OS/VS COBOL COMPILER            *
      ******************************************************

       IDENTIFICATION DIVISION.
       PROGRAM-ID. COB03AO.
      *****************************************************************
      *                                                               *
      * LICENSED MATERIALS - PROPERTY OF IBM                          *
      *                                                               *
      * 5655-P14: Debug Tool                                      *
      * (C) Copyright IBM Corp. 2005 All Rights Reserved              *
      *                                                               *
      * US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR    *
      * DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM    *
      * CORP.                                                         *
      *                                                               *
      *                                                               *
      *****************************************************************
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  INTEREST-RATE PIC 99V99 VALUE 0.22.
       LINKAGE SECTION.
       01  USING-LIST.
           02 LOANAMT   PIC 999999.
           02 INTEREST  PIC 999999.
```

```
        PROCEDURE DIVISION USING USING-LIST.

        PROG.
            COMPUTE INTEREST = LOANAMT * INTEREST-RATE.
            DISPLAY 'INTEREST-RATE: ' INTEREST-RATE.

            GOBACK.
```

Defining a compilation unit as LangX COBOL and loading debug information

Before you can debug a LangX COBOL program, you must define the compilation unit (CU) as a LangX COBOL CU and load the debug data for the CU. This can only be done for a CU that is currently known to z/OS Debugger as a disassembly CU or for a CU that is not currently known to z/OS Debugger.

You use the LOADDEBUGDATA command (abbreviated as LDD) to define a disassembly CU as a LangX COBOL CU and to cause the debug data for this CU to be loaded. When you invoke the LDD command, you can specify either a single CU name or a list of CU names enclosed in parenthesis. Each of the names specified must be either:

- the name of a disassembly CU that is currently known to z/OS Debugger
- a name that does not match the name of a CU currently known to z/OS Debugger

When the CU name is currently known to z/OS Debugger, the CU is immediately marked as a LangX COBOL CU and an attempt is made to load the debug as follows:

- If your debug data is in a partitioned data set where the high-level qualifier is the current user ID, the low-level qualifier is EQALANGX, and the member name is the same as the name of the CU that you want to debug no other action is necessary
- If your debug data is in a different partitioned data set than *userid*.EQALANGX but the member name is the same as the name of the CU that you want to debug, enter the following command before or after you enter the LDD command: `SET DEFAULT LISTINGS`
- If your debug data is in a sequential data set or is a member of a partitioned data set but the member name is different from the CU name, enter the following command before or after the LDD command: `SET SOURCE`

When the CU name specified on the LDD command is not currently known to z/OS Debugger, a message is issued and the LDD command is deferred until a CU by that name becomes known (appears). At that time, the CU is automatically created as a LangX COBOL CU and an attempt is made to load the debug data using the default data set name or the current SET DEFAULT LISTINGS specification.

After you have entered an LDD command for a CU, you cannot view the CU as a disassembly CU.

If z/OS Debugger cannot find the associated debug data after you have entered an LDD command, the CU is a LangX COBOL CU rather than a disassembly CU. You cannot enter another LDD command for this CU. However, you can enter a `SET DEFAULT LISTING` command or a `SET SOURCE` command to cause the associated debug data to be loaded from a different data set.

Defining a compilation unit in a different load module as LangX COBOL

You must use the LDD command to identify a CU as a LangX COBOL CU. If the CU is part of a load module that has not yet been loaded when you enter the LDD command, z/OS Debugger displays a message indicating that the CU was not found and that the running of the LDD command has been deferred. If the CU later appears as a disassembly CU, the LDD command is run at that time.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Defining a compilation unit as LangX COBOL and loading debug information” on page 209

Halting when certain LangX COBOL programs are called

"Example: sample LangX COBOL program for debugging" on page 207

To halt after the COB03AO routine is called, enter the following command:

```
AT ENTRY COB03AO ;
```

The AT CALL command is not supported for LangX COBOL routines. Do not use the AT CALL command to halt z/OS Debugger when a LangX COBOL routine is called.

Identifying the statement where your LangX COBOL program has stopped

If you have many breakpoints set in your program and you want to know where your program was halted, you can enter the following command:

```
QUERY LOCATION
```

The z/OS Debugger Log window displays a message similar to the following message:

```
QUERY LOCATION
You are executing commands in the ENTRY COB03O ::> COB03AO breakpoint.
The program is currently entering block COB03O ::> COB03AO.
```

Displaying and modifying the value of LangX COBOL variables or storage

To display the contents of a single variable, move the cursor to an occurrence of the variable name in the Source window and press PF4 (LIST). The value is displayed in the Log window. This is equivalent to entering the LIST *variable* command on the command line.

For example, run the COB03O program to the CALL statement by entering AT 56 ; GO ; on the z/OS Debugger command line. Move the cursor over LOAN and press PF4 (LIST). z/OS Debugger displays the following message in the Log window:

```
LIST ( 'LOAN ')
LOAN = 10000
```

To change the value of LOAN to 100, type 'LOAN' = '100' in the command line and press Enter.

To view the attributes of variable LOAN, enter the following command:

```
DESCRIBE ATTRIBUTES 'LOAN'
```

z/OS Debugger displays the following messages in the Log window:

```
ATTRIBUTES for LOAN
   Its address is 0002E500 and its length is 6
   LOAN PIC 999999
```

To step into the call to COB03AO, press PF2 (STEP).

Halting on a line in LangX COBOL only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but it fails under certain conditions. Setting a line breakpoint is inefficient because you will have to repeatedly enter the GO command.

"Example: sample LangX COBOL program for debugging" on page 207

In the COB03AO program, to halt z/OS Debugger when the LOANAMT variable is set to 100, enter the following command:

```
AT 36 DO; IF 'LOANAMT ¬= 100' THEN GO; END;
```

Line 36 is the line COMPUTE INTEREST = LOANAMT * INTEREST-RATE. The command causes z/OS Debugger to stop at line 36. If the value of LOANAMT is not 100, the program continues. The command causes z/OS Debugger to stop on line 36 only if the value of LOANAMT is 100.

Debugging LangX COBOL when debug information is only available for a few parts

"Example: sample LangX COBOL program for debugging" on page 207

Suppose you want to set a breakpoint at the entry point to COB03AO program and that debug information is available for COB03AO but not for COB03O. In this circumstance, z/OS Debugger would display an empty Source window. To display a list of compile units known to z/OS Debugger, enter the following commands:

```
SET ASSEMBLER ON
LIST NAMES CUS
```

The `LIST NAMES CUS` command displays a list of all the compile units that are known to z/OS Debugger. If COB03AO is fetched later on by the application, it might not be known to z/OS Debugger. Enter the following commands:

```
LDD COB03AO
AT ENTRY COB03AO
GO
```

Getting a LangX COBOL program traceback

Often when you get close to a programming error, you want to know what sequence of calls lead you to the programming error. This sequence is called a traceback or a traceback of callers. To get the traceback information, enter the following command:

```
LIST CALLS
```

"Example: sample LangX COBOL program for debugging" on page 207

For example, if you run the example with the following commands, the Log window displays the traceback of callers:

```
LDD (COB03O,COB03AO) ;
AT ENTRY COB03AO ;
GO ;
LIST CALLS ;
```

The Log window displays information similar to the following:

```
At ENTRY in LangX COBOL program COB03O ::> COB03AO.
From LINE 74 in LangX COBOL program COB03O ::> COB03O.
```

Finding unexpected storage overwrite errors in LangX COBOL

While your program is running, some storage might unexpectedly change its value and you want to find out when and where this happened. Suppose in the example described in "Getting a LangX COBOL

program traceback” on page 211, the program finds the value of LOAN unexpectedly modified. To set a breakpoint that watches for a change in the value of LOAN, enter the following command:

```
AT CHANGE 'LOAN';
```

When the program runs, z/OS Debugger stops if the value of LOAN changes.

Chapter 24. Debugging a PL/I program in full-screen mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for PL/I refer to the following PL/I program.

"Example: sample PL/I program for debugging" on page 213

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 32, "Debugging PL/I programs," on page 283
"Halting when certain PL/I functions are called" on page 216
"Modifying the value of a PL/I variable" on page 216
"Halting on a PL/I line only if a condition is true" on page 217
"Debugging PL/I when only a few parts are compiled with TEST" on page 217
"Displaying raw storage in PL/I" on page 218
"Getting a PL/I function traceback" on page 218
"Tracing the run-time path for PL/I code compiled with TEST" on page 218
"Finding unexpected storage overwrite errors in PL/I" on page 219
"Halting before calling an undefined program in PL/I" on page 220

Example: sample PL/I program for debugging

The program below is used in various topics to demonstrate debugging tasks.

This program is a simple calculator that reads its input from a character buffer. If integers are read, they are pushed on a stack. If one of the operators (+ - * /) is read, the top two elements are popped off the stack, the operation is performed on them and the result is pushed on the stack. The = operator writes out the value of the top element of the stack to a buffer.

Before running PLICALC, you need to allocate SYSPRINT to the terminal by entering the following command:

```
ALLOC FI(SYSPRINT) DA(*) REUSE
```

Main program PLICALC

```
plicalc: proc options(main);
/*-------------------------------------------------------------------*/
/*                                                                   */
/* A simple calculator that does operations on integers that         */
/* are pushed and popped on a stack                                  */
/*                                                                   */
/*-------------------------------------------------------------------*/
dcl index  builtin;
dcl length builtin;
dcl substr builtin;
/*                                                                   */
dcl 1 stack,
      2 stkptr fixed bin(15,0) init(0),
      2 stknum(50) fixed bin(31,0);
dcl 1 bufin,
      2 bufptr fixed bin(15,0) init(0),
      2 bufchr char (100) varying;
dcl 1 tok char (100) varying;
dcl 1 tstop char(1) init ('s');
dcl 1 ndx fixed bin(15,0);
```

© Copyright IBM Corp. 1992, 2023

```
dcl num      fixed bin(31,0);
dcl i        fixed bin(31,0);
dcl push entry external;
dcl pop  entry returns (fixed bin(31,0)) external;
dcl readtok entry returns (char (100) varying) external;
/*-------------------------------------------------------------------*/
/* input  action:                                                    */
/*    2      push 2 on stack                                         */
/*    18     push 18                                                 */
/*    +      pop 2, pop 18, add, push result (20)                    */
/*    =      output value on the top of the stack (20)               */
/*    5      push 5                                                  */
/*    /      pop 5, pop 20, divide, push result (4)                  */
/*    =      output value on the top of the stack (4)                */
/*-------------------------------------------------------------------*/
bufchr = '2 18 + = 5 / =';
do while (tok ^= tstop);
  tok = readtok(bufin);          /* get next 'token' */
  select (tok);
    when (tstop)
      leave;
    when ('+') do;
      num = pop(stack);
      call push(stack,num);      /*  CALC1  statement */
    end;
    when ('-') do;
      num = pop(stack);
      call push(stack,pop(stack)-num);
    end;
    when ('*')
      call push(stack,pop(stack)*pop(stack));
    when ('/') do;
      num = pop(stack);
      call push(stack,pop(stack)/num); /*  CALC2  statement */
    end;
    when ('=') do;
      num = pop(stack);
      put list ('PLICALC: ', num) skip;
      call push(stack,num);
    end;
    otherwise do;/* must be an integer */
      num = atoi(tok);
      call push(stack,num);
    end;
  end;
end;
return;
```

TOK function

```
atoi: procedure(tok) returns (fixed bin(31,0));
/*-------------------------------------------------------------------*/
/*                                                                   */
/* convert character string to number                                */
/* (note: string validated by readtok)                               */
/*                                                                   */
/*-------------------------------------------------------------------*/
  dcl 1 tok char (100) varying;
  dcl 1 num fixed bin (31,0);
  dcl 1 j fixed bin(15,0);
  num = 0;
  do j = 1 to length(tok);
    num = (10 * num) + (index('0123456789',substr(tok,j,1))-1);
  end;
  return (num);
end atoi;
end plicalc;
```

PUSH function

```
push: procedure(stack,num);
/*-------------------------------------------------------------------*/
/*                                                                   */
/* a simple push function for a stack of integers                    */
/*                                                                   */
/*-------------------------------------------------------------------*/
dcl 1 stack connected,
      2 stkptr fixed bin(15,0),
      2 stknum(50) fixed bin(31,0);
```

```
dcl num      fixed bin(31,0);
stkptr = stkptr + 1;
stknum(stkptr) = num; /*  PUSH1  statement */
return;
end push;
```

POP function

```
pop: procedure(stack) returns (fixed bin(31,0));
/*--------------------------------------------------------------------*/
/*                                                                    */
/* a simple pop function for a stack of integers                      */
/*                                                                    */
/*--------------------------------------------------------------------*/
dcl 1 stack connected,
      2 stkptr fixed bin(15,0),
      2 stknum(50) fixed bin(31,0);
stkptr = stkptr - 1;
return (stknum(stkptr+1));
end pop;
```

READTOK function

```
readtok: procedure(bufin) returns (char (100) varying);
/*--------------------------------------------------------------------*/
/*                                                                    */
/* a function to read input and tokenize it for a simple calculator */
/*                                                                    */
/* action: get next input char, update index for next call          */
/* return: next input char(s)                                         */
/*--------------------------------------------------------------------*/
dcl length builtin;
dcl substr builtin;
dcl verify builtin;
dcl 1 bufin connected,
      2 bufptr fixed bin(15,0),
      2 bufchr char (100) varying;
dcl 1 tok char (100) varying;
dcl 1 tstop char(1) init ('s');
dcl 1 j fixed bin(15,0);
                              /* start of processing */
if bufptr > length(bufchr) then do;
  tok = tstop;
  return ( tok );
end;
bufptr = bufptr + 1;
do while (substr(bufchr,bufptr,1) = ' ');
  bufptr = bufptr + 1;
  if bufptr > length(bufchr) then do;
    tok = tstop;
    return ( tok );
  end;
end;
tok = substr(bufchr,bufptr,1); /* get ready to return single char */
select (tok);
  when ('+','-','/','*','=')
    bufptr = bufptr;
  otherwise do;               /* possibly an integer */
    tok = '';
    do j = bufptr to length(bufchr);
      if verify(substr(bufchr,j,1),'0123456789') ^= 0 then
        leave;
    end;
    if j > bufptr then do;
       j = j - 1;
      tok = substr(bufchr,bufptr,(j-bufptr+1));
      bufptr = j;
    end;
    else
      tok = tstop;
  end;
end;
return (tok);
end readtok;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 24, “Debugging a PL/I program in full-screen mode,” on page 213

Halting when certain PL/I functions are called

This topic describes how to halt just before or just after a routine is called by using the AT CALL and AT ENTRY commands. The “Example: sample PL/I program for debugging” on page 213 is used to describe these commands.

To use the AT CALL command, you must compile the calling program with the TEST compiler option.

To halt just before READTOK is called, enter the following command:

```
AT CALL READTOK ;
```

To use the AT ENTRY command, you must compile the called program with the TEST compiler option.

To halt just after READTOK is called, enter the following command:

```
AT ENTRY READTOK ;
```

To halt just after TOK is called and only when the parameter tok equals 2, enter the following command:

```
AT ENTRY TOK WHEN tok='2';
```

Identifying the statement where your PL/I program has stopped

If you have many breakpoints set in your program, enter the following command to have z/OS Debugger identify where your program has stopped:

```
QUERY LOCATION
```

The z/OS Debugger Log window displays something similar to the following example:

```
QUERY LOCATION ;
You are executing commands in the ENTRY READTOK breakpoint.
The program is currently entering block READTOK.
```

Modifying the value of a PL/I variable

To list the contents of a single variable, move the cursor to an occurrence of the variable name in the Source window and press PF4 (LIST). The value is displayed in the Log window. This is equivalent to entering LIST TITLED variable on the command line. For example, run the PLICALC program to the statement labeled CALC1 by entering AT 22 ; GO ; on the z/OS Debugger command line. Move the cursor over NUM and press PF4 (LIST). The following appears in the Log window:

```
 LIST NUM ;
NUM =             18
```

To modify the value of NUM to 22, type over the NUM = 18 line with NUM = 22, press Enter to put it on the command line, and press Enter again to issue the command.

You can enter most PL/I expressions on the command line.

Now step into the call to PUSH by pressing PF2 (STEP) and step until the statement labeled PUSH1 is reached. To view the attributes of variable STKNUM, enter the z/OS Debugger command:

```
DESCRIBE ATTRIBUTES STKNUM;
```

The result in the Log window is:

```
 ATTRIBUTES FOR STKNUM
   ITS ADDRESS IS  0003944C AND ITS LENGTH IS 200
```

```
     PUSH : STACK.STKNUM(50)  FIXED BINARY(31,0) REAL PARAMETER
         ITS ADDRESS IS  0003944C AND ITS LENGTH IS 4
```

You can list all the values of the members of the structure pointed to by STACK with the command:

```
LIST STACK;
```

with results in the Log window appearing something like this:

```
 LIST STACK ;
STACK.STKPTR =                 2
STACK.STKNUM(1) =              2
STACK.STKNUM(2) =             18
STACK.STKNUM(3) =         233864
:
STACK.STKNUM(50) =        121604
```

You can change the value of a structure member by issuing the assignment as a command as in the following example:

```
STKNUM(STKPTR) = 33;
```

Halting on a PL/I line only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but it fails under certain conditions. You don't want to just set a line breakpoint because you will have to keep entering GO.

“Example: sample PL/I program for debugging” on page 213

For example, in PLICALC you want to stop at the division selection only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 31 DO; IF NUM ^= 0 THEN GO; END;
```

Line 31 is the statement labeled CALC2. The command causes z/OS Debugger to stop at line 31. If the value of NUM is not 0, the program continues. The command causes z/OS Debugger to stop on line 31 only if the value of NUM is 0.

Debugging PL/I when only a few parts are compiled with TEST

“Example: sample PL/I program for debugging” on page 213

Suppose you want to set a breakpoint at entry to subroutine PUSH. PUSH has been compiled with TEST, but the other files have not. z/OS Debugger comes up with an empty Source window. To display the compile units, enter the command:

```
LIST NAMES CUS
```

The `LIST NAMES CUS` command displays a list of all the compile units that are known to z/OS Debugger. If PUSH is fetched later on by the application, this compile unit might not be known to z/OS Debugger. If it is displayed, enter:

```
SET QUALIFY CU PUSH
AT ENTRY PUSH;
GO ;
```

If it is not displayed, set an appearance breakpoint as follows:

```
AT APPEARANCE PUSH ;
GO ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE PUSH AT ENTRY PUSH; GO;
```

The only purpose for this appearance breakpoint is to gain control the **first** time a function in the PUSH compile unit is run. When that happens, you can set a breakpoint at entry to PUSH like this:

```
AT ENTRY PUSH;
```

Displaying raw storage in PL/I

You can display the storage for a variable by using the `LIST STORAGE` command. For example, to display the storage for the first 30 characters of STACK enter:

```
LIST STORAGE(STACK,30)
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Displaying and modifying memory through the Memory window" on page 192

Getting a PL/I function traceback

Often when you get close to a programming error, you want to know how you got into that situation, and especially what the traceback of calling functions is. To get this information, issue the command:

```
LIST CALLS ;
```

"Example: sample PL/I program for debugging" on page 213

For example, if you run the PLICALC example with the commands:

```
AT ENTRY READTOK ;
GO ;
LIST CALLS ;
```

the Log window will contain something like:

```
At ENTRY IN PL/I subroutine READTOK.
From LINE 17.1 IN PL/I subroutine PLICALC.
```

which shows the traceback of callers.

Tracing the run-time path for PL/I code compiled with TEST

To trace a program showing the entry and exit points without changing the program, you can enter the commands described in step 1 by using a commands file or by entering the commands individually. To use a commands file, do the following steps:

1. Create a PDS member with a name similar to the following name: *userid*`.DT.COMMANDS(PLICALL)`
2. Edit the file or data set and add the following z/OS Debugger commands:

```
SET PROGRAMMING LANGUAGE PLI ;
DCL LVLSTR CHARACTER (50);
DCL LVL FIXED BINARY (15);
LVL = 0;
AT ENTRY *
DO;
LVLSTR = ' ' ;
LVL = LVL + 1 ;
LVLSTR = 'ENTERING >' || %BLOCK;
LIST UNTITLED ( LVLSTR ) ;
GO ;
END;
AT EXIT *
DO;
LVLSTR = 'EXITING < ' || %BLOCK;
LIST UNTITLED ( LVLSTR ) ;
LVL = LVL - 1 ;
```

```
GO ;
END;
```

3. Start z/OS Debugger.
4. Enter the following command:

```
USE DT.COMMANDS(PLICALL)
```

5. Run your program sequence. z/OS Debugger displays the trace in the Log window.

For example, after you enter the USE command, you run the following program sequence:

```
*PROCESS MACRO,OPT(TIME);
 *PROCESS S STMT TEST(ALL);

 PLICALL: PROC OPTIONS (MAIN);

 DCL PLIXOPT CHAR(60) VAR STATIC EXTERNAL

 INIT('STACK(20K,20K),TEST');

 CALL PLISUB;

 PUT SKIP LIST('DONE WITH PLICALL');

 PLISUB: PROC;

 DCL PLISUB1 ENTRY ;

 CALL PLISUB1;

 PUT SKIP LIST('DONE WITH PLISUB ');

 END PLISUB;

 PLISUB1: PROC;

 DCL PLISUB2 ENTRY ;

 CALL PLISUB2;

 PUT SKIP LIST('DONE WITH PLISUB1');

 END PLISUB1;

 PLISUB2: PROC;

 PUT SKIP LIST('DONE WITH PLISUB2');
 END PLISUB2;
 END PLICALL;
```

In the Log window, z/OS Debugger displays a trace similar to the following trace:

```
'ENTERING >PLICALL                                '
'ENTERING >PLISUB                                 '
'ENTERING >PLISUB1                                '
'ENTERING >PLISUB2                                '
'EXITING < PLISUB2                                '
'EXITING < PLISUB1                                '
'EXITING < PLISUB                                 '
'EXITING < PLICALL                                '
```

Finding unexpected storage overwrite errors in PL/I

During program run time, some storage might unexpectedly change its value and you want to find out when and where this happened. Consider the following example where the program changes more than the caller expects it to change.

```
2 FIELD1(2) CHAR(8);
2 FIELD2 CHAR(8);
 CTR = 3;          /* an invalid index value is set */
 FIELD1(CTR) = 'TOO MUCH';
```

Find the address of FIELD2 with the command:

```
DESCRIBE ATTRIBUTES FIELD2
```

Suppose the result is X'00521D42'. To set a breakpoint that watches for a change in storage values starting at that address for the next 8 bytes, issue the command:

```
AT CHANGE %STORAGE('00521D42'px,8)
```

When the program is run, z/OS Debugger halts if the value in this storage changes.

Halting before calling an undefined program in PL/I

Calling an undefined program or function is a severe error. To halt just before such a call is run, set this breakpoint:

```
AT CALL 0
```

When z/OS Debugger stops at this breakpoint, you can bypass the CALL by entering the GO BYPASS command. This allows you to continue your debug session without raising a condition.

Chapter 25. Debugging a C program in full-screen mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for C refer to the following C program.

"Example: sample C program for debugging" on page 221

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 33, "Debugging C and C++ programs," on page 293
"Halting when certain functions are called in C" on page 224
"Modifying the value of a C variable" on page 224
"Halting on a line in C only if a condition is true" on page 225
"Debugging C when only a few parts are compiled with TEST" on page 225
"Capturing C output to stdout" on page 226
"Calling a C function from z/OS Debugger" on page 226
"Displaying raw storage in C" on page 227
"Debugging a C DLL" on page 227
"Getting a function traceback in C" on page 227
"Tracing the run-time path for C code compiled with TEST" on page 227
"Finding unexpected storage overwrite errors in C" on page 228
"Finding uninitialized storage errors in C" on page 229
"Halting before calling a NULL C function" on page 229

Example: sample C program for debugging

The program below is used in various topics to demonstrate debugging tasks.

This program is a simple calculator that reads its input from a character buffer. If integers are read, they are pushed on a stack. If one of the operators (+ - *\) is read, the top two elements are popped off the stack, the operation is performed on them, and the result is pushed on the stack. The = operator writes out the value of the top element of the stack to a buffer.

CALC.H

```
/*----- FILE CALC.H ------------------------------------------------*/
/*                                                                  */
/* Header file for CALC.C PUSHPOP.C READTOKN.C                      */
/* a simple calculator                                              */
/*------------------------------------------------------------------*/
typedef enum toks {
  T_INTEGER,
  T_PLUS,
  T_TIMES,
  T_MINUS,
  T_DIVIDE,
  T_EQUALS,
  T_STOP
} Token;
Token read_token(char buf[]);
typedef struct int_link  {
  struct int_link * next;
  int i;
} IntLink;
typedef struct int_stack {
  IntLink * top;
} IntStack;
```

© Copyright IBM Corp. 1992, 2023

```
extern void  push(IntStack *, int);
extern int pop(IntStack *);
```

CALC.C

```
/*----- FILE CALC.C -------------------------------------------------*/
/*                                                                   */
/* A simple calculator that does operations on integers that         */
/* are pushed and popped on a stack                                  */
/*-------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
IntStack stack = { 0 };
main()
{
  Token tok;
  char word[100];
  char buf_out[100];
  int num, num2;
  for(;;)
  {
    tok=read_token(word);
    switch(tok)
    {
      case T_STOP:
        break;
      case T_INTEGER:
        num = atoi(word);
        push(&stack,num);     /* CALC1 statement */
        break;
      case T_PLUS:
        push(&stack, pop(&stack)+pop(&stack) );
        break;
      case T_MINUS:
        num = pop(&stack);
        push(&stack, num-pop(&stack));
        break;
      case T_TIMES:
        push(&stack, pop(&stack)*pop(&stack));
        break;
      case T_DIVIDE:
        num2 = pop(&stack);
        num = pop(&stack);
        push(&stack, num/num2);   /* CALC2 statement */
        break;
      case T_EQUALS:
        num = pop(&stack);
        sprintf(buf_out,"= %d ",num);
        push(&stack,num);
        break;
    }
    if (tok==T_STOP)
      break;
  }
  return 0;
}
```

PUSHPOP.C

```
/*----- FILE PUSHPOP.C ----------------------------------------------*/
/*                                                                   */
/* A push and pop function for a stack of integers                   */
/*-------------------------------------------------------------------*/
#include <stdlib.h>
#include "calc.h"
/*-------------------------------------------------------------------*/
/* input:  stk - stack of integers                                   */
/*         num - value to push on the stack                          */
/* action: get a link to hold the pushed value, push link on stack   */
/*                                                                   */
extern void push(IntStack * stk, int num)
{
  IntLink * ptr;
  ptr       = (IntLink *) malloc( sizeof(IntLink));  /* PUSHPOP1  */
  ptr->i    = num;                /* PUSHPOP2 statement */
  ptr->next = stk->top;
  stk->top  = ptr;
```

```
}
/*--------------------------------------------------------------------*/
/* return: int value popped from stack                                */
/* action: pops top element from stack and gets return value from it  */
/*--------------------------------------------------------------------*/
extern int pop(IntStack * stk)
{
  IntLink * ptr;
  int num;
  ptr      = stk->top;
  num      = ptr->i;
  stk->top = ptr->next;
  free(ptr);
  return num;
}
```

READTOKN.C

```
/*----- FILE READTOKN.C ----------------------------------------------*/
/*                                                                    */
/* A function to read input and tokenize it for a simple calculator   */
/*--------------------------------------------------------------------*/
#include <ctype.h>
#include <stdio.h>
#include "calc.h"
/*--------------------------------------------------------------------*/
/* action: get next input char, update index for next call            */
/* return: next input char                                            */
/*--------------------------------------------------------------------*/
static char nextchar(void)
{
/*--------------------------------------------------------------------*/
/* input  action:                                                     */
/*    2      push 2 on stack                                          */
/*    18     push 18                                                  */
/*    +      pop 2, pop 18, add, push result (20)                     */
/*    =      output value on the top of the stack (20)                */
/*    5      push 5                                                   */
/*    /      pop 5, pop 20, divide, push result (4)                   */
/*    =      output value on the top of the stack (4)                 */
/*--------------------------------------------------------------------*/
  char * buf_in  = "2 18 + = 5 / = ";
  static int index;     /* starts at 0 */
  char ret;
  ret = buf_in[index];
  ++index;
  return ret;
}
/*--------------------------------------------------------------------*/
/* output: buf - null terminated token                                */
/* return: token type                                                 */
/* action: reads chars through nextchar() and tokenizes them          */
/*--------------------------------------------------------------------*/
Token read_token(char buf[])
{
  int i;
  char c;
  /* skip leading white space */
  for( c=nextchar();
       isspace(c);
       c=nextchar())
    ;
  buf[0] = c;  /* get ready to return single char e.g."+" */
  buf[1] = 0;
  switch(c)
  {
    case '+' : return T_PLUS;
    case '-' : return T_MINUS;
    case '*' : return T_TIMES;
    case '/' : return T_DIVIDE;
    case '=' : return T_EQUALS;
    default:
      i = 0;
      while (isdigit(c)) {
        buf[i++] = c;
        c = nextchar();
      }
      buf[i] = 0;
      if (i==0)
        return T_STOP;
```

```
      else
        return T_INTEGER;
  }
}
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 25, “Debugging a C program in full-screen mode,” on page 221

Halting when certain functions are called in C

This topic describes how to halt just before or just after a routine is called by using the AT CALL and AT ENTRY commands. The “Example: sample C program for debugging” on page 221 is used to describe these commands.

To use the AT CALL command, you must compile the calling program with the TEST compiler option.

To halt just before `read_token` is called, enter the following command:

```
AT CALL read_token ;
```

To use the AT ENTRY command, you must compile the called program with the TEST compiler option.

To halt just after `read_token` is called, enter the following command:

```
AT ENTRY read_token ;
```

To halt just after `push` is called and only when *num* equals 16, enter the following command:

```
AT ENTRY push WHEN num=16;
```

Modifying the value of a C variable

To LIST the contents of a single variable, move the cursor to the variable name and press PF4 (LIST). The value is displayed in the Log window. This is equivalent to entering `LIST TITLED variable` on the command line.

“Example: sample C program for debugging” on page 221

Run the CALC program above to the statement labeled CALC1, move the cursor over *num* and press PF4 (LIST). The following appears in the Log window:

```
LIST ( num ) ;
num = 2
```

To modify the value of *num* to 22, type over the `num = 2` line with `num = 22`, press Enter to put it on the command line, and press Enter again to issue the command.

You can enter most C expressions on the command line.

Now step into the call to `push()` by pressing PF2 (STEP) and step until the statement labeled PUSHPOP2 is reached. To view the attributes of variable *ptr*, issue the z/OS Debugger command:

```
DESCRIBE ATTRIBUTES *ptr;
```

The result in the Log window is similar to the following:

```
ATTRIBUTES for * ptr
Its address is 0BB6E010 and its length is 8
  struct int_link
    struct int_link *next;
    int i;
```

You can use this action to browse structures and unions.

You can list all the values of the members of the structure pointed to by *ptr* with the command:

```
LIST *ptr ;
```

with results in the Log window appearing similar to the following:

```
LIST * ptr ;
(* ptr).next = 0x00000000
(* ptr).i = 0
```

You can change the value of a structure member by issuing the assignment as a command as in the following example:

```
(* ptr).i = 33 ;
```

Halting on a line in C only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but fails afterward because a specific condition is present. Setting a simple line breakpoint is an inefficient way to debug the program because you need to execute the GO command a thousand times to reach the specific condition. You can instruct z/OS Debugger to continue executing a program until a specific condition is present.

"Example: sample C program for debugging" on page 221

For example, in the `main` procedure of the program above, you want to stop at `T_DIVIDE` only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 40 { if(num2 != 0) GO; }
```

Line 40 is the statement labeled CALC2. The command causes z/OS Debugger to stop at line 40. If the value of num2 is not 0, the program continues. You can enter z/OS Debugger commands to change the value of num2 to a nonzero value.

Debugging C when only a few parts are compiled with TEST

"Example: sample C program for debugging" on page 221

Suppose you want to set a breakpoint at entry to the function `push()` in the file PUSHPOP.C. PUSHPOP.C has been compiled with TEST but the other files have not. z/OS Debugger comes up with an empty Source window. To display the compile units, enter the command:

```
LIST NAMES CUS
```

The `LIST NAMES CUS` command displays a list of all the compile units that are known to z/OS Debugger. Depending on the compiler you are using, or if `"USERID.MFISTART.C(PUSHPOP)"` is fetched later on by the application, this compile unit might not be known to z/OS Debugger. If it is displayed, enter:

```
SET QUALIFY CU "USERID.MFISTART.C(PUSHPOP)"
AT ENTRY push;
GO ;
```

or

```
AT ENTRY "USERID.MFISTART.C(PUSHPOP)":>push
GO;
```

If it is not displayed, set an appearance breakpoint as follows:

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" ;
GO ;
```

The only purpose for this appearance breakpoint is to gain control the first time a function in the PUSHPOP compile unit is run. When that happens, you can set breakpoints at entry to `push()`:

```
AT ENTRY push;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" AT ENTRY push; GO;
```

Capturing C output to stdout

To redirect `stdout` to the Log window, issue the following command:

```
SET INTERCEPT ON FILE stdout ;
```

With this SET command, you will capture not only `stdout` from your program, but also from interactive function calls. For example, you can interactively call `printf` on the command line to display a null-terminated string by entering:

```
printf(sptr);
```

You might find this easier than using `LIST STORAGE`.

Capturing C input to stdin

To redirect `stdin` input so that you can enter it from the command prompt, do the following steps

1. Enter the following command: `SET INTERCEPT ON FILE stdin ;`
2. When z/OS Debugger encounters a C statement such as `scanf`, the following message is displayed in the Log window:

   ```
   EQA1290I The program is waiting for input from stdin
   EQA1292I Use the INPUT command to enter up to a maximum of 1000
            characters for the intercepted variable-format file.
   ```

3. Enter the INPUT command to enter the input data.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

IBM z/OS Debugger Reference and Messages

Calling a C function from z/OS Debugger

You can start a library function (such as `strlen`) or one of the program functions interactively by calling it on the command line. The functions must comply with the following requirements:

- The functions cannot be in XPLINK applications.
- The functions must have debug information available.

"Example: sample C program for debugging" on page 221

Below, we call `push()` interactively to push one more value on the stack just before a value is popped off.

```
AT CALL pop  ;
GO ;
push(77);
GO ;
```

The calculator produces different results than before because of the additional value pushed on the stack.

Displaying raw storage in C

A `char *` variable *ptr* can point to a piece of storage containing printable characters. To display the first 20 characters enter:

```
LIST STORAGE(*ptr,20)
```

If the string is null terminated, you can also use an interactive function call on the command line, as in:

```
puts(ptr) ;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Displaying and modifying memory through the Memory window" on page 192

Debugging a C DLL

"Example: sample C program for debugging" on page 221

Build PUSHPOP.C as a DLL, exporting `push()` and `pop()`. Build CALC.C and READTOKN.C as the program that imports `push()` and `pop()` from the DLL named PUSHPOP. When the application CALC starts the DLL, PUSHPOP will not be known to z/OS Debugger. Use the `AT APPEARANCE` breakpoint to gain control in the DLL the first time code in that compile unit appears, as shown in the following example:

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" ;
GO ;
```

The only purpose of this appearance breakpoint is to gain control the first time a function in the PUSHPOP compile unit is run. When this happens, you can set breakpoints in PUSHPOP.

Getting a function traceback in C

Often when you get close to a programming error, you want to know how you got into that situation, and especially what the traceback of calling functions is. To get this information, issue the command:

```
LIST CALLS ;
```

"Example: sample C program for debugging" on page 221

For example, if you run the CALC example with the commands:

```
AT ENTRY read_token ;
GO ;
LIST CALLS ;
```

the Log window will contain something like:

```
At ENTRY in C function CALC ::> "USERID.MFISTART.C(READTOKN)" :> read_token.
From LINE 18 in C function CALC ::> "USERID.MFISTART.C(CALC)" :> main :> %BLOCK2.
```

which shows the traceback of callers.

Tracing the run-time path for C code compiled with TEST

To trace a program showing the entry and exit points without requiring any changes to the program, place the following z/OS Debugger commands in a file and USE them when z/OS Debugger initially displays your program. Assuming you have a data set `USERID.DTUSE(TRACE)` that contains the following z/OS Debugger commands:

```
int indent;
indent = 0;
SET INTERCEPT ON FILE stdout;
```

```
AT ENTRY * { \
  ++indent; \
  if (indent < 0) indent = 0; \
  printf("%*.s>%s\n", indent, " ", %block); \
  GO; \
}
AT EXIT * {\
  if (indent < 0) indent = 0; \
  printf("%*.s<%s\n", indent, " ", %block); \
  --indent; \
  GO; \
}
```

You can use this file as the source of commands to z/OS Debugger by entering the following command:

```
USE USERID.DTUSE(TRACE)
```

The trace of running the program listed below after executing the USE file will be displayed in the Log window.

```
int foo(int i, int j) {
  return i+j;
}
int main(void) {
  return foo(1,2);
}
```

The following trace in the Log window is displayed after running the sample program, with the USE file as a source of input for z/OS Debugger commands:

```
>main
 >foo
 <foo
<main
```

If you do not want to create the USE file, you can enter the commands through the command line, and the same effect is achieved.

Finding unexpected storage overwrite errors in C

During program run time, some storage might unexpectedly change its value and you want to find out when and where this happens. Consider this example where function `set_i` changes more than the caller expects it to change.

```
struct s { int i; int j;};
struct s a = { 0, 0 };

/* function sets only field i */
void set_i(struct s * p, int k)
{
  p->i = k;
  p->j = k;   /* error, it unexpectedly sets field j also */
}
main() {
  set_i(&a,123);
}
```

Find the address of a with the command

```
LIST &(a.j) ;
```

Suppose the result is `0x7042A04`. To set a breakpoint that watches for a change in storage values starting at that address for the next 4 bytes, issue the command:

```
AT CHANGE %STORAGE(0x7042A04,4)
```

When the program is run, z/OS Debugger will halt if the value in this storage changes.

Finding uninitialized storage errors in C

To help find your uninitialized storage errors, run your program with the Language Environment TEST run-time and STORAGE options. In the following example:

```
TEST STORAGE(FD,FB,F9)
```

the first subparameter of STORAGE is the fill byte for storage allocated from the heap. For example, storage allocated through `malloc()` is filled with the byte 0xFD. If you see this byte repeated through storage, it is likely uninitialized heap storage.

The second subparameter of STORAGE is the fill byte for storage allocated from the heap but then freed. For example, storage freed by calling `free()` might be filled with the byte 0xFB. If you see this byte repeated through storage, it is likely storage that was allocated on the heap, but has been freed.

The third subparameter of STORAGE is the fill byte for auto storage variables in a new stack frame. If you see this byte repeated through storage, it is likely uninitialized auto storage.

The values chosen in the example are odd and large, to maximize early problem detection. For example, if you attempt to branch to an odd address you will get an exception immediately.

"Example: sample C program for debugging" on page 221

As an example of uninitialized heap storage, run program CALC with the STORAGE run-time option as `STORAGE(FD,FB,F9)` to the line labeled PUSHPOP2 and issue the command:

```
LIST *ptr ;
```

You will see the byte fill for uninitialized heap storage as the following example shows:

```
LIST * ptr ;
(* ptr).next = 0xFDFDFDFD
(* ptr).i = -33686019
```

Halting before calling a NULL C function

Calling an undefined function or calling a function through a function pointer that points to NULL is a severe error. To halt just before such a call is run, set this breakpoint:

```
AT CALL 0
```

When z/OS Debugger stops at this breakpoint, you can bypass the CALL by entering the GO BYPASS command. This allows you to continue your debug session without raising a condition.

Chapter 26. Debugging a C++ program in full-screen mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for C++ refer to the following C++ program.

"Example: sample C++ program for debugging" on page 231

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 33, "Debugging C and C++ programs," on page 293
"Halting when certain functions are called in C++" on page 234
"Modifying the value of a C++ variable" on page 235
"Halting on a line in C++ only if a condition is true" on page 236
"Viewing and modifying data members of the this pointer in C++" on page 236
"Debugging C++ when only a few parts are compiled with TEST" on page 236
"Capturing C++ output to stdout" on page 237
"Calling a C++ function from z/OS Debugger" on page 238
"Displaying raw storage in C++" on page 238
"Debugging a C++ DLL" on page 238
"Getting a function traceback in C++" on page 238
"Tracing the run-time path for C++ code compiled with TEST" on page 239
"Finding unexpected storage overwrite errors in C++" on page 240
"Finding uninitialized storage errors in C++" on page 240
"Halting before calling a NULL C++ function" on page 241

Example: sample C++ program for debugging

The program below is used in various topics to demonstrate debugging tasks.

This program is a simple calculator that reads its input from a character buffer. If integers are read, they are pushed on a stack. If one of the operators (+ - */) is read, the top two elements are popped off the stack, the operation is performed on them, and the result is pushed on the stack. The = operator writes out the value of the top element of the stack to a buffer.

CALC.HPP

```
/*----- FILE CALC.HPP -------------------------------------------------*/
/*                                                                     */
/* Header file for CALC.CPP PUSHPOP.CPP READTOKN.CPP                   */
/* a simple calculator                                                 */
/*---------------------------------------------------------------------*/
typedef enum toks {
  T_INTEGER,
  T_PLUS,
  T_TIMES,
  T_MINUS,
  T_DIVIDE,
  T_EQUALS,
  T_STOP
} Token;
extern "C" Token read_token(char buf[]);
class IntLink {
  private:
    int i;
    IntLink * next;
  public:
    IntLink();
```

© Copyright IBM Corp. 1992, 2023

```
    ~IntLink();
    int get_i();
    void set_i(int j);
    IntLink * get_next();
    void set_next(IntLink * d);
};
class IntStack {
  private:
    IntLink * top;
  public:
    IntStack();
    ~IntStack();
    void push(int);
    int pop();
};
```

CALC.CPP

```
/*----- FILE CALC.CPP ------------------------------------------------*/
/*                                                                    */
/* A simple calculator that does operations on integers that          */
/* are pushed and popped on a stack                                   */
/*--------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include "calc.hpp"
IntStack stack;
int main()
{
  Token tok;
  char word[100];
  char buf_out[100];
  int num, num2;
  for(;;)
  {
    tok=read_token(word);
    switch(tok)
    {
      case T_STOP:
        break;
      case T_INTEGER:
        num = atoi(word);
        stack.push(num);      /* CALC1 statement */
        break;
      case T_PLUS:
        stack.push(stack.pop()+stack.pop());
        break;
      case T_MINUS:
        num = stack.pop();
        stack.push(num-stack.pop());
        break;
      case T_TIMES:
        stack.push(stack.pop()*stack.pop() );
        break;
      case T_DIVIDE:
        num2 = stack.pop();
        num = stack.pop();
        stack.push(num/num2);      /* CALC2 statement */
        break;
      case T_EQUALS:
        num = stack.pop();
        sprintf(buf_out,"= %d ",num);
        stack.push(num);
        break;
    }
    if (tok==T_STOP)
      break;
  }
  return 0;
}
```

PUSHPOP.CPP

```
/*----- FILE: PUSHPOP.CPP --------------------------------------------*/
/*                                                                    */
/* Push and pop functions for a stack of integers                     */
/*--------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
```

```
#include "calc.hpp"
/*--------------------------------------------------------------------*/
/* input:  num - value to push on the stack                           */
/* action: get a link to hold the pushed value, push link on stack    */
/*--------------------------------------------------------------------*/
void IntStack::push(int num) {
  IntLink * ptr;
  ptr = new IntLink;
  ptr->set_i(num);
  ptr->set_next(top);
  top = ptr;
}
/*--------------------------------------------------------------------*/
/* return: int value popped from stack (0 if stack is empty)          */
/* action: pops top element from stack and get return value from it   */
/*--------------------------------------------------------------------*/
int IntStack::pop() {
  IntLink * ptr;
  int num;
  ptr = top;
  num = ptr->get_i();
  top = ptr->get_next();
  delete ptr;
  return num;
}
IntStack::IntStack() {
  top = 0;
}
IntStack::~IntStack() {
  while(top)
    pop();
}
IntLink::IntLink() {  /* constructor leaves elements unassigned */
}
IntLink::~IntLink() {
}
void IntLink::set_i(int j) {
  i = j;
}
int IntLink::get_i() {
  return i;
}
void IntLink::set_next(IntLink * p) {
  next = p;
}
IntLink * IntLink::get_next() {
  return next;
}
```

READTOKN.CPP

```
/*----- FILE READTOKN.CPP ----------------------------------------*/
/*                                                                */
/* A function to read input and tokenize it for a simple calculator   */
/*--------------------------------------------------------------------*/
#include <ctype.h>
#include <stdio.h>
#include "calc.hpp"
/*--------------------------------------------------------------------*/
/* action: get next input char, update index for next call            */
/* return: next input char                                            */
/*--------------------------------------------------------------------*/
static char nextchar(void)
{
  /*    input  action
   *    -----  ------
   *    2      push 2 on stack
   *    18     push 18
   *    +      pop 2, pop 18, add, push result (20)
   *    =      output value on the top of the stack (20)
   *    5      push 5
   *    /      pop 5, pop 20, divide, push result (4)
   *    =      output value on the top of the stack (4)
   */
  char * buf_in  = "2 18 + = 5 / = ";
  static int index;     /* starts at 0 */
  char ret;
  ret = buf_in[index];
  ++index;
  return ret;
```

```
}
/*--------------------------------------------------------------------*/
/* output: buf - null terminated token                                */
/* return: token type                                                 */
/* action: reads chars through nextchar() and tokenizes them          */
/*--------------------------------------------------------------------*/
extern "C"
Token read_token(char buf[])
{
  int i;
  char c;
  /* skip leading white space */
  for( c=nextchar();
       isspace(c);
       c=nextchar())
    ;
  buf[0] = c;  /* get ready to return single char e.g. "+" */
  buf[1] = 0;
  switch(c)
  {
    case '+' : return T_PLUS;
    case '-' : return T_MINUS;
    case '*' : return T_TIMES;
    case '/' : return T_DIVIDE;
    case '=' : return T_EQUALS;
    default:
      i = 0;
      while (isdigit(c)) {
        buf[i++] = c;
        c = nextchar();
      }
      buf[i] = 0;
      if (i==0)
        return T_STOP;
      else
        return T_INTEGER;
  }
}
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 26, “Debugging a C++ program in full-screen mode,” on page 231

Halting when certain functions are called in C++

This topic describes how to halt just before or just after a routine is called by using the AT CALL or AT ENTRY commands. The “Example: sample C++ program for debugging” on page 231 is used to describe these commands. Before you use either of these commands, you must do the following tasks:

- To use the AT ENTRY command, you must compile the called program with the TEST compiler option.
- To use the AT CALL command, you must compile the calling program with the TEST compiler option.

When you use either of these commands, include the C++ signature along with the function name.

To facilitate entering the breakpoint, you can display PUSHPOP.CPP in the Source window by typing over the name of the file on the top line of the Source window. This makes PUSHPOP.CPP your currently qualified program. You can then enter the following command:

```
LIST NAMES
```

z/OS Debugger displays the names of all the blocks and variables for the currently qualified program. z/OS Debugger displays information similar to the following example in the Log window:

```
There are no session names.
The following names are known in block CALC ::> "USERID.MFISTART.CPP(PUSHPOP)"
IntStack::~IntStack()
IntStack::IntStack()
IntLink::get_i()
IntLink::get_next()
IntLink::~IntLink()
IntLink::set_i(int)
```

```
IntLink::set_next(IntLink*)
IntLink::IntLink()
```

Now you can save some keystrokes by inserting the command next to the block name.

To halt just before `IntStack::push(int)` is called, insert `AT CALL` next to the function signature and, by pressing Enter, the entire command is placed on the command line. Now, with `AT CALL IntStack::push(int)` on the command line, you can enter the following command:

```
AT CALL IntStack::push(int)
```

To halt just after `IntStack::push(int)` is called, enter the following command, which is the same way as the `AT CALL` command:

```
AT ENTRY IntStack::push(int) ;
```

To halt just after `IntStack::push(int)` is called and only when *num* equals 16, enter the following command:

```
AT ENTRY IntStack::push(int) WHEN num=16;
```

Modifying the value of a C++ variable

To list the contents of a single variable, move the cursor to the variable name and press PF4 (`LIST`). The value is displayed in the Log window. This is equivalent to entering `LIST TITLED` *`variable`* on the command line.

"Example: sample C++ program for debugging" on page 231

Run the CALC program and step into the first call of function `IntStack::push(int)` until just after the `IntLink` has been allocated. Enter the z/OS Debugger command:

```
LIST TITLED num
```

z/OS Debugger displays the following in the Log window:

```
LIST TITLED num;
num = 2
```

To modify the value of num to 22, type over the `num = 2` line with `num = 22`, press Enter to put it on the command line, and press Enter again to issue the command.

You can enter most C++ expressions on the command line.

To view the attributes of variable *ptr* in `IntStack::push(int)`, issue the z/OS Debugger command:

```
DESCRIBE ATTRIBUTES *ptr;
```

The result in the Log window is:

```
ATTRIBUTES for * ptr
Its address is 0BA25EB8 and its length is 8
  class IntLink
    signed int i
    struct IntLink *next
```

So for most classes, structures, and unions, this can act as a browser.

You can list all the values of the data members of the class object pointed to by `ptr` with the command:

```
LIST *ptr ;
```

with results in the Log window similar to:

```
LIST * ptr ; * ptr.i = 0 * ptr.next = 0x00000000
```

You can change the value of data member of a class object by issuing the assignment as a command, as in this example:

```
(* ptr).i = 33 ;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using C and C++ variables with z/OS Debugger" on page 294

Halting on a line in C++ only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but fails under certain conditions. You don't want to set a simple line breakpoint because you will have to keep entering GO.

"Example: sample C++ program for debugging" on page 231

For example, in `main` you want to stop in `T_DIVIDE` only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 40 { if(num2 != 0) GO; }
```

Line 40 is the statement labeled CALC2. The command causes z/OS Debugger to stop at line 40. If the value of `num` is not 0, the program will continue. z/OS Debugger stops on line 40 only if `num2` is 0.

Viewing and modifying data members of the this pointer in C++

If you step into a class method, for example, one for class `IntLink`, the command:

```
LIST TITLED ;
```

responds with a list that includes `this`. With the command:

```
DESCRIBE ATTRIBUTES *this ;
```

you will see the types of the data elements pointed to by the `this` pointer. With the command:

```
LIST *this ;
```

you will list the data member of the object pointed to and see something like:

```
 LIST * this ;
(* this).i = 4
(* this).next =  0x0
```

in the Log window. To modify element `i`, enter either the command:

```
i = 2001;
```

or, if you have ambiguity (for example, you also have an auto variable named `i`), enter:

```
(* this).i = 2001 ;
```

Debugging C++ when only a few parts are compiled with TEST

"Example: sample C++ program for debugging" on page 231

Suppose you want to set a breakpoint at entry to function `IntStack::push(int)` in the file PUSHPOP.CPP. PUSHPOP.CPP has been compiled with TEST but the other files have not. z/OS Debugger comes up with an empty Source window. To display the compile units, enter the command:

```
LIST NAMES CUS
```

The LIST NAMES CUS command displays a list of all the compile units that are known to z/OS Debugger.

Depending on the compiler you are using, or if USERID.MFISTART.CPP(PUSHPOP) is fetched later on by the application, this compile unit might or might not be known to z/OS Debugger, and the PDS member PUSHPOP might or might not be displayed. If it is displayed, enter:

```
SET QUALIFY CU "USERID.MFISTART.CPP(PUSHPOP)"
AT ENTRY IntStack::push(int) ;
GO ;
```

or

```
AT ENTRY "USERID.MFISTART.CPP(PUSHPOP)":>push
GO
```

If it is not displayed, you need to set an appearance breakpoint as follows:

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" ;
GO ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" AT ENTRY push; GO;
```

The only purpose of this appearance breakpoint is to gain control the first time a function in the PUSHPOP compile unit is run. When that happens you can, for example, set a breakpoint at entry to IntStack::push(int) as follows:

```
AT ENTRY IntStack::push(int) ;
```

Capturing C++ output to stdout

To redirect stdout to the Log window, issue the following command:

```
SET INTERCEPT ON FILE stdout ;
```

With this SET command, you will not only capture stdout from your program, but also from interactive function calls. For example, you can interactively use cout on the command line to display a null terminated string by entering:

```
cout << sptr ;
```

You might find this easier than using LIST STORAGE.

For CICS only, SET INTERCEPT is not supported.

Capturing C++ input to stdin

To redirect stdin input so that you can enter it from the command prompt, do the following steps

1. Enter the following command: SET INTERCEPT ON FILE stdin ;
2. When z/OS Debugger encounters a C++ statement such as scanf, the following message is displayed in the Log window:

   ```
   EQA1290I The program is waiting for input from stdin
   EQA1292I Use the INPUT command to enter up to a maximum of 1000
            characters for the intercepted variable-format file.
   ```

3. Enter the INPUT command to enter the input data.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

Calling a C++ function from z/OS Debugger

You can start a library function (such as `strlen`) or one of the programs functions interactively by calling it on the command line. You can also start C linkage functions such as `read_token`. However, you cannot call C++ linkage functions interactively. The functions must comply with the following requirements:

- The functions cannot be in XPLINK applications.
- The functions must have debug information available.

"Example: sample C++ program for debugging" on page 231

In the example below, we call `read_token` interactively.

```
AT CALL read_token;
GO;
read_token(word);
```

The calculator produces different results than before because of the additional token removed from input.

Displaying raw storage in C++

A `char *` variable `ptr` can point to a piece of storage that contains printable characters. To display the first 20 characters, enter;

```
LIST STORAGE(*ptr,20)
```

If the string is null terminated, you can also use an interactive function call on the command line as shown in this example:

```
puts(ptr) ;
```

You can also display storage based on offset. For example, to display 10 bytes at an offset of 2 from location 20CD0, use the following command:

```
LIST STORAGE(0x20CD0,2,10);
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Displaying and modifying memory through the Memory window" on page 192

Debugging a C++ DLL

"Example: sample C++ program for debugging" on page 231

Build PUSHPOP.CPP as a DLL, exporting IntStack::push(int) and IntStack::pop(). Build CALC.CPP and READTOKN.CPP as the program that imports IntStack::push(int) and IntStack::pop() from the DLL named PUSHPOP. When the application CALC starts, the DLL PUSHPOP is not known to z/OS Debugger. Use the `AT APPEARANCE` breakpoint, as shown in the following example, to gain control in the DLL the first time code in that compile unit appears.

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" ;
GO ;
```

The only purpose of this appearance breakpoint is to gain control the first time a function in the PUSHPOP compile unit is run. When this happens, you can set breakpoints in PUSHPOP.

Getting a function traceback in C++

Often when you get close to a programming error, you want to know how you got into that situation, especially what the traceback of calling functions is. To get this information, issue the command:

```
LIST CALLS ;
```

For example, if you run the CALC example with the following commands:

```
AT ENTRY read_token ;
GO ;
LIST CALLS ;
```

the Log window contains something like:

```
At ENTRY in C function "USERID.MFISTART.CPP(READTOKN)" :> read_token.
From LINE 18 in C function "USERID.MFISTART.CPP(CALC)" :> main :> %BLOCK2.
```

which shows the traceback of callers.

Tracing the run-time path for C++ code compiled with TEST

To trace a program showing the entry and exit of that program without requiring any changes to it, place the following z/OS Debugger commands, shown in the example below, in a file and USE them when z/OS Debugger initially displays your program. Assume you have a data set that contains USERID.DTUSE(TRACE) and contains the following z/OS Debugger commands:

```
int indent;
indent = 0;
SET INTERCEPT ON FILE stdout;
AT ENTRY * { \
  ++indent; \
  if (indent < 0) indent = 0; \
  printf("%*.s>%s\n", indent, " ", %block); \
  GO; \
}
AT EXIT * {\
  if (indent < 0) indent = 0; \
  printf("%*.s<%s\n", indent, " ", %block); \
  --indent; \
  GO; \
}
```

You can use this file as the source of commands to z/OS Debugger by entering the following command:

```
USE USERID.DTUSE(TRACE)
```

The trace of running the program listed below after executing the USE file is displayed in the Log window:

```
int foo(int i, int j) {
  return i+j;
}
int main(void) {
  return foo(1,2);
}
```

The following trace in the Log window is displayed after running the sample program, using the USE file as a source of input for z/OS Debugger commands:

```
>main
 >foo(int,int)
 <foo(int,int)
<main
```

If you do not want to create the USE file, you can enter the commands through the command line, and the same effect will be achieved.

Finding unexpected storage overwrite errors in C++

During program run time, some storage might unexpectedly change its value and you would like to find out when and where this happened. Consider this simple example where function `set_i` changes more than the caller expects it to change.

```
struct s { int i; int j;};
struct s a = { 0, 0 };

/* function sets only field i */
void set_i(struct s * p, int k)
{
  p->i = k;
  p->j = k;   /* error, it unexpectedly sets field j also */
}
main() {
  set_i(&a,123);
}
```

Find the address of a with the command:

```
LIST &(a.j) ;
```

Suppose the result is `0x7042A04`. To set a breakpoint that watches for a change in storage values, starting at that address for the next 4 bytes, issue the command:

```
AT CHANGE %STORAGE(0x7042A04,4)
```

When the program is run, z/OS Debugger will halt if the value in this storage changes.

Finding uninitialized storage errors in C++

To help find your uninitialized storage errors, run your program with the Language Environment TEST run-time and STORAGE options. In the following example:

```
TEST STORAGE(FD,FB,F9)
```

the first subparameter of STORAGE is the fill byte for storage allocated from the heap. For example, storage allocated through operator new is filled with the byte `0xFD`. If you see this byte repeated throughout storage, it is likely uninitialized heap storage.

The second subparameter of STORAGE is the fill byte for storage allocated from the heap but then freed. For example, storage freed by the operator `delete` might be filled with the byte `0xFB`. If you see this byte repeated throughout storage, it is likely storage that was allocated on the heap, but has been freed.

The third subparameter of STORAGE is the fill byte for auto storage variables in a new stack frame. If you see this byte repeated throughout storage, you probably have uninitialized auto storage.

The values chosen in the example are odd and large, to maximize early problem detection. For example, if you attempt to branch to an odd address, you will get an exception immediately.

As an example of uninitialized heap storage, run program CALC, with the STORAGE run-time option as `STORAGE(FD,FB,F9)`, to the line labeled PUSHPOP2 and issue the command:

```
LIST *ptr ;
```

You will see the byte fill for uninitialized heap storage as the following example shows:

```
 LIST * ptr ;
(* ptr).next = 0xFDFDFDFD
(* ptr).i = -33686019
```

Refer to the following topics for more information related to the material discussed in this topic.

Related references

z/OS Language Environment Programming Guide

Halting before calling a NULL C++ function

Calling an undefined function or calling a function through a function pointer that points to NULL is a severe error. To halt just before such a call is run, set this breakpoint:

```
AT CALL 0
```

When z/OS Debugger stops at this breakpoint, you can bypass the call by entering the GO BYPASS command. This command allows you to continue your debug session without raising a condition.

Chapter 27. Debugging an assembler program in full-screen mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The descriptions of basic debugging tasks for assembler refer to the following assembler program.

"Example: sample assembler program for debugging" on page 243

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 34, "Debugging an assembler program," on page 315
"Defining a compilation unit as assembler and loading debug data" on page 245
"Deferred LDDs" on page 246
"Halting when certain assembler routines are called" on page 248
"Displaying and modifying the value of assembler variables or storage" on page 248
"Halting on a line in assembler only if a condition is true" on page 249
"Getting an assembler routine traceback" on page 249
"Finding unexpected storage overwrite errors in assembler" on page 250

Example: sample assembler program for debugging

The program below is used in various topics to demonstrate debugging tasks.

To run this sample program, do the following steps:

1. Verify that the debug file for this assembler program is located in the SUBXMP and DISPARM members of the *yourid*.EQALANGX data set.
2. Start z/OS Debugger.
3. To load the information in the debug file, enter the following commands:

```
LDD (SUBXMP,DISPARM)
```

This program is a small example of an assembler main routine (SUBXMP) that calls an assembler subroutine (DISPARM).

Load module: XMPLOAD

SUBXMP.ASM

```
*****************************************************************
*                                                               *
*  NAME: SUBXMP                                                *
*                                                               *
*        A simple main assembler routine that brings up         *
*        Language Environment, calls a subroutine, and          *
*        returns with a return code of 0.                       *
*                                                               *
*****************************************************************
SUBXMP   CEEENTRY PPA=XMPPPA,AUTO=WORKSIZE
         USING WORKAREA,R13
* Invoke CEEMOUT to issue the greeting message
         CALL  CEEMOUT,(HELLOMSG,DEST,FBCODE),VL,MF=(E,CALLMOUT)
* No plist to DISPARM, so zero R1. Then call it.
         SLR   R0,R0
         ST    R0,COUNTER
         LA    R0,HELLOMSG
         SR    R01,R01 ssue a message
         CALL  DISPARM                CALL1
* Invoke CEEMOUT to issue the farewell message
```

© Copyright IBM Corp. 1992, 2023

```
         CALL  CEEMOUT,(BYEMSG,DEST,FBCODE),VL,MF=(E,CALLMOUT)
* Terminate Language Environment and return to the caller
         CEETERM  RC=0

*        CONSTANTS
HELLOMSG DC    Y(HELLOEND-HELLOSTR)
HELLOSTR DC    C'Hello from the sub example.'
HELLOEND EQU   *

BYEMSG   DC    Y(BYEEND-BYESTART)
BYESTART DC    C'Terminating the sub example.'
BYEEND   EQU   *
DEST     DC    F'2'               Destination is the LE message file
COUNTER  DC    F'-1'

XMPPPA   CEEPPA  ,                Constants describing the code block
*        The Workarea and DSA
WORKAREA DSECT
         ORG   *+CEEDSASZ         Leave space for the DSA fixed part
CALLMOUT CALL  ,(,,),VL,MF=L      3-argument parameter list
FBCODE   DS    3F                 Space for a 12-byte feedback code
         DS    0D
WORKSIZE EQU   *-WORKAREA
         PRINT NOGEN
         CEEDSA  ,                Mapping of the dynamic save area
         CEECAA  ,                Mapping of the common anchor area
R0       EQU   0
R01      EQU   1
R13      EQU   13
         END   SUBXMP             Nominate SUBXMP as the entry point
```

DISPARM.ASM

```
*************************************************************
*                                                           *
*  NAME:  DISPARM                                           *
*                                                           *
*        Shows an assembler subroutine that displays inbound *
*        parameters and returns.                            *
*                                                           *
*************************************************************
DISPARM  CEEENTRY PPA=PARMPPA,AUTO=WORKSIZE,MAIN=NO
         USING WORKAREA,R13
* Invoke CEE3PRM to retrieve the command parameters for us
         SLR   R0,R0
         ST    R0,COUNTER
         CALL  CEE3PRM,(CHARPARM,FBCODE),VL,MF=(E,CALL3PRM)      CALL2
* Check the feedback code from CEE3PRM to see if everything worked.
         CLC   FBCODE(8),CEE000
         BE    GOT_PARM
* Invoke CEEMOUT to issue the error message for us
         CALL  CEEMOUT,(BADFBC,DEST,FBCODE),VL,MF=(E,CALLMOUT)
         B     GO_HOME                Time to go....
GOT_PARM DS    0H
* See if the parm string is blank.
         LA    R1,1
SAVECTR  ST    R1,COUNTER
         CL    R1,=F'5'                  BUMPCTR
         BH    LOOPEND
         LA    R1,1(,R1)
         B     SAVECTR
LOOPEND  DS    0H
         CLC   CHARPARM(80),=CL80' '   Is the parm empty?
         BNE   DISPLAY_PARM            No. Print it out.
* Invoke CEEMOUT to issue the error message for us
         CALL  CEEMOUT,(NOPARM,DEST,FBCODE),VL,MF=(E,CALLMOUT)
         B     GO_TEST                 Time to go....

DISPLAY_PARM   DS  0H
* Set up the plist to CEEMOUT to display the parm.
         LA    R0,2
         ST    R0,COUNTER
         LA    R02,80        Get the size of the string
         STH   R02,BUFFSIZE  Save it for the len-prefixed string
* Invoke CEEMOUT to display the parm string for us
         CALL  CEEMOUT,(BUFFSIZE,DEST,FBCODE),VL,MF=(E,CALLMOUT)
*        AMODE Testing
GO_TEST  DS  0H
         L     R15,INAMODE24@
         BSM   R14,R15
```

```
InAMode24 Equ  *
          LA    R1,DEST
          O     R1,=X'FF000000'
          L     R15,0(,R1)
          LA    R15,2(,R15)
          ST    R15,0(,R1)
          L     R15,INAMODE31@
          BSM   R14,R15
InAMode31 Equ  *
* Return to the caller
GO_HOME   DS    0H
          LA    R0,3
          ST    R0,COUNTER
          CEETERM  RC=0

*         CONSTANTS
DEST      DC    F'2'               Destination is the LE message file
CEE000    DS    3F'0'              Success feedback code
InAMode24@ DC   A(InAMode24)
InAMode31@ DC   A(InAMode31+X'80000000')
BADFBC    DC    Y(BADFBEND-BADFBSTR)
BADFBSTR  DC    C'Feedback code from CEE3PRM was nonzero.'
BADFBEND  EQU   *
NOPARM    DC    Y(NOPRMEND-NOPRMSTR)
NOPRMSTR  DC    C'No user parm was passed to the application.'
NOPRMEND  EQU   *
PARMPPA   CEEPPA ,                 Constants describing the code block
* ================================================================
WORKAREA  DSECT
          ORG   *+CEEDSASZ         Leave space for the DSA fixed part
CALL3PRM  CALL  ,(,),VL,MF=L       2-argument parameter list
CALLMOUT  CALL  ,(,,),VL,MF=L      3-argument parameter list
FBCODE    DS    3F                 Space for a 12-byte feedback code
COUNTER   DS    F
BUFFSIZE  DS    H                  Halfword prefix for following string
CHARPARM  DS    CL255              80-byte buffer
          DS    0D
WORKSIZE  EQU   *-WORKAREA
          PRINT NOGEN
          CEEDSA  ,                Mapping of the dynamic save area
          CEECAA  ,                Mapping of the common anchor area
MYDATA    DSECT   ,
MYF       DS      F
R0        EQU   0
R1        EQU   1
R2        EQU   2
R3        EQU   3
R4        EQU   4
R5        EQU   5
R6        EQU   6
R7        EQU   7
R8        EQU   8
R9        EQU   9
R10       EQU   10
R11       EQU   11
R12       EQU   12
R13       EQU   13
R14       EQU   14
R15       EQU   15
R02       EQU   2
          END
```

Defining a compilation unit as assembler and loading debug data

Before you can debug an assembler program, you must define the compilation unit (CU) as an assembler CU and load the debug data for the CU. This can only be done for a CU that is currently known to z/OS Debugger as a disassembly CU.

You use the LOADDEBUGDATA command (abbreviated as LDD) to define a disassembly CU as an assembler CU and to cause the debug data for this CU to be loaded. When you run the LDD command, you can specify either a single CU name or a list of CU names enclosed in parenthesis. Each of the names specified must be either:

- the name of a disassembly CU that is currently known to z/OS Debugger
- a name that does not match the name of a CU currently known to z/OS Debugger

When the CU name is currently known to z/OS Debugger, the CU is immediately marked as an assembler CU and an attempt is made to load the debug data as follows:

- If your assembler debug data is in a partitioned data set where the high-level qualifier is the current user ID, the low-level qualifier is EQALANGX, and the member name is the same as the name of the CU that you want to debug no other action is necessary
- If your assembler debug data is in a different partitioned data set than *userid*.EQALANGX but the member name is the same as the name of the CU that you want to debug, enter the following command before or after you enter the LDD command: `SET DEFAULT LISTINGS`
- If your assembler debug data is in a sequential data set or is a member of a partitioned data set but the member name is different from the CU name, enter the following command before or after the LDD: `SET SOURCE`

When the CU name specified on the LDD command is not currently known to z/OS Debugger, a message is issued and the LDD command is deferred until a CU by that name becomes known (appears). At that time, the CU is automatically created as an assembler CU and an attempt is made to load the debug data using the default data set name or the current `SET DEFAULT LISTINGS` specification.

After you have entered an LDD command for a CU, you cannot view the CU as a disassembly CU.

If z/OS Debugger cannot find the associated assembler debug data after you have entered an LDD command, the CU is an assembler CU rather than a disassembly CU. You cannot enter another LDD command for this CU. However, you can enter a `SET DEFAULT LISTING` command or a `SET SOURCE` command to cause the associated debug data to be loaded from a different data set.

Deferred LDDs

As described in the previous section, you can use the LDD command to identify a CU as an assembler CU before the CU has become known to z/OS Debugger. This is known as a deferred LDD. In this case, whenever the CU appears, it is immediately marked as an assembler CU and an attempt is made to load the debug data from the default data set name or from the data set currently specified by SET DEFAULT LISTINGS.

If the debug data cannot be found in this way, you must using the SET SOURCE or SET DEFAULT LISTINGS command after the CU appears to cause the debug data to be loaded from the correct data set. You can do this using a command such as:

```
AT APPEARANCE mycu SET SOURCE (mycu) hlq.qual1.dsn
```

Alternatively, you might wait until you have stopped for some other reason after "mycu" has appeared and then use the SET SOURCE or SET DEFAULT LISTING commands to direct z/OS Debugger to the proper data set.

Re-appearance of an assembler CU

If a CU from which valid assembler debug data has been loaded goes away and then reappears (e.g., the load module is deleted and then reloaded), the CU is immediately marked as an assembler CU and the debug data is reloaded from the data set from which it was successfully loaded originally.

You do not need to (and cannot) issue another LDD for that CU because it is already known as an assembler CU and the debug data has already been loaded.

Multiple compilation units in a single assembly

z/OS Debugger treats each assembler CSECT as a separate compilation unit (CU). If your assembler source contains more than one CSECT, then the EQALANGX file that you create will contain debug information for all the CSECTs.

In most cases, all of the CSECTs in the assembly will be present in the load module or program object. However, in some cases, one or more of the assemblies might not be present or might be replaced by

other CSECTs of the same name. There are, therefore, two ways of loading the debug data for assemblies containing multiple CSECTs:

- When SET LDD ALL is in effect, the debug data for all CSECTs (CUs) in the assembly is loaded as the result of a single LOADDEBUGDATA (LDD) command.
- When SET LDD SINGLE is in effect, a separate LDD command must be issued for each CSECT (CU). This form must be used when one or more of the CSECTs in the assembly are not present in the load module or program object or when one or more of the CSECTs have been replaced by other CSECTs of the same name.

The following sections use an example assembly that generates two CSECTs: MYPROG and MYPROGA. The debug information for both of these CSECTs is in the data set `yourid.EQALANGX(MYPROG)`.

Loading debug data from multiple CSECTs in a single assembly using one LDD command

If SET LDD ALL is in effect, follow the process described in this section. This process is the easiest way to load debug data for assemblies containing multiple CSECTs when all of the CSECTs are present in the load module or program object.

When you enter the command LDD MYPROG, z/OS Debugger finds and loads the debug data for both MYPROG and MYPROGA. After the debug data is loaded, z/OS Debugger uses the debug data to create two CUs, one for MYPROG and another for MYPROGA.

Loading debug data from multiple CSECTs in a single assembly using separate LDD commands

If SET LDD SINGLE is in effect, follow the process described in this section.

When you enter the command `LDD MYPROG`, z/OS Debugger finds and loads the debug information for both MYPROG and MYPROGA. However, because you specified only MYPROG on the LDD command and SET LDD SINGLE is in effect, z/OS Debugger uses only the debug information for MYPROG. Then, if you enter the command `LDD MYPROGA`, z/OS Debugger does the following steps:

1. If you entered a `SET SOURCE` command before entering the `LDD MYPROG` command, z/OS Debugger loads the debug data from the data set that you specified with the `SET SOURCE` command.
2. If you did not enter the `SET SOURCE` command or if z/OS Debugger did not find debug information in step 1, z/OS Debugger searches through all previously loaded debug information. If z/OS Debugger finds a name and CSECT length that matches the name and CSECT length of MYPROGA, z/OS Debugger uses this debug information.

Debugging multiple CSECTs in a single assembly after the debug data is loaded

After you have loaded the debug data for both of the CSECTs in the assembly, you can begin debugging either of the compile units. Although the contents of both CSECTs appear in the source listing, you can only set breakpoints in the compile unit to which you are currently qualified.

When you look at the source listing, all lines contained in a CSECT to which you are not currently qualified have an asterisk immediately before the offset field and following the statement number. If you want to set a line or statement breakpoint on a statement that has this asterisk, you must first qualify to the containing compile unit by using the following command:

```
SET QUALIFY CU compile_unit_name;
```

After you enter this command, the asterisks are removed from the line on which you wanted to set a breakpoint. The absence of the asterisk indicates that you can set a line or statement breakpoint on that line.

You cannot use the SET QUALIFY command to qualify to an assembler compile unit until after you have loaded the debug data for that compile unit.

Halting when certain assembler routines are called

This topic describes how to halt just after a routine is called by using the AT ENTRY command. The "Example: sample assembler program for debugging" on page 243 is used to describe these commands.

To halt after the DISPARM routine is called, enter the following command:

```
AT ENTRY DISPARM
```

To halt after the DISPARM routine is called and only when R1 equals 0, enter the following command:

```
AT ENTRY DISPARM WHEN R1=0;
```

The AT CALL command is not supported for assembler routines. Do not use the AT CALL command to stop z/OS Debugger when an assembler routine is called.

Identifying the statement where your assembler program has stopped

If you have many breakpoints set in your program, you can enter the following command to have z/OS Debugger identify where your program has stopped:

```
QUERY LOCATION
```

The z/OS Debugger Log window displays something similar to the following example:

```
QUERY LOCATION
You are executing commands in the ENTRY XMPLOAD ::> DISPARM breakpoint.
The program is currently entering block XMPLOAD ::> DISPARM.
```

Displaying and modifying the value of assembler variables or storage

To list the contents of a single variable, move the cursor to an occurrence of the variable name in the Source window and press PF4 (LIST). The value is displayed in the Log window. This is equivalent to entering LIST *variable* on the command line.

For example, run the SUBXMP program to the statement labeled CALL1 by entering AT 70 ; GO ; on the z/OS Debugger command line. Scroll up until you see line 67. Move the cursor over COUNTER and press PF4 (LIST). The following appears in the Log window:

```
LIST ( COUNTER )
COUNTER = 0
```

To modify the value of COUNTER to 1, type over the COUNTER = 0 line with COUNTER = 1, press Enter to put it on the command line, and press Enter again to issue the command.

To list the contents of the 16 bytes of storage 2 bytes past the address contained in register R0, type the command LIST STORAGE(R0->+2,16) on the command line and press Enter. The contents of the specified storage are displayed in the Log window.

```
LIST STORAGE( R0 -> + 2 , 16 )
000C321E  C8859393 96408699 969440A3 888540A2   *Hello from the s*
```

To modify the first two bytes of this storage to X'C182', type the command R0->+2 <2> = X'C182'; on the command line and press Enter to issue the command.

Now step into the call to DISPARM by pressing PF2 (STEP) and step until the line labeled CALL2 is reached. To view the attributes of variable COUNTER, issue the z/OS Debugger command:

```
DESCRIBE ATTRIBUTES COUNTER
```

The result in the Log window is:

```
ATTRIBUTES for COUNTER
   Its address is  1B0E2150 and its length is 4
   DS F
```

Converting a hexadecimal address to a symbolic address

While you debug an assembler or disassembly program, you might want to determine the symbolic address represented by a hexadecimal address. You can do this by using the LIST command with the %WHERE built-in function. For example, the following command returns a string indicating the symbolic location of X'1BC5C':

```
LIST %WHERE(X'1BC5C')
```

After you enter the command, z/OS Debugger displays the following result:

```
PROG1+X'12C'
```

The result indicates that the address X'1BC5C' corresponds to offset X'12C' within CSECT PROG1.

Halting on a line in assembler only if a condition is true

Often a particular part of your program works fine for the first few thousand times, but it fails under certain conditions. Setting a line breakpoint is inefficient because you will have to repeatedly enter the GO command.

"Example: sample assembler program for debugging" on page 243

In the DISPARM program, to stop z/OS Debugger when the COUNTER variable is set to 3, enter the following command:

```
AT 78 DO; IF COUNTER ^= 3 THEN GO; END;
```

Line 78 is the line labeled BUMPCTR. The command causes z/OS Debugger to stop at line 78. If the value of COUNTER is not 3, the program continues. The command causes z/OS Debugger to stop on line 78 only if the value of COUNTER is 3.

Getting an assembler routine traceback

Often when you get close to a programming error, you want to know what sequence of calls lead you to the programming error. This sequence is called traceback or traceback of callers. To get the traceback information, enter the following command:

```
LIST CALLS
```

"Example: sample assembler program for debugging" on page 243

For example, if you run the SUBXMP example with the following commands, the Log window displays the traceback of callers:

```
AT ENTRY DISPARM
GO
LIST CALLS
```

The Log window displays information similar to the following:

```
At ENTRY IN Assembler routine XMPLOAD ::> DISPARM.
From LINE 76.1 IN Assembler routine XMPLOAD ::> SUBXMP.
```

Finding unexpected storage overwrite errors in assembler

While your program is running, some storage might unexpectedly change its value and you want to find out when and where this happened. Consider the following example, where the program finds a value unexpectedly modified:

```
L    R0,X'24'(R3)
```

To find the address of the operand being loaded, enter the following command:

```
LIST R3->+X'24'
```

Suppose the result is X'00521D42'. To set a breakpoint that watches for a change in storage values starting at that address and for the next 4 bytes, enter the following command:

```
AT CHANGE %STORAGE(X'00521D42',4)
```

When the program runs, z/OS Debugger stops if the value in this storage changes.

Chapter 28. Customizing your full-screen session

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You have several options for customizing your session. For example, you can resize and rearrange windows, close selected windows, change session parameters, and change session panel colors. This section explains how to customize your session using these options.

The window acted upon as you customize your session is determined by one of several factors. If you specify a window name (for example, `WINDOW OPEN MONITOR` to open the Monitor window), that window is acted upon. If the command is cursor-oriented, such as the `WINDOW SIZE` command, the window containing the cursor is acted upon. If you do not specify a window name and the cursor is not in any of the windows, the window acted upon is determined by the setting of *Default window* under the Profile Settings panel.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 21, “Using full-screen mode: overview,” on page 149
Chapter 28, “Customizing your full-screen session,” on page 251
“Defining PF keys” on page 251
“Defining a symbol for commands or other strings” on page 251
“Customizing the layout of physical windows on the session panel” on page 252
“Customizing session panel colors” on page 253
“Customizing profile settings” on page 254
“Saving customized settings in a preferences file” on page 256

Defining PF keys

To define your PF keys, use the `SET PFKEY` command. For example, to define the PF8 key as `SCROLL DOWN PAGE`, enter the following command:

```
SET PF8 "Down" = SCROLL DOWN PAGE ;
```

Use quotation marks (") for C and C++. You can use either apostrophes (') or quotation marks (") for assembler, COBOL, LangX COBOL, disassembly, and PL/I. The string set apart by the quotation marks or apostrophes (Down in this example) is the label that appears next to PF8 when you `SET KEYS ON` and your PF key definitions are displayed at the bottom of your screen.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

“Initial PF key settings” on page 163

Defining a symbol for commands or other strings

You can define a symbol to represent a long character string. For example, if you have a long command that you do not want to retype several times, you can use the `SET EQUATE` command to equate the command to a short symbol. Afterward, z/OS Debugger treats the symbol as though it were the command. The following examples show various settings for using EQUATEs:

- `SET EQUATE info = "abc, def(h+1)";` Sets the symbol `info` to the string, `"abc, def(h+1)"`.
- `CLEAR EQUATE (info);` Disassociates the symbol and the string. This example clears `info`.
- `CLEAR EQUATE;` If you do not specify what symbol to clear, all symbols created by `SET EQUATE` are cleared.

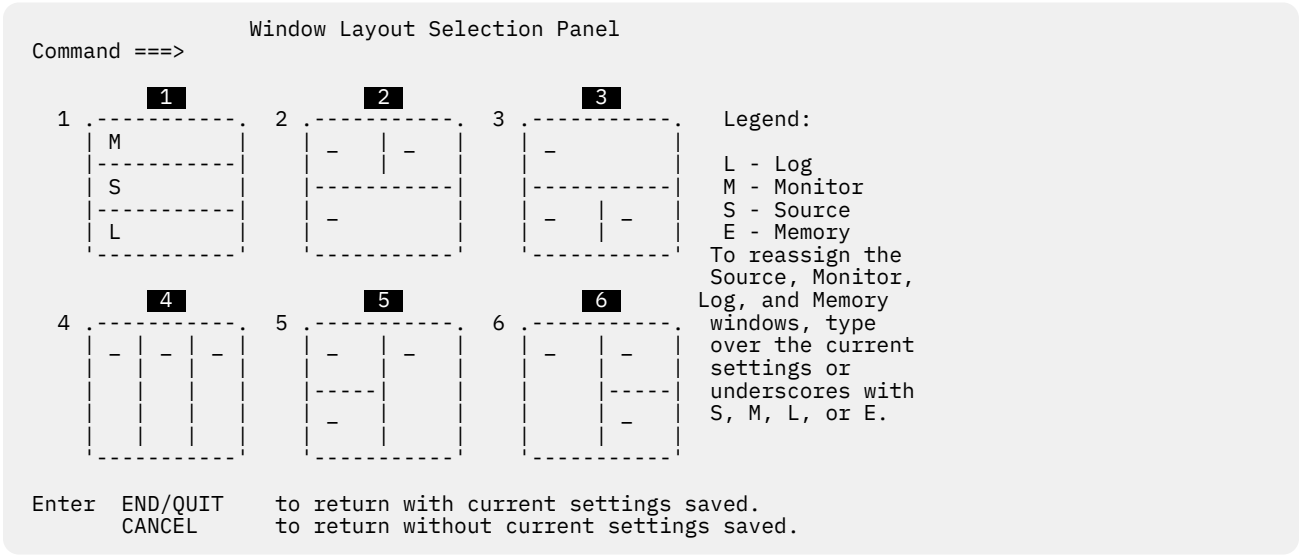
© Copyright IBM Corp. 1992, 2023

If a symbol created by a SET EQUATE command is the same as a keyword or keyword abbreviation in an HLL, the symbol takes precedence. If the symbol is already defined, the new definition replaces the old. Operands of certain commands are for environments other than the standard z/OS Debugger environment, and are not scanned for symbol substitution.

Customizing the layout of physical windows on the session panel

To change the relative layout of the physical windows, use the PANEL LAYOUT command (the PANEL keyword is optional). You can display either the Memory window or the Log window in one physical window, but you can not display both windows at the same time in separate physical windows.

The PANEL LAYOUT command displays the panel below, showing the six possible physical window layouts.



Initially, the session panel uses the default window layout 1.

Follow the instructions on the screen, then press the END PF key to save your changes and return to the main session panel in the new layout.

Note: You can choose only one of the six layouts. Also, only one of each type of window can be visible at a time on your session panel. For example, you cannot have two Log windows on a panel.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Opening and closing physical windows" on page 252
"Resizing physical windows" on page 253
"Zooming a window to occupy the whole screen" on page 253
"Saving customized settings in a preferences file" on page 256

Related references

"z/OS Debugger session panel" on page 149

Opening and closing physical windows

To close a physical window, do one of the following tasks:

- Type the WINDOW CLOSE command, move the cursor to the physical window you want to close, then press Enter.
- Enter one of the following commands:
  - WINDOW CLOSE LOG
  - WINDOW CLOSE MONITOR

  - WINDOW CLOSE SOURCE
  - WINDOW CLOSE MEMORY
- Assign the WINDOW CLOSE command to a PF key. Move the cursor to the physical window you want to close, then press the PF key.

When you close a physical window, the remaining windows occupy the full area of the screen.

To open a physical window, enter one of the following commands:

- WINDOW OPEN LOG
- WINDOW OPEN MONITOR
- WINDOW OPEN SOURCE
- WINDOW OPEN MEMORY

If you want to monitor the values of selected variables as they change during your z/OS Debugger session, you must display the Monitor window in a physical window. If it is not being displayed in a physical window, open a physical window as described above. The Monitor window occupies the available space according to your selected physical window layout.

If you open a physical window and the contents assigned to it are not available, the physical window is empty.

Resizing physical windows

To resize physical windows, do one of the following tasks:

- Type WINDOW SIZE on the command line, move the cursor to where you want the physical window boundary, then press Enter. The WINDOW keyword is optional.
- Specify the number of rows or columns you want the physical window to contain (as appropriate for the physical window layout) with the WINDOW SIZE command. For example, to change the physical window that is displaying the Source window from 10 rows deep to 12 rows deep, enter the following command:

```
WINDOW SIZE 12 SOURCE
```

- Assign the WINDOW SIZE command to a PF key. Move the cursor to where you want the physical window boundary, then press the PF key.

For the Memory window and the Monitor window, if you make a physical window too narrow to properly display the contents of that window, z/OS Debugger does not allow you to edit (by typing over) the contents of the window. If this happens, make the physical window wider.

To restore physical window sizes to their default values for the current physical window layout, enter the PANEL LAYOUT RESET command.

Zooming a window to occupy the whole screen

To toggle a window to full screen (temporarily not displaying the others), move the cursor into that window and press PF10 (ZOOM). Press PF10 to toggle back.

PF11 (ZOOM LOG) toggles the Log window in the same way, without the cursor needing to be in the Log window.

Customizing session panel colors

You can change the color and highlighting on your session panel to distinguish the fields on the panel. Consider highlighting such areas as the current line in the Source window, the prefix area, and the statement identifiers where breakpoints have been set.

To change the color, intensity, or highlighting of various fields of the session panel on a color terminal, use the PANEL COLORS command. When you issue this command, the panel shown below appears.

```
                    Color Selection Panel
Command ===>
                       Color    Highlight   Intensity
Title  : field headers  TURQ     NONE        HIGH
         output fields  GREEN    NONE        LOW      Valid Color:
Monitor: contents       TURQ     REVERSE     LOW        White Yellow Blue
         line numbers   TURQ     REVERSE     LOW        Turq Green Pink Red
Source : listing area   WHITE    REVERSE     LOW
         prefix area    TURQ     REVERSE     LOW      Valid Intensity:
         suffix area    YELLOW   REVERSE     LOW        High Low
         current line   RED      REVERSE     HIGH
         breakpoints    GREEN    NONE        LOW      Valid Highlight:
Log    : program output TURQ     NONE        HIGH       None Reverse
         test input     YELLOW   NONE        LOW        Underline Blink
         test output    GREEN    NONE        HIGH
         line numbers   BLUE     REVERSE     HIGH     Color and Highlight
Memory : information    GREEN    NONE        LOW      are valid only with
         offset column  WHITE    NONE        LOW      color terminals.
         address column YELLOW   NONE        LOW
         hex data       GREEN    NONE        LOW
         character data BLUE     NONE        LOW
 Command line           WHITE    NONE        HIGH
 Window headers         GREEN    REVERSE     HIGH
 Tofeof delimiter       BLUE     REVERSE     HIGH
 Search target          RED      NONE        HIGH
Enter  END/QUIT    to return with current settings saved.
       CANCEL      to return without current settings saved.

PF  1:?          2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP         8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

Initially, the session panel areas and fields have the default color and attribute values shown above.

The usable color attributes are determined by the type of terminal you are using. If you have a monochrome terminal, you can still use highlighting and intensity attributes to distinguish fields.

To change the color and attribute settings for your z/OS Debugger session, enter the desired colors or attributes over the existing values of the fields you want to change. The changes you make are saved when you enter QUIT.

You can also change the colors or intensity of selected areas by issuing the equivalent SET COLOR command from the command line. Either specify the fields explicitly, or use the cursor to indicate what you want to change. Changing a color or highlight with the equivalent SET command changes the value on the Color Selection Panel.

Settings remain in effect for the entire debug session.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Saving customized settings in a preferences file” on page 256

Customizing profile settings

The PANEL PROFILE command displays the Profile Settings Panel, which contains profile settings that affect the way z/OS Debugger runs. This panel is shown below with the IBM-supplied initial settings.

```
                        Profile Settings Panel
Command ===>

                                Current Setting
                                ---------------
 Change Test Granularity        STATEMENT      (All,Blk,Line,Path,Stmt)
 DBCS characters                NO             (Yes or No)
 Default Listing PDS name
 Default scroll amount          PAGE           (Page,Half,Max,Csr,Data,int)
 Default window                 SOURCE         (Log,Monitor,Source, Memory)
 Execute commands               YES            (Yes or No)
 History                        YES            (Yes or No)
 History size                   100            (nonnegative integer)
 Logging                        YES            (Yes or No)
 Pace of visual trace           2              (steps per second)
 Refresh screen                 NO             (Yes or No)
 Rewrite interval               50             (number of output lines)
 Session log size               1000           (number of retained lines)
 Show log line numbers          YES            (Yes or No)
 Show message ID numbers        NO             (Yes or No)
 Show monitor line numbers      YES            (Yes or No)
 Show scroll field              YES            (Yes or No)
 Show source/listing suffix     YES            (Yes or No)
 Show warning messages          YES            (Yes or No)
 Test level                     ALL            (All,Error,None)
Enter  END/QUIT    to return with current settings saved.
       CANCEL      to return without current settings saved.
```

You can change the settings either by typing your desired values over them, or by issuing the appropriate SET command at the command line or from within a commands file.

The profile parameters, their descriptions, and the equivalent SET commands are as follows:

Change Test Granularity
Specifies the granularity of testing for AT CHANGE. Equivalent to SET CHANGE.

DBCS characters
Controls whether the shift-in or shift-out characters are recognized. Equivalent to SET DBCS.

Default Listing PDS name
If specified, the data set where z/OS Debugger looks for the source or listing. Equivalent to SET DEFAULT LISTINGS.

Default scroll amount
Specifies the default amount assumed for SCROLL commands where no amount is specified. Equivalent to SET DEFAULT SCROLL.

Default window
Selects the default window acted upon when WINDOW commands are issued with the cursor on the command line. Equivalent to SET DEFAULT WINDOW.

Execute commands
Controls whether commands are executed or just checked for syntax errors. Equivalent to SET EXECUTE.

History
Controls whether a history (an account of each time z/OS Debugger is entered) is maintained. Equivalent to SET HISTORY.

History size
Controls the size of the z/OS Debugger history table. Equivalent to SET HISTORY.

Logging
Controls whether a log file is written. Equivalent to SET LOG.

Pace of visual trace
Sets the maximum pace of animated execution. Equivalent to SET PACE.

Refresh screen
Clears the screen before each display. REFRESH is useful when there is another application writing to the screen. Equivalent to SET REFRESH.

Rewrite interval
Defines the number of lines of intercepted output that are written by the application before z/OS Debugger refreshes the screen. Equivalent to `SET REWRITE`.

Session log size
The number of session log output lines retained for display. Equivalent to `SET LOG`.

Show log line numbers
Turns line numbers on or off in the log window. Equivalent to `SET LOG NUMBERS`.

Show message ID numbers
Controls whether ID numbers are shown in z/OS Debugger messages. Equivalent to `SET MSGID`.

Show monitor line numbers
Turns line numbers on or off in the Monitor window. Equivalent to `SET MONITOR NUMBERS`.

Show scroll field
Controls whether the scroll amount field is shown in the display. Equivalent to `SET SCROLL DISPLAY`.

Show source/listing suffix
Controls whether the frequency suffix column is displayed in the Source window. Equivalent TO `SET SUFFIX`.

Show warning messages (C and C++ and PL/I only)
Controls whether warning messages are shown or conditions raised when commands contain evaluation errors. Equivalent to `SET WARNING`.

Test level
Selects the classes of exceptions to cause automatic entry into z/OS Debugger. Equivalent to SET `TEST`.

A field indicating scrolling values is shown only if the screen is not large enough to show all the profile parameters at once. This field is not shown in the example panel above.

You can change the settings of these profile parameters at any time during your session. For example, you can increase the delay that occurs between the execution of each statement when you issue the STEP command by modifying the amount specified in the *Pace of visual trace* field at any time during your session.

To modify the profile settings for your session, enter a new value over the old value in the field you want to change. Equivalent SET commands are issued when you QUIT from the panel.

Entering the equivalent SET command changes the value on the Profile Settings panel as well.

Settings remain in effect for the entire debug session.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Saving customized settings in a preferences file" on page 256

Saving customized settings in a preferences file

You can place a set of commands into a data set, called a preferences file, and then indicate that file should be used by providing its name in the `preferences_file` suboption of the TEST run-time string. z/OS Debugger reads these commands at initialization and sets up the session appropriately.

Below is an example preferences file.

```
SET TEST ERROR;
SET DEFAULT SCROLL CSR;
SET HISTORY OFF;
SET MSGID ON;
DESCRIBE CUS;
```

Saving and restoring customizations between z/OS Debugger sessions

All of the customizations described in Chapter 28, “Customizing your full-screen session,” on page 251 can be preserved between z/OS Debugger sessions by using the save and restore settings feature. See “Recording how many times each source line runs” on page 173 for instructions.

Part 5. Debugging your programs by using z/OS Debugger commands

Note: Only some of the commands described in the chapters in this section are available in IBM Developer for z/OS (non-Enterprise Edition), IBM Z and Cloud Modernization Stack (Wazi Code). For a list of these commands, see Appendix A "z/OS Debugger commands supported in remote debug mode" in *IBM z/OS Debugger Reference and Messages*.

© Copyright IBM Corp. 1992, 2023

Chapter 29. Entering z/OS Debugger commands

z/OS Debugger commands can be issued in three modes: full-screen, line, and batch. Some z/OS Debugger commands are valid only in certain modes or programming languages. Unless otherwise noted, z/OS Debugger commands are valid in all modes, and for all supported languages.

For input typed directly at the terminal, input is free-form, optionally starting in column 1.

To separate multiple commands on a line, use a semicolon (;). This terminating semicolon is optional for a single command, or the last command in a sequence of commands.

For input that comes from a commands file or USE file, all of the z/OS Debugger commands must be terminated with a semicolon, except for the C `block` command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Entering commands on the session panel" on page 158
"Abbreviating z/OS Debugger keywords" on page 262
"Entering multiline commands in full-screen" on page 263
"Entering multiline commands in a commands file" on page 263
"Entering multiline commands without continuation" on page 263
"Using blanks in z/OS Debugger commands" on page 264
"Entering comments in z/OS Debugger commands" on page 264
"Using constants in z/OS Debugger commands" on page 264
"Getting online help for z/OS Debugger command syntax" on page 265

Related references

IBM z/OS Debugger Reference and Messages

Using uppercase, lowercase, and DBCS in z/OS Debugger commands

The character set and case vary with the double-byte character set (DBCS) or the current programming language setting in a z/OS Debugger session.

DBCS

When the DBCS setting is ON, you can specify DBCS characters in the following portions of all the z/OS Debugger commands:

- Commentary text
- Character data valid in the current programming language
- Symbolic identifiers such as variable names (for COBOL, this includes session variables), entry names, block names, and so forth (if the names contain DBCS characters in the application program).

When the DBCS setting is OFF, double-byte data is not correctly interpreted or displayed. However, if you use the shift-in and shift-out codes as data instead of DBCS indicators, you should issue `SET DBCS OFF`.

If you are debugging in full-screen mode and your terminal is not DBCS capable, the `SET DBCS ON` command is not available.

Character case and DBCS in C and C++

For both C and C++, z/OS Debugger sets the programming language to C. When the current programming language setting is C, the following rules apply:

© Copyright IBM Corp. 1992, 2023

- All keywords and identifiers must be the correct case. z/OS Debuggerdoes not convert them to uppercase.
- DBCS characters are allowed only within comments and literals.
- Either trigraphs or the equivalent special characters can be used. Trigraphs are treated as their equivalents at all times. For example, `FIND "??<"` would find not only "??<" but also "{". An exception is that column specifications other than `1 *` are not allowed in `FIND` or `SET FIND BOUNDS` if you search source code and trigraphs are found.
- The vertical bar (|) can be entered for the following C and C++ operations: bitwise or (|), logical or (||), and bitwise assignment or (|=).
- There are alternate code points for the following C and C++ characters: vertical bar (|), left brace ({), right brace (}), left bracket ([), and right bracket (]). Although alternate code points will be accepted as input for the braces and brackets, the primary code points will always be logged.

Character case in COBOL and PL/I

When the current programming language setting is *not* C, commands can generally be either uppercase, lowercase, or mixed. Characters in the range *a* through *z* are automatically converted to uppercase except within comments and quoted literals. Also, in PL/I, only "|" and "¬" can be used as the boolean operators for OR and NOT.

Abbreviating z/OS Debugger keywords

When you issue the z/OS Debugger commands, you can truncate most command keywords. You cannot truncate reserved keywords for the different programming languages, system keywords (that is, `SYS`, `SYSTEM`, or `TSO`) or special case keywords such as `BEGIN`, `CALL`, `COMMENT`, `COMPUTE`, `END`, `FILE` (in the `SET INTERCEPT` and `SET LOG` commands), `GOTO`, `INPUT`, `LISTINGS` (in the `SET DEFAULT LISTINGS` command), or `USE`. In addition, `PROCEDURE` can only be abbreviated as `PROC`.

The system keywords, and `COMMENT`, `INPUT`, and `USE` keywords, take precedence over other keywords and identifiers. If one of these keywords is followed by a blank, it is always parsed as the corresponding command. Hence, if you want to assign the value 2 to a variable named *TSO* and the current programming language setting is C, the "=" must be abutted to the reference, as in "`TSO<no space>= 2;`" not "`TSO<space>= 2;`". If you want to define a procedure named USE, you must enter "`USE<no space>: procedure;`" not "`USE<space>:: procedure;`".

When you truncate, you need only enter enough characters of the command to distinguish the command from all other valid z/OS Debugger commands. You should *not* use truncations in a commands file or compile them into programs because they might become ambiguous in a subsequent release. The following shows examples of z/OS Debugger command truncations:

If you enter the following command...	It will be interpreted as...
`A 3`	`AT 3`
`G`	`GO`
`Q B B`	`QUALIFY BLOCK B`
`Q Q`	`QUERY QUALIFY`
`Q`	`QUIT`

If you specify a truncation that is also a variable in your program, the keyword is chosen if this is the only ambiguity. For example, `LIST A` does not display the value of variable *A*, but executes the `LIST AT` command, listing your current AT breakpoints. To display the value of *A*, issue `LIST (A)`.

In addition, ambiguous commands that cannot be resolved cause an error message and are not performed. That is, there are two commands that could be interpreted by the truncation specified. For example, `D A A;` is an ambiguous truncation since it could either be `DESCRIBE ATTRIBUTES a;` or `DISABLE AT APPEARANCE;`. Instead, you would have to enter `DE A A;` if you wanted `DESCRIBE`

`ATTRIBUTES a;` or `DI A A;` if you wanted `DISABLE AT APPEARANCE;`. There are, of course, other variations that would work as well (for example, `D ATT A;`).

Entering multiline commands in full-screen

If you need to use more than one line to enter a command, you can do one of the following actions:

- Enter a continuation character when you reach the end of the command line.
- Enter the POPUP command before you enter the command.

In either case, z/OS Debugger displays the Command pop-up window.

When you enter a command in interactive mode, the continuation character must be the last non-blank character in the command line. In the following example, the continuation character is the single-byte character set (SBCS) hyphen (-):

```
LIST (" this is a very very very vvvvvvvvvvvvvvvvvvvvvvvvvvvvv -
very long string");
```

If you want to end a line with a character that z/OS Debugger might interpret as a continuation character, follow that character with another valid non-blank character. For example, in C and C++, if you want to enter "i––", you could enter "(i––)" or "i––;". When the current programming language setting is C and C++, you can use the backslash character (\).

When z/OS Debugger is awaiting the continuation of a command in full-screen mode, the Command pop-up window remains open and displays the message "Current® command is incomplete, enter more input below".

Entering multiline commands in a commands file

The rules for line continuation when input comes from a commands file are language-specific:

- When the current programming language setting is C and C++, identifiers, keywords, and literals can be continued from one line to the next if the backslash continuation character is used. The following is an example of the continuation character for C:

  ```
  LIST (" this is a very very very vvvvvvvvvvvvvvvvvvvvvvvvvvvvv\
  very long string");
  ```

- When the current programming language setting is COBOL, columns 1-6 are ignored by z/OS Debugger and input can be continued from one line to the next if the SBCS hyphen (-) is used in column 7 of the next line. Command text must begin in column 8 or later and end in or before column 72.

  In literal string continuation, a quotation mark (") or apostrophe (') is required at the end of the continued line. Then, a quotation mark (") or apostrophe (') is required at the beginning of the continuation line. The character following the quotation mark or apostrophe in the continuation line is considered to follow immediately after the last character in the continued line. The following is an example of line continuation for COBOL:

  ```
  123456 LIST (" this is a very very very vvvvvvvvvvvvvvvvvvvvvv"
  123456-"very long string");
  ```

  Continuation is not allowed within a DBCS name or literal string when the current programming language setting is COBOL.

Entering multiline commands without continuation

You can enter the following command parts on separate lines without using the SBCS hyphen (-) continuation character:

- Subcommands and the END keyword in the PROCEDURE command
- The programming language neutral BEGIN command.

- When the current programming language setting is C, statements that are part of a `compound` or `block` statement
- When the current programming language setting is COBOL:
  - `EVALUATE`
    - Subcommands in `WHEN` and `OTHER` clauses
    - `END-EVALUATE` keyword
  - `IF`
    - Subcommands in `THEN` and `ELSE` clauses
    - `END-IF` keyword
  - `PERFORM`
    - Subcommands
    - Subcommands in `UNTIL` clause
    - `END-PERFORM` keyword
- When the current programming language setting is PL/I, the `DO` command is for conditional looping.
- When the current programming language setting is assembler, disassembly, LangX COBOL, or COBOL, use the language-neutral `DO` command.

Refer to the following topics for more information related to the material discussed in this topic.

BEGIN command in *IBM z/OS Debugger Reference and Messages*
DO command (PL/I) in *IBM z/OS Debugger Reference and Messages*

Using blanks in z/OS Debugger commands

Blanks cannot occur within keywords, identifiers, and numeric constants; however, they can occur within character strings. Blanks between keywords, identifiers, or constants are ignored except as delimiters. Blanks are required when no other delimiter exists and ambiguity is possible.

Entering comments in z/OS Debugger commands

z/OS Debugger lets you insert descriptive comments into the command stream (except within constants and other comments); however, the comment format depends on the current programming language. The entire line, including comments and delimiter, must not extend beyond column 72.

For C++ only: Comments in the form "//" are not processed by z/OS Debugger in C++.

- For all supported programming languages, comments can be entered by:
  - Enclosing the text in comment brackets "/*" and "*/". Comments can occur anywhere a blank can occur between keywords, identifiers, and numeric constants. Comments entered in this manner do not appear in the session log.
  - Using the `COMMENT` command to insert commentary text in the session log. Comments entered in this manner cannot contain embedded semicolons.
- When the current programming language setting is COBOL, comments can also be entered by using an asterisk (*) in column 7. This is valid for file input only.
- For assembler and disassembly, comments can also be entered by using an asterisk (*) in column 1.

Comments are most helpful in file input. For example, you can insert comments in a USE file to explain and describe the actions of the commands.

Using constants in z/OS Debugger commands

Constants are entered as required by the current programming language setting. Most constants defined for each of the supported HLLs are also supported by z/OS Debugger.

z/OS Debugger allows the use of hexadecimal addresses in COBOL and PL/I.

The COBOL `H constant` is a fullword address value that can be specified in hex using numeric-hex-literal format (hexadecimal characters only, delimited by either quotation marks (") or apostrophes (') and preceded by H). The value is right-justified and padded on the left with zeros.

Note: The `H constant` can only be used where an address or `POINTER` variable can be used. You can use this type of constant with the SET command. For example, to assign a hexadecimal value of 124BF to the variable *ptr*, specify:

```
SET ptr TO  H"124BF";
```

The COBOL hexadecimal notation for alphanumeric literals, such as `MOVE X'C1C2C3C4' TO NON-PTR-VAR`, must be used for all other situations where a hexadecimal value is needed.

The PL/I `PX constant` is a hexadecimal value, delimited by apostrophes (') and followed by PX. The value is right-justified and can be used in any context in which a pointer value is allowed. For example, to display the contents at a given address in hexadecimal format, specify:

```
LIST STORAGE ('20CD0'PX);
```

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using constants in COBOL expressions" on page 274

Related references

"C and C++ expressions" on page 296

Getting online help for z/OS Debugger command syntax

You can get help with z/OS Debugger command syntax by either pressing PF1 or entering a question mark (?) on the command line. This lists all z/OS Debugger commands in the Log window.

To get a list of options for a command, enter a partial command followed by a question mark.

For example, in full-screen mode, enter on the command line:

```
?
WINDOW ?
WINDOW CLOSE ?
WINDOW CLOSE SOURCE
```

Now reopen the Source window with:

```
WINDOW OPEN SOURCE
```

to see the results.

The z/OS Debugger SYSTEM and TSO commands followed by ? do not invoke the syntax help; instead the ? is sent to the host as part of the system command. The `COMMENT` command followed by ? also does not invoke the syntax help.

Chapter 30. Debugging COBOL programs

Each version of the COBOL compiler provides enhancements that you can use to develop COBOL programs. These enhancements can create different levels of debugging capabilities. The topics below describe how to use these enhancements when you debug your COBOL programs.

"Qualifying variables and changing the point of view in COBOL" on page 275
"z/OS Debugger evaluation of COBOL expressions" on page 273
Chapter 22, "Debugging a COBOL program in full-screen mode," on page 197
"Using COBOL variables with z/OS Debugger" on page 268
"Using DBCS characters in COBOL" on page 271
"Using z/OS Debugger functions with COBOL" on page 274
"z/OS Debugger commands that resemble COBOL statements" on page 267
"%PATHCODE values for COBOL" on page 271
"Debugging VS COBOL II programs" on page 277

z/OS Debugger commands that resemble COBOL statements

To test COBOL programs, you can write debugging commands that resemble COBOL statements. z/OS Debugger provides an interpretive subset of COBOL statements that closely resembles or duplicates the syntax and action of the appropriate COBOL statements. You can therefore work with familiar commands and insert into your source code program patches that you developed during your debug session.

The table below shows the interpretive subset of COBOL statements recognized by z/OS Debugger.

Command	Description
`CALL`	Subroutine call
`COMPUTE`	Computational assignment (including expressions)
Declarations	Declaration of session variables
`EVALUATE`	Multiway switch
`IF`	Conditional execution
`MOVE`	Noncomputational assignment
`PERFORM`	Iterative looping
`SET`	`INDEX` and `POINTER` assignment

This subset of commands is valid only when the current programming language is COBOL.

Related references
IBM z/OS Debugger Reference and Messages

COBOL command format

When you are entering commands directly at your terminal or workstation, the format is free-form, because you can begin your commands in column 1 and continue long commands using the appropriate method. You can continue on the next line during your z/OS Debugger session by using an SBCS hyphen (-) as a continuation character.

However, when you use a file as the source of command input, the format for your commands is similar to the source format for the COBOL compiler. The first six positions are ignored, and an SBCS hyphen in column 7 indicates continuation from the previous line. You must start the command text in column 8 or later, and end it in column 72.

© Copyright IBM Corp. 1992, 2023

The continuation line (with a hyphen in column 7) optionally has one or more blanks following the hyphen, followed by the continuing characters. In the case of the continuation of a literal string, an additional quotation mark is required. When the token being continued is not a literal string, blanks following the last nonblank character on the previous line are ignored, as are blanks following the hyphen.

When z/OS Debugger copies commands to the log file, they are formatted according to the rules above so that you can use the log file during subsequent z/OS Debugger sessions.

Continuation is not allowed within a DBCS name or literal string. This restriction applies to both interactive and commands file input.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
“COBOL compiler options in effect for z/OS Debugger commands ” on page 268
“COBOL reserved keywords” on page 268
Enterprise COBOL for z/OS Language Reference

COBOL compiler options in effect for z/OS Debugger commands

While z/OS Debugger allows you to use many commands that are either similar or equivalent to COBOL commands, z/OS Debugger does not necessarily interpret these commands according to the compiler options you chose when compiling your program. This is due to the fact that, in the z/OS Debugger environment, the following settings are in effect:

```
DYNAM
NOCMPR2
NODBCS
NOWORD
NUMPROC(NOPFD)
QUOTE
TRUNC(BIN)
ZWB
```

Related references
Enterprise COBOL for z/OS Language Reference

COBOL reserved keywords

In addition to the subset of COBOL commands you can use while in z/OS Debugger, there are reserved keywords used and recognized by COBOL that cannot be abbreviated, used as a variable name, or used as any other type of identifier.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Enterprise COBOL for z/OS Language Reference

Using COBOL variables with z/OS Debugger

z/OS Debugger can process all variable types valid in the COBOL language.

For programs compiled with Enterprise COBOL for z/OS Version 6 Release 2 or earlier, the decimal point is always a period for numeric constants and variables. The `DECIMAL-POINT IS COMMA` clause has no effect.

In addition to being allowed to assign values to variables and display the values of variables during your session, you can declare session variables to suit your testing needs.

“Example: assigning values to COBOL variables” on page 269

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Accessing COBOL variables" on page 269
"Assigning values to COBOL variables" on page 269
"Displaying values of COBOL variables" on page 270
"Declaring session variables in COBOL" on page 272

Accessing COBOL variables

z/OS Debugger obtains information about a program variable by name, using information that is contained in the symbol table built by the compiler. You make the symbol table available to z/OS Debugger by compiling with the TEST compiler option.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27

Assigning values to COBOL variables

z/OS Debugger provides three COBOL-like commands to use when assigning values to variables: COMPUTE, MOVE, and SET. z/OS Debugger assigns values according to COBOL rules. See *IBM z/OS Debugger Reference and Messages* for tables that describe the allowable values for the source and receiver of the COMPUTE, MOVE, and SET commands.

Example: assigning values to COBOL variables

The examples for the COMPUTE, MOVE, and SET commands use the declarations defined in the following COBOL program segment.

```
01  GRP.
    02  ITM-1 OCCURS 3 TIMES INDEXED BY INX1.
        03  ITM-2 PIC 9(3) OCCURS 3 TIMES INDEXED BY INX2.
01  B.
    02  A     PIC 9(10).
01  D.
    02  C     PIC 9(10).
01  F.
    02  E     PIC 9(10)     OCCURS 5 TIMES.
77     AA          PIC X(5)     VALUE 'ABCDE'.
77     BB          PIC X(5).
        88         BB-GOOD-VALUE VALUE 'BBBBB'.
77     XX          PIC 9(9)    COMP.
77     ONE         PIC 99       VALUE 1.
77     TWO         PIC 99       VALUE 2.
77     PTR         POINTER.
```

Assign the value of TRUE to BB-GOOD-VALUE. Only the TRUE value is valid for level-88 receivers. For example:

```
SET BB-GOOD-VALUE TO TRUE;
```

Assign to variable xx the result of the expression `(a + e(1))/c * 2.`

```
COMPUTE xx =(a + e(1))/c * 2;
```

You can also use table elements in such assignments as shown in the following example:

```
COMPUTE itm-2(1,2)=(a + 1)/e(2);
```

The value assigned to a variable is always assigned to the storage for that variable. In an optimized program, a variable might be temporarily assigned to a register, and a new value assigned to that variable might not alter the value used by the program.

Assign to the program variable c , found in structure d , the value of the program variable a , found in structure b:

```
MOVE a OF b TO c OF d;
```

Note the qualification used in this example.

Assign the value of 123 to the first table element of `itm-2`:

```
MOVE 123 TO itm-2(1,1);
```

You can also use reference modification to assign values to variables as shown in the following two examples:

```
MOVE aa(2:3)TO bb;
MOVE aa TO bb(1:4);
```

Assign the value 3 to `inx1`, the index to `itm-1`:

```
SET inx1 TO 3;
```

Assign the value of `inx1` to `inx2`:

```
SET inx2 TO inx1;
```

Assign the value of an invalid address (nonnumeric 0) to `ptr`:

```
SET ptr TO NULL;
```

Assign the address of XX to `ptr`:

```
SET ptr TO ADDRESS OF XX;
```

Assigns the hexadecimal value of X'20000' to the pointer `ptr`:

```
SET ptr TO H'20000';
```

Displaying values of COBOL variables

To display the values of variables, issue the LIST command. The LIST command causes z/OS Debugger to log and display the current values (and names, if requested) of variables. For example, if you want to display the variables `aa, bb, one`, and their respective values at statement 52 of your program, issue the following command:

```
AT 52 LIST TITLED (aa, bb, one); GO;
```

z/OS Debugger sets a breakpoint at statement 52 (AT), begins execution of the program (GO), stops at statement 52, and displays the variable names (TITLED) and their values.

Put commas between the variables when listing more than one. If you do not want to display the variable names when issuing the LIST command, issue `LIST UNTITLED` instead of `LIST TITLED`.

The value displayed for a variable is always the value that was saved in storage for that variable. In an optimized program, a variable can be temporarily assigned to a register, and the value shown for that variable might differ from the value being used by the program.

If you use the LIST command to display a National variable, z/OS Debugger converts the Unicode data to EBCDIC before displaying it. If the conversion results in characters that cannot be displayed, enter the `LIST %HEX()` command to display the unconverted Unicode data in hexadecimal format.

If you use the LIST command to display a UTF-8 variable, z/OS Debugger converts the UTF-8 data to EBCDIC before displaying it. If the conversion results in characters that cannot be displayed, enter the `LIST %HEX()` command to display the unconverted UTF-8 data in hexadecimal format.

Using DBCS characters in COBOL

Programs you run with z/OS Debugger can contain variables and character strings written using the double-byte character set (DBCS). z/OS Debugger also allows you to issue commands containing DBCS variables and strings. For example, you can display the value of a DBCS variable (LIST), assign it a new value, monitor it in the Monitor window (MONITOR), or search for it in a window (FIND).

To use DBCS with z/OS Debugger, enter:

```
SET DBCS ON;
```

If you are debugging in full-screen mode and your terminal is not DBCS capable, the SET DBCS ON is not available.

The DBCS default for COBOL is OFF.

The DBCS syntax and continuation rules you must follow to use DBCS variables in z/OS Debugger commands are the same as those for the COBOL language.

For COBOL you must type a DBCS literal, such as G, in front of a DBCS value in a Monitor or Data pop-up window if you want to update the value.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Enterprise COBOL for z/OS Language Reference

%PATHCODE values for COBOL

The table below shows the possible values for the z/OS Debugger variable %PATHCODE when the current programming language is COBOL.

–1	z/OS Debugger is not in control as the result of a path or attention situation.
0	Attention function (*not* ATTENTION condition).
1	A block has been entered.
2	A block is about to be exited.
3	Control has reached a label coded in the program (a paragraph name or section name).
4	Control is being transferred as a result of a CALL or INVOKE. The invoked routine's parameters, if any, have been prepared.
5	Control is returning from a CALL or INVOKE. If GPR 15 contains a return code, it has already been stored.
6	Some logic contained by an inline PERFORM is about to be executed. (Out-of-line PERFORM ranges must start with a paragraph or section name, and are identified by %PATHCODE = 3.)
7	The logic following an IF...THEN is about to be executed.
8	The logic following an ELSE is about to be executed.
9	The logic following a WHEN within an EVALUATE is about to be executed.
10	The logic following a WHEN OTHER within an EVALUATE is about to be executed.
11	The logic following a WHEN within a SEARCH is about to be executed.
12	The logic following an AT END within a SEARCH is about to be executed.

13	The logic following the end of one of the following structures is about to be executed: • An IF statement (with or without an ELSE clause) • An EVALUATE or SEARCH • A PERFORM
14	Control is about to return from a declarative procedure such as USE AFTER ERROR. (Declarative procedures must start with section names, and are identified by %PATHCODE = 3.)
15	The logic associated with one of the following phrases is about to be run: • [NOT] ON SIZE ERROR • [NOT] ON EXCEPTION • [NOT] ON OVERFLOW • [NOT] AT END (other than SEARCH AT END) • [NOT] AT END-OF-PAGE • [NOT] INVALID KEY
16	The logic following the end of a statement containing one of the following phrases is about to be run: • [NOT] ON SIZE ERROR • [NOT] ON EXCEPTION • [NOT] ON OVERFLOW • [NOT] AT END (other than SEARCH AT END) • [NOT] AT END-OF-PAGE • [NOT] INVALID KEY.

Note: Values in the range 3–16 can be assigned to %PATHCODE only if your program was compiled with an option supporting path hooks.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Choosing TEST or NOTEST compiler suboptions for COBOL programs” on page 27

Declaring session variables in COBOL

You might want to declare session variables during your z/OS Debugger session. The relevant variable assignment commands are similar to their counterparts in the COBOL language. The rules used for forming variable names in COBOL also apply to the declaration of session variables during a z/OS Debugger session.

The following declarations are for a string variable, a decimal variable, a pointer variable, and a floating-point variable. To declare a string named `description`, enter:

```
77 description      PIC X(25)
```

To declare a variable named `numbers`, enter:

```
77 numbers          PIC 9(4) COMP
```

To declare a pointer variable named `pinkie`, enter:

```
77 pinkie           POINTER
```

To declare a floating-point variable named `shortfp`, enter:

```
77 shortfp          COMP-1
```

Session variables remain in effect for the entire debug session.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Using session variables across different programming languages" on page 377

Related references
Enterprise COBOL for z/OS Language Reference

z/OS Debugger evaluation of COBOL expressions

z/OS Debugger interprets COBOL expressions according to COBOL rules. Some restrictions do apply. For example, the following restrictions apply when arithmetic expressions are specified:

- Floating-point operands are not supported (`COMP-1, COMP-2`, external floating point, floating-point literals).
- Only integer exponents are supported.
- Intrinsic functions and user defined functions are not supported.
- Windowed date-field operands are not supported in arithmetic expressions in combination with any other operands.

When arithmetic expressions are used in relation conditions, both comparand attributes are considered. Relation conditions follow the `IF` rules rather than the `EVALUATE` rules.

Only simple relation conditions are supported. Sign conditions, class conditions, condition-name conditions, switch-status conditions, complex conditions, and abbreviated conditions are not supported. When either of the comparands in a relation condition is stated in the form of an arithmetic expression (using operators such as plus and minus), the restriction concerning floating-point operands applies to both comparands. See *IBM z/OS Debugger Reference and Messages* for a table that describes the allowable comparisons for the `IF` command. See the *Enterprise COBOL for z/OS Programming Guide* for a description of the COBOL rules of comparison.

Windowed date fields are not supported in relation conditions.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Displaying the results of COBOL expression evaluation" on page 273
"Using constants in COBOL expressions" on page 274
Enterprise COBOL for z/OS Programming Guide

Related references
IBM z/OS Debugger Reference and Messages

Displaying the results of COBOL expression evaluation

Use the `LIST` command to display the results of your expressions. For example, to evaluate the expression and displays the result in the Log window, enter:

```
LIST a + (a - 10) + one;
```

You can also use structure elements in expressions. If e is an array, the following two examples are valid:

```
LIST a + e(1) ,' c * two;
```

```
LIST xx ,' e(two + 3);
```

Conditions for expression evaluation are the same ones that exist for program statements.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
“COBOL compiler options in effect for z/OS Debugger commands ” on page 268
Enterprise COBOL for z/OS Language Reference

Using constants in COBOL expressions

During your z/OS Debugger session you can use expressions that use string constants as one operand, as well as expressions that include variable names or number constants as single operands. All COBOL string constant types discussed in the *Enterprise COBOL for z/OS Language Reference* are valid in z/OS Debugger, with the following restrictions:

- The following COBOL figurative constants are supported:

  ```
  ZERO, ZEROS, ZEROES
  SPACE, SPACES
  HIGH-VALUE, HIGH-VALUES
  LOW-VALUE, LOW-VALUES
  QUOTE, QUOTES
  NULL, NULLS
  ```
  Any of the above preceded by `ALL`
  Symbolic-character (whether or not preceded by `ALL`).
- An N literal, which starts with `N"` or `N'`, is always treated as a national literal.
- A U literal, which starts with `U"` or `U'`, is always treated as a UTF-8 literal.

Additionally, z/OS Debugger allows the use of a hexadecimal constant that represents an address. This *H-constant* is a fullword value that can be specified in hex using numeric-hex-literal format (hexadecimal characters only, delimited by either quotation marks (") or apostrophes (') and preceded by H). The value is right-justified and padded on the left with zeros. The following example:

```
LIST STORAGE (H'20cd0');
```

displays the contents at a given address in hexadecimal format. You can use this type of constant with the SET command. The following example:

```
SET ptr TO H'124bf';
```

assigns a hexadecimal value of 124bf to the variable `ptr`.

Using z/OS Debugger functions with COBOL

z/OS Debugger provides certain functions you can use to find out more information about program variables and storage.

Using %HEX with COBOL

You can use the %HEX function with the `LIST` command to display the hexadecimal value of an operand. For example, to display the external representation of the packed decimal `pvar3`, defined as `PIC 9(9)`, from 1234 as its hexadecimal (or internal) equivalent, enter:

```
LIST %HEX (pvar3);
```

The Log window displays the hexadecimal string `X'F0F0F0F0F0F1F2F3F4'`.

Using the %STORAGE function with COBOL

This z/OS Debugger function allows you to reference storage by address and length. By using the %STORAGE function as the reference when setting a CHANGE breakpoint, you can watch specific areas of storage for changes. For example, to monitor eight bytes of storage at the hex address 22222 for changes, enter:

```
AT CHANGE %STORAGE (H'00022222', 8)
   LIST 'Storage has changed at Hex address 22222'
```

Qualifying variables and changing the point of view in COBOL

Qualification is a method of specifying an object through the use of qualifiers, and changing the point of view from one block to another so you can manipulate data not known to the currently executing block. For example, the assignment `MOVE 5 TO x;` does not appear to be difficult for z/OS Debugger to process. However, you might have more than one variable named x. You must tell z/OS Debugger which variable x to assign the value of five.

You can use qualification to specify to what compile unit or block a particular variable belongs. When z/OS Debugger is invoked, there is a default qualification established for the currently executing block; it is *implicitly* qualified. Thus, you must explicitly qualify your references to all statement numbers and variable names in any other block. It is necessary to do this when you are testing a compile unit that calls one or more blocks or compile units. You might need to specify what block contains a particular statement number or variable name when issuing commands.

Qualifying variables in COBOL

Qualifiers are combinations of load modules, compile units, blocks, section names, or paragraph names punctuated by a combination of greater-than signs (>), colons, and the COBOL data qualification notation, OF or IN, that precede referenced statement numbers or variable names.

When qualifying objects on a block level, use only the COBOL form of data qualification. If data names are unique, or defined as GLOBAL, they do not need to be qualified to the block level.

The following is a fully qualified object:

```
load_name::>cu_name:>block_name:>object;
```

If required, *load_name* is the name of the load module. It is required only when the program consists of multiple load modules and you want to change the qualification to other than the current load module. *load_name* can also be the z/OS Debugger variable %LOAD.

If required, *cu_name* is the name of the compile unit. The *cu_name* must be the fully qualified compile unit name. It is required only when you want to change the qualification to other than the currently qualified compile unit. It can be the z/OS Debugger variable %CU.

If required, *block_name* is the name of the block. The *block_name* is required only when you want to change the qualification to other than the currently qualified block. It can be the z/OS Debugger variable %BLOCK. If *block_name* is case sensitive, enclose the block name in quotation marks (") or apostrophes ('). If the name is not inside quotation marks or apostrophes, z/OS Debugger converts the name to upper case.

Below are two similar COBOL programs (blocks).

```
MAIN
⋮
   01  VAR1.
       02  VAR2.
           03  VAR3     PIC XX.
   01  VAR4     PIC  99..

  ***************MOVE commands entered here***************
```

```
SUBPROG
⋮
   01  VAR1.
       02  VAR2.
           03  VAR3     PIC XX.
   01  VAR4     PIC  99.
   01  VAR5     PIC  99.

  ****************LIST commands entered here*****************
```

You can distinguish between the `main` and `subprog` blocks using qualification. If you enter the following `MOVE` commands when `main` is the currently executing block:

```
MOVE 8 TO var4;
MOVE 9 TO subprog:>var4;
MOVE 'A' TO var3 OF var2 OF var1;
MOVE 'B' TO subprog:>var3 OF var2 OF var1;
```

and the following `LIST` commands when `subprog` is the currently executing block:

```
LIST TITLED var4;
LIST TITLED main:>var4;
LIST TITLED var3 OF var2 OF var1;
LIST TITLED main:>var3 OF var2 OF var1;
```

each `LIST` command results in the following output (without the commentary) in your Log window:

```
VAR4 = 9;    /*  var4 with no qualification refers to a variable     */
             /*  in the currently executing block (subprog).         */
             /*  Therefore, the LIST command displays the value of 9.*/

MAIN:>VAR4 = 8     /*  var4 is qualified to main.                    */
                   /*  Therefore, the LIST command displays 8,       */
                   /*  the value of the variable declared in main.   */

VAR3 OF VAR2 OF VAR1 = 'B';
               /*  In this example, although the data qualification  */
               /*  of var3 is OF var2 OF var1, the                   */
               /*  program qualification defaults to the currently   */
               /*  executing block and the LIST command displays     */
               /*  'B', the value declared in subprog.               */

VAR3 OF VAR2 OF VAR1 = 'A'
               /*  var3 is again qualified to var2 OF var1           */
               /*  but further qualified to main.                    */
               /*  Therefore, the LIST command displays              */
               /*  'A', the value declared in main.                  */
```

The above method of qualifying variables is necessary for commands files.

Changing the point of view in COBOL

The point of view is usually the currently executing block. You can also get to inaccessible data by changing the point of view using the SET QUALIFY command. The SET keyword is optional. For example, if the point of view (current execution) is in `main` and you want to issue several commands using variables declared in `subprog`, you can change the point of view by issuing the following:

```
QUALIFY BLOCK subprog;
```

You can then issue commands using the variables declared in `subprog` without using qualifiers. z/OS Debugger does not see the variables declared in procedure `main`. For example, the following assignment commands are valid with the `subprog` point of view:

```
MOVE 10 TO var5;
```

However, if you want to display the value of a variable in `main` while the point of view is still in `subprog`, you must use a qualifier, as shown in the following example:

```
LIST (main:>var-name);
```

The above method of changing the point of view is necessary for command files.

Considerations when debugging a COBOL class

The block structure of a COBOL class created with Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 or later, is different from the block structure of a COBOL program. The block structure of a COBOL class has the following differences:

- The CLASS is a compile unit.
- The FACTORY paragraph is a block.
- The OBJECT paragraph is a block.
- Each method is a block.

A method belongs to either the FACTORY block or the OBJECT block. A fully qualified block name for a method in the FACTORY paragraph is:

```
class-name:>FACTORY:>method-name
```

A fully qualified block name for a method in the OBJECT paragraph is:

```
class-name:>OBJECT:>method-name
```

When you are at a breakpoint in a method, the currently qualified block is the method. If you enter the `LIST TITLED` command with no parameters, z/OS Debugger lists all of the data items associated with the method. To list all of the data items in a FACTORY or OBJECT, do the following steps:

1. Enter the `QUALIFY` command to set the point of view to the FACTORY or OBJECT.
2. Enter the `LIST TITLED` command.

For example, to list all of the object instance data items for a class called ACCOUNT, enter the following command:

```
QUALIFY BLOCK ACCOUNT:>OBJECT; LIST TITLED;
```

Debugging VS COBOL II programs

There are limitations to debugging VS COBOL II programs compiled with the TEST compiler option and linked with the Language Environment library. Language Environment callable services, including CEETEST, are not available. However, you must use the Language Environment run time.

z/OS Debugger does not get control of the program at breakpoints that you set by the following commands:

- `AT PATH`
- `AT CALL`
- `AT ENTRY`
- `AT EXIT`
- `AT LABEL`

However, if you set the breakpoint with an `AT CALL` command that calls a non-VS COBOL II program, z/OS Debugger does get control of the program. Use the `AT ENTRY *`, `AT EXIT *`, `AT GLOBAL ENTRY`, and `AT GLOBAL EXIT` commands to set breakpoints that z/OS Debugger can use to get control of the program.

Breakpoints that you set at entry points and exit statements have no statement associated with them. Therefore, they are triggered only at the compile unit level. When they are triggered, the current view of the listing moves to the top and no statement is highlighted. Breakpoints that you set at entry points and exit statements are ignored by the STEP command.

If you are debugging your VS COBOL II program in remote debug mode, use the same TEST run-time options as for any COBOL program.

Finding the listing of a VS COBOL II program

The VS COBOL II compiler does not place the name of the listing data set in the object (load module). z/OS Debugger tries to find the listing data set in the following location: `userid.CUName.LIST`. If the listing is in a PDS, direct z/OS Debugger to the location of the PDS in one of the following ways:

- In full-screen mode, do one of the following options:
  - Enter the `SET DEFAULT LISTINGS` command.
  - Enter the `SET SOURCE` command.
  - Enter the `PANEL PROFILE` command, which displays the Profile Settings panel. Enter the new file name in the Default Listing PDS name field.
  - Enter the command `PANEL LISTINGS` command, which displays the Source Identification Panel. Enter the name of the PDS over the existing name in the Listings/Source File column, then press PF3.
- In remote debug mode, enter the command `SET DEFAULT LISTINGS`.
- Use the EQADEBUG DD statement to define the location of the data set.
- Code the EQAUEDAT user exit with the location of the data set.

For additional information about how you can debug VS COBOL II programs, see *Using CODE/370 with VS COBOL II and OS PL/I*, SC09-1862.

Chapter 31. Debugging a LangX COBOL program

Note: This chapter is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

You can use most of the z/OS Debugger commands to debug LangX COBOL programs that have debug information available. Any exceptions are noted in *IBM z/OS Debugger Reference and Messages*. Before debugging a LangX COBOL program, prepare your program as described in Chapter 6, "Preparing a LangX COBOL program," on page 67.

As you read through the information in this document, remember that OS/VS COBOL programs are non-Language Environment programs, even though you might have used Language Environment libraries to link and run your program.

VS COBOL II programs are non-Language Environment programs when you link them with the non-Language Environment library. VS COBOL II programs are Language Environment programs when you link them with the Language Environment library.

Enterprise COBOL programs are always Language Environment programs. Note that COBOL DLL's cannot be debugged as LangX COBOL programs.

Read the information regarding non-Language Environment programs for instructions on how to start z/OS Debugger and debug non-Language Environment COBOL programs, unless information specific to LangX COBOL is provided.

Loading a LangX COBOL program's debug information

Use the LOADDEBUGDATA (LDD) command to indicate to z/OS Debugger that a compile unit is a LangX COBOL compile unit and to load the debug information associated with that compile unit. The LDD command can be used only for compile units that are considered disassembly compile units. In the following example, `mypgm` is the compile unit name of an OS/VS COBOL program: `LDD mypgm`

z/OS Debugger locates the debug information in a data set with the following name: `yourid.EQALANGX(mypgm)`. If z/OS Debugger finds this data set, you can begin to debug your LangX COBOL program. If z/OS Debugger does not find the data set, enter the `SET SOURCE` or `SET DEFAULT LISTINGS` command to indicate to z/OS Debugger where to find the debug information.

Normally, compile units without debug information are not listed when you enter the `DESCRIBE CUS` or `LIST NAMES CUS` commands. To include these compile units, enter the `SET ASSEMBLER ON` command. The next time you enter the `DESCRIBE CUS` or `LIST NAMES CUS` command, these compile units are listed.

z/OS Debugger session panel while debugging a LangX COBOL program

The z/OS Debugger session panel below shows the information displayed in the Source window while you debug a LangX COBOL program.

© Copyright IBM Corp. 1992, 2023

```
 1 LX COBOL LOCATION: COB03O initialization
Command ===>                                                  Scroll ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 0 OF 0
****************************** TOP OF MONITOR ******************************
***************************** BOTTOM OF MONITOR ****************************
SOURCE: COB03O ---1----+----2----+----3----+----4----+----5----+ LINE: 1 OF 111
   2  1     3 ****************************************************         .
       2       *  PROGRAM NAME: COB03O                             *         .
       3       *                                                   *         .
       4       *  COMPILED WITH IBM OS/VS COBOL COMPILER           *         .
       5       ****************************************************         .
       7        IDENTIFICATION DIVISION.                                    .
       8        PROGRAM-ID. COB03O.                                         .
       9       ************************************************************ .
      10       *                                                            .
      11       * LICENSED MATERIALS - PROPERTY OF IBM                       .
      12       *                                                            .
      13       * 5655-P14: Debug Tool
      14       * (C) Copyright IBM Corp. 2004 All Rights Reserved
      15       *                                                            .
      16       * US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR  .
      17       * DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM  .
      18       * CORP.                                                      .
      19       *                                                            .
      20       *                                                            .
      21       ************************************************************ .
      22        ENVIRONMENT DIVISION.                                       .
      23        DATA DIVISION.                                              .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 7
******************************** TOP OF LOG ********************************
IBM z/OS Debugger 15.0.n
08/04/2020 03:55:40 AM
5724-T07: Copyright IBM Corp. 1992, 2020
0004 *** Commands file commands follow ***
0005  SET MSGID ON ;
0006  LDD ( COB03O, COB03AO ) ;
0007 EQA1891I *** Commands file commands end ***
******************************* BOTTOM OF LOG ******************************


PF  1:?          2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP         8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

The information displayed in the Source window is similar to the listing generated by the COBOL compiler. The Source window displays the following information:

1 LX COBOL
This indicates that the current source program is LangX COBOL.

2 line number
The line number is a number assigned by the EQALANGX program by sequentially numbering the source lines. Use the numbers in this column to set breakpoints and identify statements.

3 source statement
The original source statement.

Restrictions for debugging a LangX COBOL program

When you debug LangX COBOL programs the following general restrictions apply:

- When you compose z/OS Debugger commands, all expressions must be enclosed in apostrophes (')
- The AT CALL command is not supported
- The AT EXIT command is not supported
- The STEP RETURN command is not supported
- You cannot use the LIST command on a level-88 variables.
- You cannot use the assignment statement to alter the contents of a level-88 variable.
- If you enter a STEP command when stopped on a statement that returns control to a higher-level program, the STEP command acts like a GO command.

- The only path-points for the AT PATH statement that are supported in a LangX COBOL program are Entry and Label.
- There are behavioral differences between LangX COBOL programs and other COBOL programs. LangX COBOL programs behave more like assembler programs than COBOL programs in many situations. For example, in COBOL, a CU is not known to z/OS Debugger until it is called, even if it is statically link-edited into the same load module as the calling CU. However, LangX COBOL CU's are all known to z/OS Debugger when the module is loaded.
- If you are debugging a non-Language Environment VS COBOL II program that uses the CALL statement to invoke a Language Environment program, you cannot stop at or debug the Language Environment program.
- The output of the `DESCRIBE ATTRIBUTES` command might not match the attributes originally coded in the following situations:
  - For packed decimal numbers (COMP-3) the PIC attribute always indicate an odd number of digits. This might be one more digit than was coded in the original PIC.
  - The only non-numeric PIC code that is displayed by z/OS Debugger is 'X'.
- Under CICS, the initialization of a non-Language Environment COBOL transaction is single-threaded; therefore, when multiple users try to concurrently debug a non-Language Environment COBOL program, the CICS environment initializes one non-Language Environment COBOL transaction at a time. Consider the following example:
  1. Three users start a transaction that runs non-Language Environment COBOL program.
  2. The transaction that started first is initialized first. The other two transactions have to wait until that initialization is completed.
  3. After the initialization of the transaction that started first is done, the transaction that started second is initialized. While this transaction is being initialized, the user of the transaction that started first can run his z/OS Debugger session and the user of the transaction that started third continues to wait.
  4. After the initialization of the transaction that started second is done, the transaction that started third is initialized. While this transaction is being initialized, the users of the other two transactions can run their z/OS Debugger sessions.
  5. After the initialization of the transaction that started third is done, all three users can run their z/OS Debugger sessions, independently, for the same non-Language Environment COBOL program.

%PATHCODE values for LangX COBOL programs

This table shows the possible values for the z/OS Debugger variable %PATHCODE when the current programming language is LangX COBOL:

%PATHCODE	Entry Type
1	A block has been entered
3	Control has reached a label coded in the program.

Restrictions for debugging non-Language Environment programs

If you specify the TEST run-time option with the NOPROMPT suboption when you start your program, and z/OS Debugger is subsequently started by CALL CEETEST or the raising of a Language Environment condition, then you can debug both Language Environment and non-Language Environment programs and detect both Language Environment and non-Language Environment events in the enclave that started z/OS Debugger and in subsequent enclaves. You cannot debug non-Language Environment programs or detect non-Language Environment events in higher-level enclaves. After control has returned from the enclave in which z/OS Debugger was started, you can no longer debug non-Language Environment programs or detect non-Language Environment events.

Chapter 32. Debugging PL/I programs

The topics below describe how to use z/OS Debugger to debug your PL/I programs.

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts
"z/OS Debugger evaluation of PL/I expressions" on page 289

Related tasks
Chapter 24, "Debugging a PL/I program in full-screen mode," on page 213
Chapter 32, "Debugging PL/I programs," on page 283
"Accessing PL/I program variables" on page 287

Related references
"z/OS Debugger subset of PL/I commands" on page 283
"Supported PL/I built-in functions" on page 289

z/OS Debugger subset of PL/I commands

The table below lists the z/OS Debugger *interpretive subset* of PL/I commands. This subset is a list of commands recognized by z/OS Debugger that either closely resemble or duplicate the syntax and action of the corresponding PL/I command. This subset of commands is valid only when the current programming language is PL/I.

Command	Description
`Assignment`	Scalar and vector assignment
`BEGIN`	Composite command grouping
`CALL`	z/OS Debugger procedure call
`DECLARE or DCL`	Declaration of session variables
`DO`	Iterative looping and composite command grouping
`IF`	Conditional execution
`ON`	Define an exception handler
`SELECT`	Conditional execution

PL/I language statements

PL/I statements are entered as z/OS Debugger *commands*. z/OS Debugger makes it possible to issue commands in a manner similar to each language.

The following types of z/OS Debugger commands will support the syntax of the PL/I statements:

Expression
This command evaluates an expression.

Block
`BEGIN/'END, DO/'END, PROCEDURE/'END`

These commands provide a means of grouping any number of z/OS Debugger commands into "one" command.

Conditional
`IF/'THEN, SELECT/'WHEN/'END`

© Copyright IBM Corp. 1992, 2023

These commands evaluate an expression and control the flow of execution of z/OS Debugger commands according to the resulting value.

Declaration
DECLARE or DCL

These commands provide a means for declaring session variables.

Looping
DO,'WHILE,'UNTIL,'END

These commands provide a means to program an iterative or conditional loop as a z/OS Debugger command.

Transfer of Control
GOTO, ON

These commands provide a means to unconditionally alter the flow of execution of a group of commands.

The table below shows the commands that are new or changed for this release of z/OS Debugger when the current programming language is PL/I.

Command	Description or changes
ANALYZE	Displays the PL/I style of evaluating an expression, and the precision and scale of the final and intermediate results. z/OS Debugger does not support this command for Enterprise PL/I programs.
ON	Performs as the AT OCCURRENCE command except it takes PL/I conditions as operands.
BEGIN	BEGIN/END blocks of logic.
DECLARE	Session variables can now include COMPLEX (CPLX), POINTER, BIT, BASED, ALIGNED, UNALIGNED, etc. Arrays can be declared to have upper and lower bounds. Variables can have precisions and scales. You cannot declare arrays and structures when you debug Enterprise PL/I programs.
DO	The three forms of DO are added; one is an extension of C's do. 1. DO; command(s); END; 2. DO WHILE | UNTIL expression; command(s); END; 3. DO reference=specifications; command(s); END;
IF	The IF ,' ELSE does not require the ENDIF.
SELECT	The SELECT ,' WHEN ,' OTHERWISE ,' END programming structure is added.

%PATHCODE values for PL/I

The table below shows the possible values for the z/OS Debugger variable %PATHCODE when the current programming language is PL/I.

Value	Description
0	An attention interrupt occurred.
1	A block has been entered.
2	A block is about to be exited.
3	Control has reached a label constant.

Value	Description
4	Control is being sent somewhere else as the result of a CALL or a function reference.
5	Control is returning from a CALL invocation or a function reference. Register 15, if it contains a return code, has not yet been stored.
6	Some logic contained in a complex DO statement is about to be executed.
7	The logic following an IF..THEN is about to be executed.
8	The logic following an ELSE is about to be executed.
9	The logic following a WHEN within a *select-group* is about to be executed.
10	The logic following an OTHERWISE within a *select-group* is about to be executed.

PL/I conditions and condition handling

All PL/I conditions are recognized by z/OS Debugger. They are used with the AT OCCURRENCE and ON commands.

When an OCCURRENCE breakpoint is triggered, the z/OS Debugger %CONDITION variable holds the following values:

Triggered condition	%CONDITION value
AREA	AREA
ATTENTION	CEE35J
COND (CC#1)	CONDITION
CONVERSION	CONVERSION
ENDFILE (MF)	ENDFILE
ENDPAGE (MF)	ENDPAGE
ERROR	ERROR
FINISH	CEE066
FOFL	CEE348
KEY (MF)	KEY
NAME (MF)	NAME
OVERFLOW	CEE34C
PENDING (MF)	PENDING
RECORD (MF)	RECORD
SIZE	SIZE
STRG	STRINGRANGE
STRINGSIZE	STRINGSIZE
SUBRG	SUBSCRIPTRANGE
TRANSMIT (MF)	TRANSMIT
UNDEFINEDFILE (MF)	UNDEFINEDFILE
UNDERFLOW	CEE34D

Triggered condition	%CONDITION value
ZERODIVIDE	CEE349

Note: For Enterprise PL/I programs, the following condition is not supported:

- AT OCCURRENCE CONDITION conditions (name)

Note: The z/OS Debugger condition ALLOCATE raises the ON ALLOCATE condition when a PL/I program encounters an ALLOCATE statement for a controlled variable.

These PL/I language-oriented commands are only a subset of all the commands that are supported by z/OS Debugger.

Entering commands in PL/I DBCS freeform format

Statements can be entered in PL/I's DBCS freeform. This means that statements can freely use shift codes provided that the statement is not ambiguous.

This will change the description or characteristics of LIST NAMES in that:

```
LIST NAMES  db<.c.skk.w>ord
```

will search for

```
<.D.B.C.Skk.W.O.R.D>
```

This will result in different behavior depending upon the language. For example, the following will find a<kk>b in C and <.Akk.b> in PL/I.

```
LIST NAMES a<kk>*
```

where <kk> is shiftout-kanji-shiftin.

Freeform will be added to the parser and will be in effect while the current programming language is PL/I.

Initializing z/OS Debugger for PL/I programs when TEST(ERROR, ...) run-time option is in effect

With the run-time option, TEST(ERROR, ...) only the following can initialize z/OS Debugger:

- The ERROR condition
- Attention recognition
- CALL PLITEST
- CALL CEETEST

z/OS Debugger enhancements to LIST STORAGE PL/I command

LIST STORAGE address has been enhanced so that the address can be a POINTER, a Px constant, or the ADDR built-in function.

PL/I support for z/OS Debugger session variables

PL/I will support all z/OS Debugger scalar session variables. In addition, arrays and structures can be declared.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using session variables across different programming languages" on page 377

Accessing PL/I program variables

z/OS Debugger obtains information about a program variable by name using information that is contained in the symbol table built by the compiler. The symbol table is made available to the compiler by compiling with TEST(SYM).

z/OS Debugger uses the symbol table to obtain information about program variables, controlled variables, automatic variables, and program control constants such as file and entry constants and also CONDITION condition names. Based variables, controlled variables, automatic variables and parameters can be used with z/OS Debugger only after storage has been allocated for them in the program. An exception to this is DESCRIBE ATTRIBUTES, which can be used to display attributes of a variable.

Variables that are based on any of the following data types must be explicitly qualified when used in expressions:

- an OFFSET variable
- an expression
- a pointer that is either BASED or DEFINED
- a parameter
- a member of either an array or a structure
- an address of a member of either an array or a structure

For example, assume you made the following declaration:

```
DECLARE P1 POINTER;
DECLARE P2 POINTER BASED(P1);
DECLARE DX FIXED BIN(31) BASED(P2);
```

You would not be able to reference the variable directly by name. You can only reference it by specifying either:

```
P2->DX
  or
P1->P2->DX
```

The following types of program variables cannot be used with z/OS Debugger:

- iSUB defined variables
- Variables defined:
  - On a controlled variable
  - On an array with one or more adjustable bounds
  - With a POSITION attributed that specifies something other than a constant
- Variables that are members of a based structure declared with the REFER options.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Choosing TEST or NOTEST compiler suboptions for PL/I programs” on page 34

Accessing PL/I structures

The examples in this topic assume the following declaration for a structure called PAYROLL:

```
Declare 1 Payroll(100),
          2 Name,
            4 Last       char(20),
            4 First      char(15),
          2 Hours,
            4 Regular    Fixed Decimal(5,2),
            4 Overtime   Fixed Decimal(5,2);
```

To display the 10th element in the array, enter the following command:

```
LIST PAYROLL(10);
```

z/OS Debugger displays the following results:

```
LIST PAYROLL ( 10 );
PAYROLL.NAME.LAST(10)='Johnson             '
PAYROLL.NAME.FIRST(10)='Eric                '
PAYROLL.HOURS.REGULAR(10)='40'
PAYROLL.HOURS.OVERTIME(10)='0'
```

To display the first and last name of the 31st element in the array, enter the following command:

```
LIST PAYROLL.NAME(31);
```

z/OS Debugger displays the following results:

```
LIST PAYROLL.NAME ( 31 );
PAYROLL.NAME.LAST(31)='Rivers              '
PAYROLL.NAME.FIRST(31)='Doug                '
```

To display all the elements of the array by the order of each element in the structure, enter the following command:

```
LIST PAYROLL;
```

z/OS Debugger displays results similar to the following list, with ellipses (...) used to indicate that additional information has been removed from this list to condense the list:

```
LIST PAYROLL;
PAYROLL.NAME.LAST(1)='Smith               '
PAYROLL.NAME.LAST(2)='Ramirez             '
PAYROLL.NAME.LAST(3)='Patel               '
...
PAYROLL.NAME.LAST(100)='Li                  '
PAYROLL.NAME.FIRST(1)='Jason               '
PAYROLL.NAME.FIRST(2)='Ricardo             '
PAYROLL.NAME.FIRST(3)='Aisha               '
...
PAYROLL.NAME.FIRST(100)='Xian                 '
PAYROLL.HOURS.REGULAR(1)='40'
PAYROLL.HOURS.REGULAR(2)='40'
PAYROLL.HOURS.REGULAR(3)='40'
...
PAYROLL.HOURS.REGULAR(100)='40'
PAYROLL.HOURS.OVERTIME(1)='0'
PAYROLL.HOURS.OVERTIME(2)='2'
PAYROLL.HOURS.OVERTIME(3)='3'
...
PAYROLL.HOURS.OVERTIME(100)='0'
```

To display all the elements of the array by the order in which the information is stored in memory, enter the following commands:

```
SET LIST BY SUBSCRIPT ON;
LIST PAYROLL;
```

z/OS Debugger displays results similar to the following list, with ellipses (...) used to indicate that additional information has been removed from this list to condense the list:

```
LIST PAYROLL;
PAYROLL.NAME.LAST(1)='Smith               '
PAYROLL.NAME.FIRST(1)='Jason                '
PAYROLL.HOURS.REGULAR(1)='40'
PAYROLL.HOURS.OVERTIME(1)='0'
PAYROLL.NAME.LAST(2)='Ramirez             '
PAYROLL.NAME.FIRST(2)='Ricardo              '
PAYROLL.HOURS.REGULAR(2)='40'
PAYROLL.HOURS.OVERTIME(2)='2'
PAYROLL.NAME.LAST(3)='Patel               '
```

```
PAYROLL.NAME.FIRST(3)='Aisha                '
PAYROLL.HOURS.REGULAR(3)='40'
PAYROLL.HOURS.OVERTIME(3)='3'
...
PAYROLL.NAME.LAST(100)='Li                   '
PAYROLL.NAME.FIRST(100)='Xian                 '
PAYROLL.HOURS.REGULAR(100)='40'
PAYROLL.HOURS.OVERTIME(100)='0'
```

z/OS Debugger evaluation of PL/I expressions

When the current programming language is PL/I, expression interpretation is similar to that defined in the PL/I language, except for the PL/I language elements not supported in z/OS Debugger.

The z/OS Debugger expression is similar to the PL/I expression. If the source of the command is a variable-length record source (such as your terminal) and if the expression extends across more than one line, a continuation character (an SBCS hyphen) must be specified at the end of all but the last line.

z/OS Debugger cannot evaluate PL/I expressions until you step past the ENTRY location of the PL/I program.

All PL/I constant types are supported, plus the z/OS Debugger PX constant.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"Unsupported PL/I language elements" on page 291

Supported PL/I built-in functions

z/OS Debugger supports the following built-in functions for PL/I for MVS & VM:

ABS	CSTG[2]	LOG1	REAL
ACOS	CURRENTSTORAGE	LOG2	REPEAT
ADDR	DATAFIELD	LOW	SAMEKEY
ALL	DATE	MPSTR	SIN
ALLOCATION	DATETIME	NULL	SIND
ANY	DIM	OFFSET	SINH
ASIN	EMPTY	ONCHAR	SQRT
ATAN	ENTRYADDR	ONCODE	STATUS
ATAND	ERF	ONCOUNT	STORAGE
ATANH	ERFC	ONFILE	STRING
BINARYVALUE	EXP	ONKEY	SUBSTR
BINVALUE[1]	GRAPHIC	ONLOC	SYSNULL
BIT	HBOUND	ONSOURCE	TAN
BOOL	HEX	PLIRETV	TAND
CHAR	HIGH	POINTER	TANH
COMPLETION	IMAG	POINTERADD	TIME
COS	LBOUND	POINTERVALUE	TRANSLATE
COSD	LENGTH	PTRADD[3]	UNSPEC
COSH	LINENO	PTRVALUE[4]	VERIFY
COUNT	LOG		

Note:

1. Abbreviation for BINARYVALUE
2. Abbreviation for CURRENTSTORAGE
3. Abbreviation for POINTERADD
4. Abbreviation for POINTERVALUE

z/OS Debugger supports the following built-in functions for Enterprise PL/I:

ACOS
ADDR
ADDRDATA
ALLOCATION[3]
ASIN
ATAN
ATAND
ATANH
BIF_DIM
BINARYVALUE
BINVALUE
COPY[1]
COS
COSD
COSH
COUNT
DATAFIELD
DATE[1]
DATETIME[1]
DIMENSION
ENDFILE
ENTRYADDR[1,2]
ERF
ERFC
EXP
FILEOPEN
GAMMA
HBOUND
HEX
HEXIMAGE
HIGH[1]
IAND
IEOR
IOR
INDEX
INOT
ISRL
ISLL
LBOUND
LENGTH
LINENO
LOG
LOG10
LOG2
LOGGAMMA
LOW[1]
LOWER2
LOWERCASE[1]
MAXLENGTH
NULL
OFFSET
OFFSETADD
OFFSETSUBTRACT
OFFSETDIFF
OFFSETVALUE
ORDINALNAME
ORDINALPRED
ORDINALSUCC
ONCODE
ONCONDCOND
ONCHAR
ONGSOURCE
ONSOURCE
ONCONDID
ONCOUNT
ONFILE
ONKEY
ONLOC
PAGENO
POINTER
PTR
POINTERADD
PTRADD
POINTERSUBTRACT
PTRSUBTRACT
POINTERDIFF
PTRDIFF
POINTERVALUE
PTRVALUE
PLIRETV
RAISE2
REPEAT[1]
SAMEKEY
SEARCH
SEARCHR
SIN
SIND
SINH
SQRT
SUBSTR[1]
SYSNULL
TAN
TAND
TANH
TALLY
TIME[1]
TRANSLATE[1]
UNSPEC[1]
UPPERCASE[1]
VERIFY
VERIFYR

Note: Pseudovariables are not supported for the ENTRYADDR built-in function under z/OS Debugger.

z/OS Debugger does not support the following built-in functions for Enterprise PL/I:

ABS
ALL
ANY
BIT
BOOL
CHAR
COMPLETION
CSTG(2)
CURRENTSTORAGE
DEFINE STRUCTURE
EMPTY
GRAPHIC
IMAG
MPSTR
REAL
STATUS
STORAGE
STRING

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Using SET WARNING PL/I command with built-in functions” on page 290

Using SET WARNING PL/I command with built-in functions

Certain checks are performed when the z/OS Debugger SET WARNING command setting is ON and a built-in function (BIF) is evaluated:

- Division by zero
- The remainder (%) operator for a zero value in the second operand

- Array subscript out of bounds for defined arrays
- Bit shifting by a number that is negative or greater than 32
- On a built-in function call for an incorrect number of parameters or for parameter type mismatches
- On a built-in function call for differing linkage calling conventions

These checks are restrictions that can be removed by issuing `SET WARNING OFF`.

Unsupported PL/I language elements

The following list summarizes PL/I functions not available:

- Use of `iSUB`
- Interactive declaration or use of user-defined functions
- All preprocessor directives
- Multiple assignments
- `BY NAME` assignments
- `LIKE` attribute
- `FILE`, `PICTURE`, and `ENTRY` data attributes
- All I/O statements, including `DISPLAY`
- `INIT` attribute
- Structures with the built-in functions `CSTG`, `CURRENTSTORAGE`, and `STORAGE`
- The repetition factor is not supported for string constants
- `GRAPHIC` string constants are not supported for expressions involving other data types
- Declarations cannot be made as sub-commands (for example in a `BEGIN`, `DO`, or `SELECT` command group)

Debugging OS PL/I programs

There are restrictions on how you can debug OS PL/I programs, which are described in *Using CODE/370 with VS COBOL II and OS PL/I*, SC09-1862-01.

The OS PL/I compiler does not place the name of the listing data set in the object (load module). z/OS Debugger tries to find the listing data set in the following location: `userid.CUName.LIST`. If the listing is in a PDS, direct z/OS Debugger to the location of the PDS in one of the following ways:

- In full-screen mode, enter the following command:

```
SET DEFAULT LISTINGS my.listing.pds
```

- Use the EQADEBUG DD statement to define the location of the data set.
- Code the EQAUEDAT user exit with the location of the data set.

Restrictions while debugging Enterprise PL/I programs

While debugging Enterprise PL/I programs, you cannot use the following commands:

- `ANALYZE`
- `AT ALLOCATE` (of a controlled variable)
- `AT OCCURRENCE CONDITION` conditions (name)
- `GOTO LABEL`

While debugging Enterprise PL/I programs, the following restrictions apply:

- If you are running any version of VisualAge PL/I or Enterprise PL/I Version 3 Release 1 through Version 3 Release 3 programs, you cannot use the `AT LABEL` command.

- If you are running Enterprise PL/I for z/OS, Version 3.4, or later, programs and you comply with the following requirements, you can use the `AT LABEL` command to set breakpoints (except at a label variable):
  - If you are compiling with Enterprise PL/I Version 3 Release 4, apply PTFs for APARs PK00118 and PK00339.
- For expressions, you cannot do either of the following:
  - preface variables with the block, CU, and load module qualification
  - Reference or list at the block entry
- You cannot use some of built-in functions. See "Supported PL/I built-in functions" on page 289 for more information.
- You cannot use the `DECLARE` command to declare arrays, structures, or multiple variables in one line
- The `SET WARNING ON` command has no effect.
- For typed structures, the following restrictions apply:

  z/OS Debugger does not support the debugging of PL/I typed structures for procedures compiled with the Enterprise PL/I V4R1 or earlier compiler releases. A typed structure is a variable or structure that is declared as TYPE *X*, where *X* is declared using DEFINE STRUCTURE.

  z/OS Debugger supports the debugging of PL/I typed structures for procedures compiled with the Enterprise PL/I V4R2 or later compilers. You can use the TEST (SEPARATE) options at compile time to get the full benefit of this support.

  With a few exceptions, references to typed structures require the qualified name of an elementary member. For nested typed structures, any parent that has a type reference in its declaration must be included in the qualification. References to the structure type or references that are qualified to an intermediate level of a typed structure cannot be resolved. (See the *Enterprise PL/I Language Reference Manual* for more information about typed structures.)

  Typed structure references are supported for the following:

  - ASSIGNMENT:
    - A typed structure that is assigned to a typed structure of the same type
    - A handle that is assigned to a handle declared as the same type
    - A value that is assigned to an elementary member of a typed structure
  - COMPARISONS
  - AUTOMONITOR
  - DESCRIBE ATTRIBUTES
  - LIST
  - LIST STORAGE()

Chapter 33. Debugging C and C++ programs

The topics below describe how to use z/OS Debugger to debug your C and C++ programs.

“Example: referencing variables and setting breakpoints in C and C++ blocks” on page 305

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts

“C and C++ expressions” on page 296
“z/OS Debugger evaluation of C and C++ expressions” on page 300
“Scope of objects in C and C++” on page 303
“Blocks and block identifiers for C” on page 304
“Blocks and block identifiers for C++” on page 305
“Monitoring storage in C++” on page 312

Related tasks

Chapter 25, “Debugging a C program in full-screen mode,” on page 221
Chapter 26, “Debugging a C++ program in full-screen mode,” on page 231
“Using C and C++ variables with z/OS Debugger” on page 294
“Declaring session variables with C and C++” on page 296
“Calling C and C++ functions from z/OS Debugger” on page 298
“Intercepting files when debugging C and C++ programs” on page 301
“Displaying environmental information for C and C++ programs” on page 307
“Stepping through C++ programs” on page 309
“Setting breakpoints in C++” on page 310
“Examining C++ objects” on page 311
“Qualifying variables in C and C++” on page 307

Related references

“z/OS Debugger commands that resemble C and C++ commands” on page 293
“%PATHCODE values for C and C++” on page 295
“C reserved keywords” on page 299
“C operators and operands” on page 299
“Language Environment conditions and their C and C++ equivalents” on page 300

z/OS Debugger commands that resemble C and C++ commands

z/OS Debugger's command language is a subset of C and C++ commands and has the same syntactical requirements. z/OS Debugger allows you to work in a language you are familiar with so learning a new set of commands is not necessary.

The table below shows the interpretive subset of C and C++ commands recognized by z/OS Debugger.

Command	Description
block ({})	Composite command grouping
break	Termination of loops or `switch` commands
declarations	Declaration of session variables
do/while	Iterative looping
expression	Any C expression except the conditional (?) operator
for	Iterative looping

© Copyright IBM Corp. 1992, 2023

Command	Description
if	Conditional execution
switch	Conditional execution

This subset of commands is valid only when the current programming language is C or C++.

In addition to the subset of C and C++ commands that you can use is a list of reserved keywords used and recognized by C and C++ that you cannot abbreviate, use as variable names, or use as any other type of identifier.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
"C reserved keywords" on page 299
z/OS XL C/C++ Language Reference

Using C and C++ variables with z/OS Debugger

z/OS Debugger can process all program variables that are valid in C or C++. You can assign and display the values of variables during your session. You can also declare session variables with the recognized C declarations to suit your testing needs.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Accessing C and C++ program variables" on page 294
"Displaying values of C and C++ variables or expressions" on page 294
"Assigning values to C and C++ variables" on page 295

Accessing C and C++ program variables

z/OS Debugger obtains information about a program variable by name using the symbol table built by the compiler. If you specify TEST(SYM) at compile time, the compiler builds a symbol table that allows you to reference any variable in the program.

Note: There are no suboptions for C++. Symbol information is generated by default when the TEST compiler option is specified.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
"Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
"Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44

Displaying values of C and C++ variables or expressions

To display the values of variables or expressions, use the LIST command. The LIST command causes z/OS Debugger to log and display the current values (and names, if requested) of variables, including the evaluated results of expressions.

Suppose you want to display the program variables X, row[X], and col[X], and their values at line 25. If you issue the following command:

```
AT 25 LIST ( X, row[X], col[X] ); GO;
```

z/OS Debugger sets a breakpoint at line 25 (AT), begins execution of the program (GO), stops at line 25, and displays the variable names and their values.

If you want to see the result of their addition, enter:

```
AT 25 LIST ( X + row[X] + col[X] ); GO;
```

z/OS Debugger sets a breakpoint at line 25 (AT), begins execution of the program (GO), stops at line 25, and displays the result of the expression.

Put commas between the variables when listing more than one. If you do not want to display the variable names when issuing the LIST command, enter `LIST UNTITLED`.

You can also list variables with the `printf` function call as follows:

```
printf ("X=%d, row=%d, col=%d\n", X, row[X], col[X]);
```

The output from `printf`, however, does not appear in the Log window and is not recorded in the log file unless you `SET INTERCEPT ON FILE stdout`.

Assigning values to C and C++ variables

To assign a value to a C and C++ variable, you use an assignment expression. Assignment expressions assign a value to the left operand. The left operand must be a modifiable lvalue. An lvalue is an expression representing a data object that can be examined and altered.

C contains two types of assignment operators: simple and compound. A simple assignment operator gives the value of the right operand to the left operand.

Note: Only the assignment operators that work for C will work for C++, that is, there is no support for overloaded operators.

The following example demonstrates how to assign the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

Compound assignment operators perform an operation on both operands and give the result of that operation to the left operand. For example, this expression gives the value of `index plus 2` to the variable `index`:

```
index += 2
```

z/OS Debugger supports all C operators except the tenary operator, as well as any other full C language assignments and function calls to user or C library functions.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Calling C and C++ functions from z/OS Debugger” on page 298

%PATHCODE values for C and C++

The table below shows the possible values for the z/OS Debugger variable %PATHCODE when the current programming language is C and C++.

Value	Description
–1	z/OS Debugger is not in control as the result of a path or attention situation.
0	`Attention` function (*not* ATTENTION condition).
1	A block has been entered.
2	A block is about to be exited.
3	Control has reached a user label.
4	Control is being transferred as a result of a function reference. The invoked routine's parameters, if any, have been prepared.

Value	Description
5	Control is returning from a function reference. Any return code contained in register 15 has not yet been stored.
6	Some logic contained by a conditional `do`/`while`, `for`, or `while` statement is about to be executed. This can be a single or `Null` statement and not a block statement.
7	The logic following an `if(...)` is about to be executed.
8	The logic following an `else` is about to be executed.
9	The logic following a `case` within an `switch` is about to be executed.
10	The logic following a default within a `switch` is about to be executed.
13	The logic following the end of a `switch`, `do`, `while`, `if(...)`, or `for` is about to be executed.
17	A `goto`, `break`, `continue`, or `return` is about to be executed.

Values in the range 3–17 can only be assigned to %PATHCODE if your program was compiled with an option supporting path hooks.

Declaring session variables with C and C++

You might want to declare session variables for use during the course of your session. You cannot initialize session variables in declarations. However, you can use an assignment statement or function call to initialize a session variable.

As in C, keywords can be specified in any order. Variable names up to 255 characters in length can be used. Identifiers are case-sensitive, but if you want to use the session variable when the current programming language changes from C to another HLL, the variable must have an uppercase name and compatible attributes.

To declare a hexadecimal floating-point variable called `maximum`, enter the following C declaration:

```
double maximum;
```

You can only declare scalars, arrays of scalars, structures, and unions in z/OS Debugger (pointers for the above are allowed as well).

If you declare a session variable with the same name as a programming variable, the session variable hides the programming variable. To reference the programming variable, you must qualify it. For example:

```
main:>x for the program variable x
x for the session variable x
```

Session variables remain in effect for the entire debug session, unless they are cleared using the CLEAR command.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Using session variables across different programming languages” on page 377
“Qualifying variables and changing the point of view in C and C++” on page 307

C and C++ expressions

z/OS Debugger allows evaluation of expressions in your test program. All expressions available in C and C++ are also available within z/OS Debugger except for the conditional expression (`? :`). That is, all operators such as `+`, `-`, `%:`, and `+=` are fully supported with the exception of the conditional operator.

C and C++ language expressions are arranged in the following groups based on the operators they contain and how you use them:

Primary expression
Unary expression
Binary expression
Conditional expression
Assignment expression
Comma expression
lvalue
Constant

An lvalue is an expression representing a data object that can be examined and altered. For a more detailed description of expressions and operators, see the C and C++ Program Guides.

The semantics for C and C++ operators are the same as in a compiled C or C++ program. Operands can be a mixture of constants (`integer`, `floating-point`, `character`, `string`, and `enumeration`), C and C++ variables, z/OS Debugger variables, or session variables declared during a z/OS Debugger session. Language constants are specified as described in the C and C++ Language Reference publications.

The z/OS Debugger command `DESCRIBE ATTRIBUTES` can be used to display the resultant type of an expression, without actually evaluating the expression.

The C and C++ language does not specify the order of evaluation for function call arguments. Consequently, it is possible for an expression to have a different execution sequence in compiled code than within z/OS Debugger. For example, if you enter the following in an interactive session:

```
int x;
int y;

x = y = 1;

printf ("%d %d %d%" x, y, x=y=0);
```

the results can differ from results produced by the same statements located in a C or C++ program segment. Any expression containing behavior undefined by ANSI standards can produce different results when evaluated by z/OS Debugger than when evaluated by the compiler.

The following examples show you various ways z/OS Debugger supports the use of expressions in your programs:

- z/OS Debugger assigns 12 to a (the result of the `printf()`) function call, as in:

```
a = (1,2,3,a++,b++,printf("hello world\n"));
```

- z/OS Debugger supports structure and array referencing and pointer dereferencing, as in:

```
league[num].team[1].player[1]++;
league[num].team[1].total += 1;
++(*pleague);
```

- Simple and compound assignment is supported, as in:

```
v.x = 3;
a = b = c = d = 0;
*(pointer++) -= 1;
```

- C and C++ language constants in expressions can be used, as in:

```
*pointer_to_long = 3521L = 0x69a1;
float_val = 3e-11 + 6.6E-10;
char_val = '7';
```

- The comma expression can be used, as in:

```
intensity <<= 1, shade * increment, rotate(direction);
alpha = (y>>3, omega % 4);
```

- z/OS Debugger performs all implicit and explicit C conversions when necessary. Conversion to long double is performed in:

```
long_double_val = unsigned_short_val;
long_double_val = (long double) 3;
```

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"z/OS Debugger evaluation of C and C++ expressions" on page 300

z/OS XL C/C++ Language Reference

Calling C and C++ functions from z/OS Debugger

You can perform calls to user and C library functions within z/OS Debugger, unless your program was compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option.

You can make calls to C library functions at any time. In addition, you can use the C library variables `stdin`, `stdout`, `stderr`, `__amrc`, and `errno` in expressions including function calls.

The library function `ctdli` cannot be called unless it is referenced in a compile unit in the program, either `main` or a function linked to `main`.

Calls to user functions can be made, provided z/OS Debugger is able to locate an appropriate definition for the function within the symbol information in the user program. These definitions are created when the program is compiled with TEST(SYM) for C or TEST for C++.

z/OS Debugger performs parameter conversions and parameter-mismatch checking where possible. Parameter checking is performed if:

- The function is a library function
- A prototype for the function exists in the current compile unit
- z/OS Debugger is able to locate a prototype for the function in another compile unit, or the function itself was compiled with TEST(SYM) for C or with TEST for C++.

You can turn off this checking by specifying `SET WARNING OFF`.

Calls can be made to any user functions that have linkage supported by the C or C++ compiler. However, for C++ calls made to any user function, the function must be declared as:

```
extern "C"
```

For example, use this declaration if you want to debug an application signal handler. When a condition occurs, control passes to z/OS Debugger which then passes control to the signal handler.

z/OS Debugger attempts linkage checking, and does not perform the function call if it determines there is a linkage mismatch. A linkage mismatch occurs when the target program has one linkage but the source program believes it has a different linkage.

It is important to note the following regarding function calls:

- The evaluation order of function arguments can vary between the C and C++ program and z/OS Debugger. No discernible difference exists if the evaluation of arguments does not have side effects.
- z/OS Debugger knows about the function return value, and all the necessary conversions are performed when the return value is used in an expression.
- The functions cannot be in XPLINK applications.
- The functions must have debug information available.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
"Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44

Related references

z/OS XL C/C++ Language Reference

C reserved keywords

The table below lists all keywords reserved by the C language. When the current programming language is C or C++, these keywords cannot be abbreviated, used as variable names, or used as any other type of identifiers.

`auto`	`else`	`long`	`switch`
`break`	`enum`	`register`	`typedef`
`case`	`extern`	`return`	`union`
`char`	`float`	`short`	`unsigned`
`const`	`for`	`signed`	`void`
`continue`	`goto`	`sizeof`	`volatile`
`default`	`if`	`static`	`while`
`do`	`int`	`struct`	`_Packed`
`double`			

C operators and operands

The table below lists the C language operators in order of precedence and shows the direction of associativity for each operator. The primary operators have the highest precedence. The comma operator has the lowest precedence. Operators in the same group have the same precedence.

Precedence level	Associativity	Operators
Primary	left to right	`() [ ] . ->`
Unary	right to left	`++ -- - + ! ~ & * (typename) sizeof`
Multiplicative	left to right	`* / %`
Additive	left to right	`+ -`
Bitwise shift	left to right	`<< >>`
Relational	left to right	`< > <= >=`
Equality	left to right	`== !=`
Bitwise logical AND	left to right	`&`
Bitwise exclusive OR	left to right	`^` or `¬`
Bitwise inclusive OR	left to right	`|`
Logical AND	left to right	`&&`
Logical OR	left to right	`||`
Assignment	right to left	`= += -= *= /= <<= >>= %= &= ^= |=`

Precedence level	Associativity	Operators
Comma	left to right	,

Language Environment conditions and their C and C++ equivalents

Language Environment condition names (the symbolic feedback codes CEE*xxx*) can be used interchangeably with the equivalent C and C++ conditions listed in the following table. For example, AT OCCURRENCE CEE341 is equivalent to AT OCCURRENCE SIGILL. Raising a CEE341 condition triggers an AT OCCURRENCE SIGILL breakpoint and vice versa.

Language Environment condition	Description	Equivalent C/C++ condition
CEE341	Operation exception	SIGILL
CEE342	Privileged operation exception	SIGILL
CEE343	Execute exception	SIGILL
CEE344	Protection exception	SIGSEGV
CEE345	Addressing exception	SIGSEGV
CEE346	Specification exception	SIGILL
CEE347	Data exception	SIGFPE
CEE348	Fixed point overflow exception	SIGFPE
CEE349	Fixed point divide exception	SIGFPE
CEE34A	Decimal overflow exception	SIGFPE
CEE34B	Decimal divide exception	SIGFPE
CEE34C	Exponent overflow exception	SIGFPE
CEE34D	Exponent underflow exception	SIGFPE
CEE34E	Significance exception	SIGFPE
CEE34F	Floating-point divide exception	SIGFPE

z/OS Debugger evaluation of C and C++ expressions

z/OS Debugger interprets most input as a collection of one or more expressions. You can use expressions to alter a program variable or to extend the program by adding expressions at points that are governed by AT breakpoints.

z/OS Debugger evaluates C and C++ expressions following the rules presented in *z/OS XL C/C++ Language Reference*. The result of an expression is equal to the result that would have been produced if the same expression had been part of your compiled program.

Implicit string concatenation is supported. For example, "abc" "def" is accepted for "abcdef" and treated identically. Concatenation of wide string literals to string literals is not accepted. For example, L"abc"L"def" is valid and equivalent to L"abcdef", but "abc" L"def" is not valid.

Expressions you use during your session are evaluated with the same sensitivity to enablement as are compiled expressions. Conditions that are enabled are the same ones that exist for program statements.

During a z/OS Debugger session, if the current setting for WARNING is ON, the occurrence in your C or C++ program of any one of the conditions listed below causes the display of a diagnostic message.

- Division by zero

- Remainder (%) operator for a zero value in the second operand
- Array subscript out of bounds for a defined array
- Bit shifting by a number that is either negative or greater than 32
- Incorrect number of parameters, or parameter type mismatches for a function call
- Differing linkage calling conventions for a function call
- Assignment of an integer value to a variable of enumeration data type where the integer value does not correspond to an integer value of one of the enumeration constants of the enumeration data type
- Assignment to an lvalue that has the `const` attribute
- Attempt to take the address of an object with `register` storage class
- A signed integer constant not in the range -2**31 to 2**31
- A real constant not having an exponent of 3 or fewer digits
- A float constant not larger than 5.39796053469340278908664699142502496E-79 or smaller than 7.23700557733226221397318656304299 29E+75
- A hex escape sequence that does not contain at least one hexadecimal digit
- An octal escape sequence with an integer value of 256 or greater
- An unsigned integer constant greater than the maximum value of 4294967295.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
"C and C++ expressions" on page 296
z/OS XL C/C++ Language Reference

Intercepting files when debugging C and C++ programs

Several considerations must be kept in mind when using the `SET INTERCEPT` command to intercept files while you are debugging a C application.

For CICS only: `SET INTERCEPT` is not supported for CICS.

For C++, there is no specific support for intercepting IOStreams. IOStreams is implemented using C I/O which implies that:

- If you intercept I/O for a C standard stream, this implicitly intercepts I/O for the corresponding IOStreams' standard stream.
- If you intercept I/O for a file, by name, and define an IOStream object associated with the same file, IOStream I/O to that file will be intercepted.

Note: Although you can intercept IOStreams indirectly via C/370 I/O, the behaviors might be different or undefined in C++.

You can use the following names with the `SET INTERCEPT` command during a debug session:

- `stdout`, `stderr`, and `stdin` (lowercase only)
- any valid `fopen()` file specifier.

The behavior of I/O interception across `system()` call boundaries is global. This implies that the setting of `INTERCEPT ON for xx` in Program A is also in effect for Program B (when Program A `system()` calls to Program B). Correspondingly, setting INTERCEPT OFF for `xx` in Program B turns off interception in Program A when Program B returns to A. This is also true if a file is intercepted in Program B and returns to Program A. This model applies to disk files, memory files, and standard streams.

When a stream is intercepted, it inherits the text/binary attribute specified on the `fopen` statement. The output to and input from the z/OS Debugger log file behaves like terminal I/O, with the following considerations:

- Intercepted input behaves as though the terminal was opened for record I/O. Intercepted input is truncated if the data is longer than the record size and the truncated data is not available to subsequent reads.
- Intercepted output is not truncated. Data is split across multiple lines.
- Some situations causing an error with the real file might not cause an error when the file is intercepted (for example, truncation errors do not occur). Files expecting specific error conditions do not make good candidates for interception.
- Only sequential I/O can be performed on an intercepted stream, but file positioning functions are tolerated and the real file position is not changed. `fseek`, `rewind`, `ftell`, `fgetpos`, and `fsetpos` do not cause an error, but have no effect.
- The logical record length of an intercepted stream reflects the logical record length of the real file.
- When an unintercepted memory file is opened, the record format is always fixed and the open mode is always binary. These attributes are reflected in the intercepted stream.
- Files opened to the terminal for write are flushed before an input operation occurs from the terminal. This is not supported for intercepted files.

Other characteristics of intercepted files are:

- When an `fclose()` occurs or `INTERCEPT` is set `OFF` for a file that was intercepted, the data is flushed to the session log file before the file is closed or the `SET INTERCEPT OFF` command is processed.
- When an `fopen()` occurs for an intercepted file, an open occurs on the real file before the interception takes effect. If the `fopen()` fails, no interception occurs for that file and any assumptions about the real file, such as the ddname allocation and data set defaults, take effect.
- The behavior of the `ASIS` suboption on the `fopen()` statement is not supported for intercepted files.
- When the `clrmemf()` function is invoked and memory files have been intercepted, the buffers are flushed to the session log file before the files are removed.
- If the `fldata()` function is invoked for an intercepted file, the characteristics of the real file are returned.
- If `stderr` is intercepted, the interception overrides the Language Environment message file (the default destination for `stderr`). A subsequent `SET INTERCEPT OFF` command returns `stderr` to its `MSGFILE` destination.
- If a file is opened with a ddname, interception occurs only if the ddname is specified on the `INTERCEPT` command. Intercepting the underlying file name does not cause interception of the stream.
- User prefix qualifications are included in MVS data set names entered in the `INTERCEPT` command, using the same rules as defined for the `fopen()` function.
- If library functions are invoked when z/OS Debugger is waiting for input for an intercepted file (for example, if you interactively enter `fwrite(..)` when z/OS Debugger is waiting for input), subsequent behavior is undefined.
- I/O intercepts remain in effect for the entire debug session, unless you terminate them by selecting `SET INTERCEPT OFF`.

Command line redirection of the standard streams is supported under z/OS Debugger, as shown below.

1>&2
If `stderr` is the target of the interception command, `stdout` is also intercepted. If `stdout` is the target of the `INTERCEPT` command, `stderr` is not intercepted. When `INTERCEPT` is set `OFF` for `stdout`, the stream is redirected to `stderr`.

2>&1
If `stdout` is the target of the `INTERCEPT` command, `stderr` is also intercepted. If `stderr` is the target of the `INTERCEPT` command, `stdout` is not intercepted. When `INTERCEPT` is set `OFF` for `stderr`, the stream is redirected to `stdout` again.

1>file.name
`stdout` is redirected to **file.name**. For interception of `stdout` to occur, `stdout` or **file.name** can be specified on the interception request. This also applies to **1>>file.name**

2>file.name
`stderr` is redirected to `file.name`. For interception of `stderr` to occur, `stderr` or **file.name** can be specified on the interception request. This also applies to **2>>file.name**

2>&1 1>file.name
`stderr` is redirected to `stdout`, and both are redirected to **file.name**. If `file.name` is specified on the interception command, both `stderr` and `stdout` are intercepted. If you specify `stderr` or `stdout` on the `INTERCEPT` command, the behavior follows rule 1b above.

1>&2 2>file.name
`stdout` is redirected to `stderr`, and both are redirected to **file.name**. If you specify **file.name** on the `INTERCEPT` command, both `stderr` and `stdout` are intercepted. If you specify `stdout` or `stderr` on the `INTERCEPT` command, the behavior follows rule 1a above.

The same standard stream cannot be redirected twice on the command line. Interception is undefined if this is violated, as shown below.

2>&1 2>file.name
Behavior of `stderr` is undefined.

1>&2 1>file.name
Behavior of `stdout` is undefined.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS XL C/C++ Programming Guide

Scope of objects in C and C++

An object is *visible* in a block or source file if its data type and declared name are known within the block or source file. The region where an object is visible is referred to as its scope. In z/OS Debugger, an object can be a variable or function and is also used to refer to line numbers.

Note: The use of an object here is not to be confused with a C++ object. Any reference to C++ will be qualified as such.

In ANSI C, the four kinds of scope are:

Block
File
Function
Function prototype

For C++, in addition to the scopes defined for C, it also has the class scope.

An object has block scope if its declaration is located inside a block. An object with block scope is visible from the point where it is declared to the closing brace (}) that terminates the block.

An object has file scope if its definition appears outside of any block. Such an object is visible from the point where it is declared to the end of the source file. In z/OS Debugger, if you are qualified to the compilation unit with the file static variables, file static and global variables are always visible.

The only type of object with function scope is a label name.

An object has function prototype scope if its declaration appears within the list of parameters in a function prototype.

A class member has class scope if its declaration is located inside a class.

You cannot reference objects that are visible at function prototype scope, but you can reference ones that are visible at file or block scope if:

- For C variables and functions, the source file was compiled with TEST(SYM) and the object was referenced somewhere within the source.

- For C variables declared in a block that is nested in another block, the source file was compiled with `TEST(SYM, BLOCK)`.
- For line numbers, the source file was compiled with `TEST(LINE) GONUMBER`.
- For labels, the source file was compiled with `TEST(SYM, PATH)`. In some cases (for example, when using `GOTO`), labels can be referenced if the source file was compiled with `TEST(SYM, NOPATH)`.

z/OS Debugger follows the same scoping rules as ANSI, except that it handles objects at file scope differently. An object at file scope can be referenced from within z/OS Debugger at any point in the source file, not just from the point in the source file where it is declared. z/OS Debugger session variables always have a higher scope than program variables, and consequently have higher precedence than a program variable with the same name. The program variable can always be accessed through qualification.

In addition, z/OS Debugger supports the referencing of variables in multiple load modules. Multiple load modules are managed through the C library functions `dllload()`, `dllfree()`, `fetch()`, and `release()`.

"Example: referencing variables and setting breakpoints in C and C++ blocks" on page 305

Related concepts

"Storage classes in C and C++" on page 304

Storage classes in C and C++

z/OS Debugger supports the change and reference of all objects declared with the following storage classes:

```
auto
register
static
extern
```

Session variables declared during the z/OS Debugger session are also available for reference and change.

An object with `auto` storage class is available for reference or change in z/OS Debugger, provided the block where it is defined is active. Once a block finishes executing, the `auto` variables within this block are no longer available for change, but can still be examined using `DESCRIBE ATTRIBUTES`.

An object with `register` storage class might be available for reference or change in z/OS Debugger, provided the variable has not been optimized to a register.

An object with `static` storage class is always available for change or reference in z/OS Debugger. If it is not located in the currently qualified compile unit, you must specifically qualify it.

An object with `extern` storage class is always available for change or reference in z/OS Debugger. It might also be possible to reference such a variable in a program even if it is not defined or referenced from within this source file. This is possible provided z/OS Debugger can locate another compile unit (compiled with `TEST(SYM)`) with the appropriate definition.

Blocks and block identifiers for C

It is often necessary to set breakpoints on entry into or exit from a given block or to reference variables that are not immediately visible from the current block. z/OS Debugger can do this, provided that all blocks are named. It uses the following naming convention:

- The outermost block of a function has the same name as the function.
- For C programs compiled with the ISD compiler option, blocks enclosed in this outermost block are sequentially named: %BLOCK2, %BLOCK3, %BLOCK4, and so on in order of their appearance in the function.
- For C programs compiled with the DWARF compiler option, blocks are named in a non-sequential manner. To determine the names of the blocks, enter the `DESCRIBE CU;` command.

When these block names are used in the z/OS Debugger commands, you might need to distinguish between nested blocks in different functions within the same source file. This can be done by naming the blocks in one of two ways:

Short form
function_name:>%BLOCK*zzz*

Long form
function_name:>%BLOCK*xxx* :>%BLOCK*yyy*: … :>%BLOCK*zzz*

%BLOCK*zzz* is contained in %BLOCK*yyy*, which is contained in %BLOCK*xxx*. The short form is always allowed; it is never necessary to specify the long form.

The currently active block name can be retrieved from the z/OS Debugger variable %BLOCK. You can display the names of blocks by entering:

```
DESCRIBE CU;
```

Blocks and block identifiers for C++

Block Identifiers tend to be longer for C++ than C because C++ functions can be overloaded. In order to distinguish one function name from the other, each block identifier is like a prototype. For example, a function named shapes(int,int) in C would have a block named `shapes`; however, in C++ the block would be called `shapes(int,int)`.

You must always refer to a C++ block identifier in its entirety, even if the function is not overloaded. That is, you cannot refer to `shapes(int,int)` as `shapes` only.

Note: The block name for `main()` is always `main` (without the qualifying parameters after it) even when compiled with C++ because `main()` has extern **C** linkage.

Since block names can be quite long, it is not unusual to see the name truncated in the `LOCATION` field on the first line of the screen. If you want to find out where you are, enter:

```
QUERY LOCATION
```

and the name will be shown in its entirety (wrapped) in the session log.

Block identifiers are restricted to a length of 255 characters. Any name longer than 255 characters is truncated.

Example: referencing variables and setting breakpoints in C and C++ blocks

The program below is used as the basis for several examples, described after the program listing.

```
#pragma runopts(EXECOPS)
#include <stdlib.h>

main()
{
      >>> z/OS Debugger is given <<<
      >>> control here.    <<<
  init();
  sort();
}

short length = 40;
static long *table;

init()
{
  table = malloc(sizeof(long)*length);
:
}

sort ()
{                                     /* Block sort */
```

```
  int i;
  for (i = 0; i < length-1; i++) {    /* If compiled with ISD, Block %BLOCK2; */
                                      /* if compiled with DWARF, Block %BLOCK8 */
    int j;
    for (j = i+1; j < length; j++) {  /* If compiled with ISD, Block %BLOCK3; */
                                      /* if compiled with DWARF, Block %BLOCK13 */
      static int temp;
      temp = table[i];
      table[i] = table[j];
      table[j] = temp;
    }
  }
}
```

Scope and visibility of objects in C and C++ programs

Let's assume the program shown above is compiled with TEST(SYM). When z/OS Debugger gains control, the file scope variables `length` and `table` are available for change, as in:

```
length = 60;
```

The block scope variables `i`, `j`, and `temp` are not visible in this scope and cannot be directly referenced from within z/OS Debugger at this time. You can list the line numbers in the current scope by entering:

```
LIST LINE NUMBERS;
```

Now let's assume the program is compiled with TEST(SYM, NOBLOCK). Since the program is explicitly compiled using NOBLOCK, z/OS Debugger will never know about the variables `j` and `temp` because they are defined in a block that is nested in another block. z/OS Debugger does know about the variable `i` since it is not in a scope that is nested.

Blocks and block identifiers in C and C++ programs

In the program above, the function `sort` has the following three blocks:

If program is compiled with the ISD compiler option	If program is compiled with the DWARF compiler option
sort	sort
%BLOCK2	%BLOCK8
%BLOCK3	%BLOCK13

The following examples set a breakpoint on entry to the second block of `sort`:

- If program is compiled with the ISD compiler option: `at entry sort:>%BLOCK2;`.
- If program is compiled with the DWARF compiler option: `at entry sort:>%BLOCK8;`.

The following example sets a breakpoint on exit of the first block of `main` and lists the entries of the sorted table.

```
at exit main {
  for (i = 0; i < length; i++)
    printf("table entry %d is %d\n", i, table[i]);
}
```

The following examples list the variable `temp` in the third block of `sort`. This is possible because `temp` has the `static` storage class.

- If program is compiled with the ISD compiler option: `LIST sort:>%BLOCK3:temp;`.
- If program is compiled with the DWARF compiler option: `LIST sort:>%BLOCK13:temp;`.

Displaying environmental information for C and C++ programs

You can also use the DESCRIBE command to display a list of attributes applicable to the current run-time environment. The type of information displayed varies from language to language.

Issuing DESCRIBE ENVIRONMENT displays a list of open files and conditions being monitored by the run-time environment. For example, if you enter DESCRIBE ENVIRONMENT while debugging a C or C++ program, you might get the following output:

```
Currently open files
    stdout
    sysprint
The following conditions are enabled:
    SIGFPE
    SIGILL
    SIGSEGV
    SIGTERM
    SIGINT
    SIGABRT
    SIGUSR1
    SIGUSR2
    SIGABND
```

Qualifying variables and changing the point of view in C and C++

Qualification is a method of:

- Specifying an object through the use of qualifiers
- Changing the point of view

Qualification is often necessary due to name conflicts, or when a program consists of multiple load modules, compile units, and/or functions.

When program execution is suspended and z/OS Debugger receives control, the default, or *implicit* qualification is the active block at the point of program suspension. All objects visible to the C or C++ program in this block are also visible to z/OS Debugger. Such objects can be specified in commands without the use of qualifiers. All others must be specified using *explicit qualification*.

Qualifiers depend, of course, upon the naming convention of the system where you are working.

"Example: using qualification in C" on page 308

Related tasks

"Qualifying variables in C and C++" on page 307
"Changing the point of view in C and C++" on page 308

Qualifying variables in C and C++

You can precisely specify an object, provided you know the following:

- Load module or DLL name
- Source file (compilation unit) name
- Block name (must include function prototype for C++ block qualification).

These are known as qualifiers and some, or all, might be required when referencing an object in a command. Qualifiers are separated by a combination of greater than signs (>) and colons and precede the object they qualify. For example, the following is a fully qualified object:

```
load_name::>cu_name:>block_name:>object
```

If required, *load_name* is the name of the load module. It is required only when the program consists of multiple load modules and when you want to change the qualification to other than the current load module. *load_name* is enclosed in quotation marks ("). If it is not, it must be a valid identifier in the C or C++ programming language. *load_name* can also be the z/OS Debugger variable %LOAD.

If required, *CU_NAME* is the name of the compilation unit or source file. The *cu_name* must be the fully qualified source file name or an absolute pathname. It is required only when you want to change the qualification to other than the currently qualified compilation unit. It can be the z/OS Debugger variable %CU. If there appears to be an ambiguity between the compilation unit name, and (for example), a block name, you must enclose the compilation unit name in quotation marks (").

If required, *block_name* is the name of the block. *block_name* can be the z/OS Debugger variable %BLOCK.

"Example: using qualification in C" on page 308

Refer to the following topics for more information related to the material discussed in this topic.

Related concepts

"Blocks and block identifiers for C" on page 304

Changing the point of view in C and C++

To change the point of view from the command line or a commands file, use qualifiers with the SET QUALIFY command. This can be necessary to get to data that is inaccessible from the current point of view, or can simplify debugging when a number of objects are being referenced.

It is possible to change the point of view to another load module or DLL, to another compilation unit, to a nested block, or to a block that is not nested. The SET keyword is optional.

"Example: using qualification in C" on page 308

Example: using qualification in C

The examples below use the following program.

```
LOAD MODULE NAME:  MAINMOD
SOURCE FILE  NAME:  MVSID.SORTMAIN.C

short length = 40;
main ()
{
  long *table;
  void (*pf)();

  table = malloc(sizeof(long)*length);
⋮
  pf = fetch("SORTMOD");
  (*pf)(table);
⋮
  release(pf);
⋮
}

LOAD MODULE NAME:  SORTMOD
SOURCE FILE  NAME:  MVSID.SORTSUB.C

short length = 40;
short sn = 3;
void (long table[])
{
  short i;
  for (i = 0; i < length-1; i++) {
    short j;
    for (j = i+1; j < length; j++) {
      float sn = 3.0;
      short temp;
      temp = table[i];
⋮
      >>> z/OS Debugger is given <<<
      >>> control here.    <<<
⋮
      table[i] = table[j];
      table[j] = temp;
    }
  }
}
```

When z/OS Debugger receives control, variables `i`, `j`, `temp`, `table`, and `length` can be specified without qualifiers in a command. If variable `sn` is referenced, z/OS Debugger uses the variable that is a `float`. However, the names of the blocks and compile units differ, maintaining compatibility with the operating system.

Qualifying variables in C

- Change the file scope variable `length` defined in the compilation unit MVSID.SORTSUB.C in the load module SORTMOD:

```
"SORTMOD"::>"MVSID.SORTSUB.C":>length = 20;
```

- Assume z/OS Debugger gained control from `main()`. The following changes the variable `length`:

```
%LOAD::>"MVSID.SORTMAIN.C":>length = 20;
```

  Because `length` is in the current load module and compilation unit, it can also be changed by:

```
length = 20;
```

- Assume z/OS Debugger gained control as shown in the example program above. You can break whenever the variable `temp` in load module SORTMOD changes in any of the following ways:

```
AT CHANGE temp;
AT CHANGE %BLOCK3:>temp;
AT CHANGE sort:%BLOCK3:>temp;
AT CHANGE %BLOCK:>temp;
AT CHANGE %CU:>sort:>%BLOCK3:>temp;
AT CHANGE "MVSID.SORTSUB.C":>sort:>%BLOCK3:>temp;
AT CHANGE "SORTMOD"::>"MVSID.SORTSUB.C":>sort:>%BLOCK3:>temp;
```

  The %BLOCK and %BLOCK3 variables in this example assume the program was compiled with the ISD compiler option. If the example was compiled with the DWARF compiler option, enter the DESCRIBE PROGRAM command to determine the correct %BLOCK variables.

Changing the point of view in C

- Qualify to the second nested block in the function `sort()` in `sort`.

```
SET QUALIFY BLOCK %BLOCK2;
```

  You can do this in a number of other ways, including:

```
QUALIFY BLOCK sort:>%BLOCK2;
```

  Once the point of view changes, z/OS Debugger has access to objects accessible from this point of view. You can specify these objects in commands without qualifiers, as in:

```
j = 3;
temp = 4;
```

- Qualify to the function `main` in the load module MAINMOD in the compilation unit MVSID.SORTMAIN.C and list the entries of `table`.

```
QUALIFY BLOCK "MAINMOD"::>"MVSID.SORTMAIN.C":>main;
LIST table[i];
```

Stepping through C++ programs

You can step through methods as objects are constructed and destructed. In addition, you can step through static constructors and destructors. These are methods of objects that are executed before and after `main()` respectively.

If you are debugging a program that calls a function that resides in a header file, the cursor moves to the applicable header file. You can then view the function source as you step through it. Once the function returns, debugging continues at the line following the original function call.

You can step around a header file function by issuing the STEP OVER command. This is useful in stepping over Library functions (for example, string functions defined in string.h) that you cannot debug anyway.

Setting breakpoints in C++

The differences between setting breakpoints in C++ and C are described below.

Setting breakpoints in C++ using AT ENTRY/EXIT

AT ENTRY/EXIT sets a breakpoint in the specified block. You can set a breakpoint on methods, methods within nested classes, templates, and overloaded operators. An example is given for each below.

A block identifier can be quite long, especially with templates, nested classes, or class with many levels of inheritance. In fact, it might not even be obvious at first as to the block name for a particular function. To set a breakpoint for these nontrivial blocks can be quite cumbersome. Therefore, it is recommended that you make use of DESCRIBE CU and retrieve the block identifier from the session log.

When you do a DESCRIBE CU, the methods are always shown qualified by their class. If a method is unique, you can set a breakpoint by using just the method name. Otherwise, you must qualify the method with its class name. The following two examples are equivalent:

```
AT ENTRY method()
AT ENTRY classname::method()
```

The following examples are valid:

Example	Description
`AT ENTRY square(int,int)`	'simple' method square
`AT ENTRY shapes::square(int)`	Method square qualified by its class shapes.
`AT EXIT outer::inner::func()`	Nested classes. Outer and inner are classes. `func()` is within class inner.
`AT EXIT Stack<int,5>::Stack()`	Templates.
`AT ENTRY Plus::operator++(int)`	Overloaded operator.
`AT ENTRY ::fail()`	Functions defined at file scope must be referenced by the global scope operator ::

The following examples are invalid:

Example	Description
`AT ENTRY shapes`	Where shapes is a class. Cannot set breakpoint on a class. (There is no block identifier for a class.)
`AT ENTRY shapes::square`	Invalid since method square must be followed by its parameter list.
`AT ENTRY shapes:>square(int)`	Invalid since shapes is a class name, not a block name.

Setting breakpoints in C++ using AT CALL

AT CALL gives z/OS Debugger control when the application code attempts to call the specified entry point. The entry name must be a fully qualified name. That is, the name shown in DESCRIBE CU must be used. Using the example

```
AT ENTRY shapes::square(int)
```

to set a breakpoint on the method square, you must enter:

```
AT CALL shapes::square(int)
```

even if square is uniquely identified.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Composing commands from lines in the Log and Source windows" on page 164

Examining C++ objects

When displaying an C++ object, only the local member variables are shown. Access types (public, private, protected) are not distinguished among the variables. The member functions are not displayed. If you want to see their attributes, you can display them individually, but not in the context of a class. When displaying a derived class, the base class within it is shown as type class and will not be expanded.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Example: displaying attributes of C++ objects" on page 311

Example: displaying attributes of C++ objects

The examples below use the following definitions.

```
class shape { … };

class line : public shape {
  member variables of class line…
}

line edge;
```

Displaying object attributes of C++ objects

To describe the attributes of the object edge, enter the following command.

```
DESCRIBE ATTRIBUTES edge;
```

The Log window displays the following output.

```
DESCRIBE ATTRIBUTES edge;
ATTRIBUTES for edge
  Its address is yyyyyyyy and its length is xx
  class line
     class shape
     member variables of class shape....
```

Note that the base class is shown as class shape _shape.

Displaying class attributes in C++

To display the attributes of class shape, enter the following command.

```
DESCRIBE ATTRIBUTES class shape;
```

The Log window displays the following output.

```
DESCRIBE ATTRIBUTES class shape ;
 ATTRIBUTES for class shape
   const class shape…
```

Displaying static data in C++

If a class contains static data, the static data will be shown as part of the class when displayed. For example:

```
class A {
  int x;
  static int y;
}

A obj;
```

You can also display the static member by referencing it as `A::y` since each object of class A has the same value.

Displaying global data in C++

To avoid ambiguity, variables declared at file scope can be referenced using the global scope operator ::. For example:

```
int x;
class A {
   int x;
:
   }
}
```

If you are within a member function of A and want to display the value of x at file scope, enter `LIST ::x`. If you do not use ::, entering `LIST x` will display the value of x for the current object (i.e., `this->x`).

Monitoring storage in C++

You might find it useful to monitor registers (general-purpose and floating-point) while stepping through your code and assembly listing by using the `LIST REGISTERS` command. The compiler listing displays the pseudo assembly code, including z/OS Debugger hooks. You can watch the hooks that you stop on and watch expected changes in register values step by step in accordance with the pseudo assembly instructions between the hooks. You can also modify the value of machine registers while stepping through your code.

You can list the contents of storage in various ways. Using the `LIST REGISTERS` command, you can receive a list of the contents of the General Purpose Registers or the floating-point registers.

You can also monitor the contents of storage by specifying a dump-format display of storage. To accomplish this, use the `LIST STORAGE` command. You can specify the address of the storage that you want to view, as well as the number of bytes.

Example: monitoring and modifying registers and storage in C

The examples below use the following C program to demonstrate how to monitor and modify registers and storage.

```
int dbl(int j)            /* line 1 */
{                         /* line 2 */
  return 2*j;             /* line 3 */
}                         /* line 4 */
int main(void)
{
  int i;
  i = 10;
  return dbl(i);
}
```

If you compile the program above using the compiler options TEST(ALL),LIST, then your pseudo assembly listing will be similar to the listing shown below.

```
*  int dbl(int j)
           ST    r1,152(,r13)
*  {
           EX    r0,HOOK..PGM-ENTRY
*    return 2*j;
           EX    r0,HOOK..STMT
           L     r15,152(,r13)
           L     r15,0(,r15)
           SLL   r15,1
           B     @5L2
           DC    A@5L2-ep)
           NOPR
 @5L1      DS    0D
*  }
 @5L2      DS    0D
           EX    r0,HOOK..PGM-EXIT
```

To display a continuously updated view of the registers in the Monitor window, enter the following command:

```
MONITOR LIST REGISTERS
```

After a few steps, z/OS Debugger halts on line 1 (the program entry hook, shown in the listing above). Another STEP takes you to line 3, and halts on the statement hook. The next STEP takes you to line 4, and halts on the program exit hook. As indicated by the pseudo assembly listing, only register 15 has changed during this STEP, and it contains the return value of the function. In the Monitor window, register 15 now has the value 0x00000014 (decimal 20), as expected.

You can change the value from 20 to 8 just before returning from dbl() by issuing the command:

```
%GPR15 = 8 ;
```

Chapter 34. Debugging an assembler program

To debug programs that have been assembled with debug information, you can use most of the z/OS Debugger commands. Any exceptions are noted in *IBM z/OS Debugger Reference and Messages*. Before debugging an assembler program, prepare your program as described in Chapter 7, "Preparing an assembler program," on page 71.

The SET ASSEMBLER and SET DISASSEMBLY commands

The `SET ASSEMBLER ON` and `SET DISASSEMBLY ON` commands enable some of the same functions. However, you must consider which type of CUs that you will be debugging (assembler, disassembly, or both) before deciding which command to use. The following guidelines can help you decide which command to use:

- If you are debugging assembler CUs but no disassembly CUs, you might want to use the `SET ASSEMBLER ON` command. If you need the following functions, use the `SET ASSEMBLER ON` command:
  - Use the `LIST`, `LIST NAMES CUS`, or `DESCRIBE CUS` commands to see the name of disassembly CUs.
  - Use `AT APPEARANCE` to stop z/OS Debugger when the disassembly CU is loaded.

  If you don't need any of these functions, you don't need to use either command.
- If you are debugging a disassembly CU, you must use the `SET DISASSEMBLY ON` command so that you can see the disassembly view of the disassembly CUs. The `SET DISASSEMBLY ON` command enables the functions enabled by `SET ASSEMBLER ON` and also enables the following functions that are not available through the `SET ASSEMBLER ON` command:
  - View the disassembled listing in the Source window.
  - Use the `STEP INTO` command to enter the disassembly CU.
  - Use the `AT ENTRY *` command to stop at the entry point of disassembly CUs.

If you are debugging an assembler CU and later decide you want to debug a disassembly CU, you can enter the `SET DISASSEMBLY ON` command after you enter the `SET ASSEMBLER ON` command.

Loading an assembler program's debug information

Use the `LOADDEBUGDATA` (or `LDD`) command to indicate to z/OS Debugger that a compile unit is an assembler compile unit and to load the debug information associated with that compile unit. The `LDD` command can be issued only for compile units which have no debug information and are, therefore, considered disassembly compile units. In the following example, `mypgm` is the compile unit (CSECT) name of an assembler program:

```
LDD mypgm
```

z/OS Debugger locates the debug information in a data set with the following name: *`yourid`*`.EQALANGX(mypgm)`. If z/OS Debugger finds this data set, you can begin to debug your assembler program. Otherwise, enter the `SET SOURCE` or `SET DEFAULT LISTINGS` command to indicate to z/OS Debugger where to find the debug information. In remote debug mode, the remote debugger prompts you for the data set information when the program is stepped into.

Normally, compile units without debug information are not listed when you enter the `DESCRIBE CUS` or `LIST NAMES CUS` commands. To include these compile units, enter the `SET ASSEMBLER ON` command. The next time you enter the `DESCRIBE CUS` or `LIST NAMES CUS` command, these compile units are listed.

© Copyright IBM Corp. 1992, 2023

z/OS Debugger session panel while debugging an assembler program

The z/OS Debugger session panel below shows the information displayed in the Source window while you debug an assembler program.

```
Assemble LOCATION: PUBS :> 34
Command ===>                                                                                                  Scroll ===> CSR
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---10----+--- LINE: 0 OF 0
******************************************************** TOP OF MONITOR ********************************************************
******************************************************** BOTTOM OF MONITOR *****************************************************

SOURCE: PUBS +----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---10-- LINE: 60 OF 513

   1 34 2      3                 *         7                                                                                  .
      34  00000078                OPENIT   EQU   *                                                                            .
      34  00000078               +         OPEN  ((2),INPUT)                                                                   .
      34           4             +         CNOP     0,4                       ALIGN LIST TO FULLWORD                           .
      34  00000078 4510 B080     +         BAL   1,*+8                        LOAD REG1 W/LIST ADDR. @L2A                      .
      35  0000007C               +         DC    A(0)                         OPT BYTE AND DCB ADDR.                           .
      36  00000080 5021 0000     +         ST    2,0(1,0)                     STORE INTO LIST        @L1C.                     .
      37  00000084 9280 1000     +         MVI   0(1),128                     MOVE IN OPTION BYTE                              .
      38  00000088 0A13          +         SVC   19                           ISSUE OPEN SVC                                   .
      39                      5  6         CALL  CEEMOUT,(STRING,DEST,0),VL   Omitted feedback code                            .
      39                         +         SYSSTATE TEST                                             @L3A                       .
      39                         +         CNOP  0,4                                                                            .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---10----+---1 LINE: 1 OF 9
******************************************************** TOP OF LOG ************************************************************
IBM z/OS Debugger 15.0.n
08/04/2020 03:55:40 AM
5724-T07: Copyright IBM Corp. 1992, 2020
0004 EQA1872E An error occurred while opening file: INSPPREF. The file may not exist, or is not accessible.
0005  Source or Listing data is not available, or the CU was not compiled with the correct compile options.
0006  LDD PUBS ;
0007  SET DEFAULT SCROLL CSR ;
0008  AT 34 ;
0009  GO ;
******************************************************** BOTTOM OF LOG *********************************************************
PF  1:?          2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP         8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

The information displayed in the Source window is similar to the listing generated by the assembler. The Source window displays the following information:

1 statement number
The statement number is a number assigned by the EQALANGX program. Use this column to set breakpoints and identify statements.

The same statement number can sometimes be assigned to more than one line. Comments, labels and macro invocations are assigned the same statement number as the machine instruction that follows these statements. All of these statements have the same offset within the CSECT, which allows you to put the cursor on any of these lines and press PF6 to set a breakpoint. When the statement is reached, the focus is set on the last line within the statement that contains either a macro invocation or a machine instruction.

2
An asterisk in the column preceding the offset indicates that the line is contained in a compile unit to which you are not currently qualified. Before you attempt to set a line or statement breakpoint on that a line, you must enter the `SET QUALIFY CU` *`compile_unit`* and specify the name of the containing compile unit for the *compile_unit* parameter.

3 offset
The offset from the start of the CSECT. This column matches the left-most column in the assembler listing.

4 object
The object code for instructions. This column matches the "Object Code" column in the assembler listing. Object code for data fields is not displayed.

5 modified instruction
An "X" in this column indicates an executable instruction that is modified by the program at some point. You cannot set a breakpoint on such an instruction nor can you STEP into such an instruction.

6 macro generated
A "+" in this column indicates that the line is generated by macro expansion. Lines generated by macro expansion appear only in the standard view. These lines are suppressed when the NOMACGEN view is in effect.

7 source statement
The original source statement. This column corresponds to the "Source Statement" column in the assembler listing.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

%PATHCODE values for assembler programs

This table shows the possible values for the z/OS Debugger %PATHCODE variable when the current programming language is Assembler:

%PATHCODE	Entry type	Instruction	Additional requirements or comments
1	A block has been entered.	Any	External symbol whose offset corresponds to an instruction
2	A block is about to be exited.	`BR R14` `(07FE)`	
		`BALR R14,R15` `(05EF)`	These instructions are considered an Exit only if this instruction is not followed by an valid instruction.
		`BASR R14,R15` `(0DEF)`	
		`BASSM R14,R15` `(0CEF)`	
		`BCR 15,x` `(07Fx)`	
3	Control has reached a label coded in the program.	Any	Label whose offset corresponds to an instruction.

%PATHCODE	Entry type	Instruction	Additional requirements or comments
4	Control is being transferred as a result of a CALL.	`BALR R14,R15 (05EF)`	
		`BASR R14,R15 (0DEF)`	
		`BASSM R14,R15 (0CEF)`	
		SVC (0A)	
		PC (B218)	
		BAL (45)	Except BAL 1,xxx is not considered a CALL
		BAS (4D)	
		`BALR x,y (05)`	
		`BASR x,y (0D)`	
		`BASSM x,y (0C)`	
		BRAS (A7x5)	
		BRASL (C0x5)	
5	Control is returning from a CALL.	Statement after CALL	If the statement after a CALL is an instruction, it gets an entry here.

%PATHCODE	Entry type	Instruction	Additional requirements or comments
6	A conditional branch is about to be executed.	BC x (47x)	x^=15 & X^=0
		BCR x (07x)	x^=15 & X^=0
		BCT (46)	
		BCTR (06)	
		BCTGR (B946)	
		BXH (86)	
		BXHG (EB44)	
		BXLE (87)	
		BXLEG (EB45)	
		BRC x (A7x4)	x^=15 & X^=0
		BRCL (C0x4)	
		BRCT (A7x6)	
		BRCTG (A7x7)	
		BRXH (84)	
		BRXHG (EC44)	
		BRXLE (85)	
		BRXLG (EC45)	
7	A conditional branch was not executed and control has "fallen-through" to the next instruction.	Statement after Conditional Branch	
8	An unconditional branch is about to be executed.	BC 15,x (47Fx)	
		BRC 15,x (A7F4)	
		BRCL 15,x (C0F4)	
		BSM (0B)	

Using the STANDARD and NOMACGEN view

The information displayed in the Source window for an assembler program can be viewed in either of two views. The STANDARD view shows all lines in the assembler listing including lines generated through macro expansion. The NOMACGEN view omits lines generated by macro expansion and, therefore, is similar to the assembler listing generated when PRINT NOGEN is in effect.

You can use the following commands to control the view that you see in the Source window for an assembler program:

- SET DEFAULT VIEW is used to indicate the initial view that you see. The setting that is in effect for SET DEFAULT VIEW when you enter the LOADDEBUGDATA (LDD) command for an assembler program determines the initial view for that program.
- QUERY DEFAULT VIEW can be used to see the current setting of SET DEFAULT VIEW.
- QUERY CURRENT VIEW can be used to determine the view in effect for the currently qualified CU.

Debugging non-reentrant assembler

When a load module is marked as non-reentrant and loaded multiple times without a corresponding delete, multiple copies of the load module exist in memory at the same time. Because high level language programs are typically marked as reentrant by default, debugging non-reentrant programs primarily applies to the debugging of assembler programs. The following situations have the special considerations described in the following sections when debugging non-reentrant assembler programs:

- Manipulating breakpoints
- Manipulating local variables

The following descriptions apply only to full screen mode and line mode debugging. There are no corresponding features for supporting debugging of non-reentrant assembler when using the remote debugger.

Manipulating breakpoints in non-reentrant assembler load modules

When you manipulate breakpoints in a compile unit in a non-reentrant load module by using one of the following commands, the command applies to all copies of the compile unit in load modules with the same name:

- `AT`
- `DISABLE AT`
- `ENABLE AT`
- `LIST AT`
- `CLEAR AT`
- `SET SAVE BPS`
- `SET RESTORE BPS`

Manipulating local variables in non-reentrant assembler load modules

If you want refer to a local variable that is in a compile unit in a non-reentrant load module and multiple copies of that load module exist in memory, you must identify the copy of the compile unit to which you want the command to apply. To identify the copy of the compile unit, you must first obtain an address in the specific compile unit. The following list describes some ways you can obtain an address in a specific compile unit:

- Inspect a variable or register in the calling program for the address of the specific compile unit.
- Enter the `QUERY LOCATION` command to obtain the address of the specific compile unit.
- Enter the `DESCRIBE CU` command to see a list of addresses for each compile unit. Then, enter the `QUALIFY` command with each address until you find the specific compile unit.

After you obtain the address, enter the `SET QUALIFY` *`address`*`;` command, where *address* is an address in the specific compile unit you identified.

Restrictions for debugging an assembler program

When you debug assembler programs the following general restrictions apply:

- Only application programs are supported. No support is provided for debugging system routines, authorized programs, CICS exits, and so on.

- Debugging of Private Code (also known as an unnamed CSECT or blank CSECT) is not supported.
- To debug subtasks that are started by the ATTACH macro, delay debug mode must be in effect. Subtasks that are started by the ATTACH macro can be debugged in one of the following circumstances:
  - If the main task starts in a non-Language Environment program, the task must be started by calling EQANMDBG and supplying the TEST option. For more information, see "Starting z/OS Debugger for programs that start outside of Language Environment" on page 134.
  - If the main task starts in a Language Environment program, or if a Language Environment program is the first program to be debugged, you must specify the TEST run time option (for example, via a CEEOPTS DD statement).

  For more information, see "Debugging subtasks created by the ATTACH assembler macro" on page 398.
- You cannot debug programs that do not use standard linkage conventions for registers 13, 14, and 15 or that use the Linkage Stack. Not using standard linkage conventions or the Linkage Stack can cause the following commands to function incorrectly:
  - `LIST CALLS`
  - `STEP RETURN`
  - STEP (when stopped at a return instruction)
  - `%EPA`
- Debugging of programs that use the MVS XCTL SVC is not supported.
- Debugging of the 64-bit Language Environment-enabled and Language Environment XPLINK programs is not supported.
- CICS does not support 64-bit programs interfacing to CICS services; therefore, z/OS Debugger does not support debugging of 64-bit programs under CICS.
- Support for binary and decimal floating-point items requires 64-bit hardware and Decimal Floating Point hardware (for decimal floating point support).
- If your current hardware does not support 64-bit instructions or your program is suspended at a point where the 64-bit General Purpose Registers are not available, the 64-bit General Purpose Registers are not available and any reference to symbols for the 64-bit General Purpose Registers are treated as undefined.
- The 64-bit General Purpose Registers are available only in the compile unit in which z/OS Debugger is stopped at a breakpoint. If you use the `QUALIFY` command to qualify to a compile unit higher in the calling sequence, the 64-bit General Purpose Registers are not accessible.
- When your program is suspended in a compile unit, that compile unit is the only one from which you can access the 64-bit General Purpose Registers. If you use the `QUALIFY` command to qualify to a different compile unit, you can no longer access the 64-bit General Purpose Registers.
- Debugging of programs that use Access Register mode is not supported.
- Debugging of programs that use the IDENTIFY macro or service is not supported.
- You cannot debug programs that were assembled with features that depend on the GOFF option, for example, CSECT names longer than eight characters. If the program can assemble correctly without the GOFF option, then you can debug programs that are assembled with the GOFF option.
- If you are debugging a program that uses ESTAE or ESTAEX, the program behaves as if `TRAP(OFF)` were specified for all Abends while the ESTAE or ESTAEX is active, except program checks. In other words, no condition is seen by z/OS Debugger. Any Abends except program checks are handled by the ESTAE(X) exit in your program.
- If you are debugging a program that uses SPIE or ESPIE, the program behaves as if `TRAP(OFF)` were specified for all program checks while the SPIE or ESPIE is active, except a program check that might arise from the use of the `CALL` z/OS Debugger command.
- The debugging of TSO Command Processors is not supported.

- If you start debugging in a non-CICS load module that is not the "top" load module, you cannot continue debugging after that load module returns to its caller. In order to do this, you must invoke z/OS Debugger using CEEUOPT or some other internal method. You cannot do this by using JCL alone.
- Debugging of assembler or disassembly code requires the use of the Dynamic Debug Facility. z/OS Debugger does not support the use of the Dynamic Debug Facility to debug code that is not known to the z/OS Contents Supervisor. This can occur in situations similar to the following situations:
  - Debugging load modules loaded by a directed LOAD.
  - Debugging segments of code which have been relocated. For example, a GETMAIN is used to obtain a new piece of storage. Then a section of code is moved into this new piece of storage and control is passed to it for execution.

Restrictions for debugging a Language Environment assembler MAIN program

When you debug a Language Environment-enabled assembler main program, the following restrictions apply:

- If z/OS Debugger is positioned at the entry point to the assembler main program and you enter a STEP command, the STEP command stops at the instruction that is after the prologue BALR instruction that initializes Language Environment. You cannot step through the portion of the prologue that is before the completion of Language Environment initialization.
- If you set a breakpoint in the prologue before the completion of Language Environment initialization, the breakpoint is accepted. However, z/OS Debugger does not stop or gain control at this breakpoint.

To debug a Language Environment-conforming assembler MAIN program running under CICS, you must run with CICS Transaction Server, Version 3.1 or later.

Restrictions on setting breakpoints in the prologue of Language Environment assembler programs

The following restrictions apply when you attempt to set explicit or implicit breakpoints in the prologue of a Language Environment assembler program:

- If you try to step across the portion of the prologue code that is between the point where the stack extend routine is called and the `LR 13,x` instruction that loads the address of the new DSA into register 13, the STEP command stops at the instruction immediately following the `LR 13,x` instruction.
- If you try to set a breakpoint in the portion of the prologue code between the point where the stack extend routine is called and the `LR 13,x` instruction that loads the address of the new DSA into register 13, z/OS Debugger will not set the breakpoint.

Restrictions for debugging non-Language Environment programs

If you specify the TEST runtime option with the `NOPROMPT` suboption when you start your program and z/OS Debugger is subsequently started by `CALL CEETEST` or the raising of a Language Environment condition, you can debug both Language Environment and non-Language Environment programs and detect both Language Environment and non-Language Environment events in the enclave that started z/OS Debugger and in subsequent enclaves. You cannot debug non-Language Environment programs or detect non-Language Environment events in higher-level enclaves. After control has returned from the enclave in which z/OS Debugger was started, you can no longer debug non-Language Environment programs or detect non-Language Environment events.

Restrictions for debugging assembler code that uses instructions as data

z/OS Debugger cannot debug code that uses instructions as data. If your program references one or more instructions as data, the result can be unpredictable, including an abnormal termination (ABEND) of z/OS Debugger. This is because z/OS Debugger sometimes replaces instructions with SVCs in order to create breakpoints.

For example, z/OS Debugger cannot process the following code correctly:

```
Entry1   BRAS  15,0
         NOPR  0
         B     Common
Entry2   BRAS  15,0
         NOPR  4
Common   DS    0H
         IC    15,1(15)
```

In this code, the IC is used to examine the second byte of the NOPR instructions. However, if the NOPR instructions are replaced by an SVC to create a breakpoint, a value that is neither 0 nor 4 might be obtained, which causes unexpected results in the user program.

You can use the following coding techniques can be used to eliminate this problem:

- Method 1: Change the code to reference constants instead of instructions.
- Method 2: Define the referenced instructions by using DC instructions instead of executable instructions.

Using Method 1, you can change the above example to the following code:

```
Entry1   BAL   15,*+L'*+2
         DC    H'0'
         B     Common
Entry2   BAL   15,*+L'*+2
         DC    H'4'
Common   DS    0H
         IC    15,1(15)
```

Using Method 2, you can change the above example to the following code:

```
Entry1   BRAS  15,0
         DC    X'0700'
         B     Common
Entry2   BRAS  15,0
         DC    X'0704'
Common   DS    0H
         IC    15,1(15)
```

Restrictions for debugging self-modifying assembler code

z/OS Debugger defines two types of self-modifying code: detectable and non-detectable. Detectable self-modifying code is code that either:

- Modifies an instruction via a direct reference to a label on the instruction or on an EQU * or DS 0H immediately preceding the instruction. For example:

  ```
  Inst1    NOP   Label1
           MVI   Inst1+1,X'F0'
  ```

- Uses the EQAMODIN macro instruction to identify the instruction being modified. For example:

  ```
           EQAModIn  Inst1
  Inst1    NOP       Label1
           LA        R3,Inst1
           MVI       0(R3),X'F0'
  ```

Any self-modifying code that does not meet one of these criteria is classified as non-detectable.

Handling of detectable self-modifying assembler code

When z/OS Debugger identifies detectable, self-modifying code, it indicates the situation in the Source window by putting an "X" in the column immediately before the column indicating a macro-generated instruction. A breakpoint cannot be set on such an instruction nor will STEP stop on such an instruction.

The EQAMODIN macro in shipped in the z/OS Debugger sample library (*hlq*.SEQASAMP). This macro can be used to make non-detectable, self-modifying code detectable. It generates no executable code.

Instead it simply adds information to the SYSADATA file to identify the specified operand as modified. The operand can be specified either as a label name or as "*" to indicate that the immediately following instruction is modified.

Non-detectable self-modifying assembler code

If your program contains non-detectable, self-modifying code that modifies an instruction while the containing compilation unit is being debugged, the result can be unpredictable, including an abnormal termination (ABEND) of z/OS Debugger. If your program contains self-modifying code that completely replaces an instruction while the containing compilation unit is being debugged and you do not step through the code that modifies the instruction, the result might not be an ABEND. However, z/OS Debugger might miss a breakpoint on that instruction or display a message indicating an invalid hook address at delete. If you *do* step through the code that modifies the instruction, the instruction that is moved may contain a breakpoint causing a z/OS Debugger failure when the modified instruction is executed.

The following coding techniques can be used to minimize problems debugging non-detectable, self-modifying code:

- Define instructions to be modified by using DC instructions instead of executable instructions. For example, use the instruction `ModInst DC X'4700',S(Target)` instead of the instruction `BC 0,Target`.

Code that modifies an instruction defined by an instruction op-code	**Code that modifies an instruction defined by a DC**

`ModInst BC  0,Target`
`...`
`MVI ModInst+1,X'F0'`
	`ModInst DC  X'4700',S(Target)`
	`...`
	`MVI ModInst+1,X'F0'`

- Do not modify part of an instruction. Instead, replace an instruction with one that is generated with a DC or marked as modified by use of the EQAMODIN macro. The following table compares coding techniques:

Code that modifies an instruction	**Corresponding code that replaces an instruction with one defined by a DC**

`ModInst  BC 0,Target`
`...`
`MVI ModInst+1,X'F0'`
	`ModInst  BC 0,Target`
	`...`
	`MVC ModInst(4),NewInst`
	`...`
	`NewInst DC  X'47F0',S(Target)`

Code that modifies an instruction	**Corresponding code that replaces an instruction marked by EQAMODIN**

`ModInst BC  0,Target`
`...`
`MVI ModInst+1,X'F0'`
	`ModInst BC  0,Target`
	`...`
	`MVC ModInst(4),NewInst`
	`...`
	`EQAMODIN NewInst`
	`NewInst BC  15,Target`

Restrictions for debugging assembler programs that consist of multiple sections

When your assembler program consists of multiple sections, indicate that debug information should be loaded for all sections by using command **SET LDD ALL** before using the **LDD** command.

For more information, see “Multiple compilation units in a single assembly” on page 246.

If the debug information is not loaded for all sections in the compile unit (CU), the result can be unpredictable, and an abnormal termination (ABEND) might occur.

In the following example, the EX instruction and its target are coded in separate CSECTs of the program:

```
A     CSECT
            L     15,B$BASE
            BASR  14,15
B           CSECT
            LA    5,8
            EX    5,MVC
            BR    14
A           CSECT
B$BASE      DC    A(B)
MVC         MVC   TO(0),FROM
TO          DC    CL9'123456789'
FROM        DC    CL9'987654321'
```

If the debug information is loaded for `CSECT A` that contains the target of the EX instruction, but not loaded for `CSECT B` that contains the EX instruction, the EX instruction will abend immediately after the **STEP** command is performed on the `BASR 14,15` instruction.

Restrictions for debugging assembler programs when SET DEFAULT VIEW NOMACGEN is in use

The result can be unpredictable, and an abnormal termination (ABEND) might occur, if you debug an assembler program using `SET DEFAULT VIEW NOMACGEN` and both of the following conditions are met:

- The program uses a macro that contains an EX instruction, and the program logic goes through this instruction.
- You use the STEP command to go through the invocation of the macro with the EX instruction.

In the following example, `ABEND0C1` occurs if you use the STEP command to go through the invocation of macro EXMAC at location `A ::> A :> 6`.

Note: In this example, a target of the EX instruction is coded among other instructions and branched around, and not in the constants' part of the program.

```
              MACRO
              EXMAC
              XR     14,14
              EX     14,EXCLC
              MEND
A             CSECT
              STM    14,12,12(13)
              LR     10,15
              USING  A,10
              ST     13,SAVEAREA+4
              LA     11,SAVEAREA
              LR     13,11
              EXMAC
              B      RETURN
EXCLC         CLC    TARGET(0),SOURCE
RETURN        DS     0H
              L      13,SAVEAREA+4
              LM     14,12,12(13)
              XR     15,15
              BR     14
SAVEAREA      DC     18F'0'
SOURCE        DS     C
TARGET        DS     C
              END    A
```

To avoid this kind of failure, set a breakpoint on the following statement and use the `GO` command instead of the STEP command to go through the invocation of the macro with the EX instruction.

Chapter 35. Debugging a disassembled program

To debug programs that have been compiled or assembled without debug information, you can use the disassembly view. When you use the disassembly view, symbolic information from the original source program (program variables, labels, and other symbolic references to a section of memory) is not available. The DYNDEBUG switch must be ON before you use the disassembly view.

If you are not familiar with the program that you are debugging, we recommend that you have a copy of the listing that was created by the compiler or High Level Assembler (HLASM) available while you debug the program. There are no special assembly or compile requirements that the program must comply with to use the disassembly view.

The SET ASSEMBLER and SET DISASSEMBLY commands

The SET ASSEMBLER ON and SET DISASSEMBLY ON commands enable some of the same functions. However, you must consider which type of CUs that you will be debugging (assembler, disassembly, or both) before deciding which command to use. The following guidelines can help you decide which command to use:

- If you are debugging assembler CUs but no disassembly CUs, you might want to use the SET ASSEMBLER ON command. If you need the following functions, use the SET ASSEMBLER ON command:
  - Use the LIST, LIST NAMES CUS, or DESCRIBE CUS commands to see the name of disassembly CUs.
  - Use AT APPEARANCE to stop z/OS Debugger when the disassembly CU is loaded.

  If you don't need any of these functions, you don't need to use either command.
- If you are debugging a disassembly CU, you must use the SET DISASSEMBLY ON command so that you can see the disassembly view of the disassembly CUs. The SET DISASSEMBLY ON command enables the functions enabled by SET ASSEMBLER ON and also enables the following functions that are not available through the SET ASSEMBLER ON command:
  - View the disassembled listing in the Source window.
  - Use the STEP INTO command to enter the disassembly CU.
  - Use the AT ENTRY * command to stop at the entry point of disassembly CUs.

If you are debugging an assembler CU and later decide you want to debug a disassembly CU, you can enter the SET DISASSEMBLY ON command after you enter the SET ASSEMBLER ON command.

Capabilities of the disassembly view

When you use the disassembly view, you can do the following tasks:

- Set breakpoints at the start of any assembler instruction.
- Step through the disassembly instructions of your program.
- Display and modify registers.
- Display and modify storage.
- Monitor General Purpose Registers or areas of main storage.
- Switch the debug view.
- Use most z/OS Debugger commands.

© Copyright IBM Corp. 1992, 2023

Starting the disassembly view

To start the disassembly view:

1. Enter the `SET DISASSEMBLY ON` command
2. Open the program that does not contain debug data. z/OS Debugger then changes the language setting to **Disassem** and the Source window displays the assembler code.

If you enter a program that does contain debug data, the language setting does not change and the Source window does not display disassembly code.

The disassembly view

When you debug a program through the disassembly view, the Source window displays the disassembly instructions. The language area of the z/OS Debugger screen (upper left corner) displays the word **Disassem**. The z/OS Debugger screen appears as follows:

```
Disassem LOCATION: MAIN initialization
Command ===>                                                  Scroll ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 0 OF 0
****************************** TOP OF MONITOR ******************************
****************************** BOTTOM OF MONITOR ***************************



SOURCE: MAIN +----1----+----2----+----3----+----4----+----5----+ LINE: 1 OF 160
      0 1950C770    47F0 F014       BC    15,20(,R15)                        .
   A  4 1950C774    00C3            ????                                     .
      6 1950C776 B  C5C5            ????                                     .
      8 1950C778    0000            ????                                     .
      A 1950C77A    0080 C          ????                                     .
      C 1950C77C    0000            ????                                     .
      E 1950C77E    00C4            ???? D                                   .
     10 1950C780    47F0 F001       BC    15,1(,R15)                         .
     14 1950C784    90EC D00C       STM   R14,R12,12(R13)                    .
     18 1950C788    18BF            LR    R11,R15  E                         .
     1A 1950C78A    5820 B130       L     R2,304(,R11)                       .
     1E 1950C78E    58F0 B134       L     R15,308(,R11)                      .
     22 1950C792    05EF            BALR  R14,R15                            .
     24 1950C794    1821            LR    R2,R1                              .
     26 1950C796    58E0 C2F0       L     R14,752(,R12)                      .
     2A 1950C79A    9680 E008       OI    8(R14),128                         .
     2E 1950C79E    05B0            BALR  R11,0                              .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 5
********************************* TOP OF LOG *********************************
IBM z/OS Debugger 15.0.n
08/04/2020 03:55:40 AM
5724-T07: Copyright IBM Corp. 1992, 2020
0004 EQA1872E An error occurred while opening file: INSPPREF. The file may not
0005          exist, or is not accessible.
0006 SET DISASSEMBLY ON ;
PF  1:?           2:STEP       3:QUIT       4:LIST       5:FIND      6:AT/CLEAR
PF  7:UP          8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG 12:RETRIEVE
```

A Prefix Area
Displays the offset from the start of the CU or CSECT.

B Columns 1-8
Displays the address of the machine instruction in memory.

C Columns 13-26
Displays the machine instruction in memory.

D Columns 29-32
Displays the op-code mnemonic or ???? if the op-code is not valid.

E Columns 35-70
Displays the disassembled machine instruction.

When you use the disassembly view, the disassembly instructions displayed in the source area are not guaranteed to be accurate because it is not always possible to distinguish data from instructions. Because

of the possible inaccuracies, we recommend that you have a copy of the listing that was created by the compiler or by HLASM. z/OS Debugger keeps the disassembly view as accurate as possible by refreshing the Source window whenever it processes the machine code, for example, after a STEP command.

Performing single-step operations in the disassembly view

Use the STEP command to single-step through your program. In the disassembly view, you step from one disassembly instruction to the next. z/OS Debugger highlights the instruction that it runs next.

If you try to step back into the program that called your program, set a breakpoint at the instruction to which you return in the calling program. If you try to step over another program, set a breakpoint immediately after the instruction that calls another program. When you try to step out of your program, z/OS Debugger displays a warning message and lets you set the appropriate breakpoints. Then you can do the step.

z/OS Debugger refreshes the disassembly view whenever it determines that the disassembly instructions that are displayed are no longer correct. This refresh can happen while you are stepping through your program.

Setting breakpoints in the disassembly view

You can use a special breakpoint when you debug your program through the disassembly view. AT OFFSET sets a breakpoint at the point that is calculated from the start of the entry point address of the CSECT. You can set a breakpoint by entering the AT OFFSET command on the command line or by placing the cursor in the prefix area of the line where you want to set a breakpoint and press the AT function key or type AT in the prefix area.

z/OS Debugger lets you set breakpoints anywhere within the starting and ending address range of the CU or CSECT provided that the address appears to be a valid op-code and is an even number offset. To avoid setting breakpoints at the wrong offset, we recommend that you verify the offset by referring to a copy of the listing that was created by the compiler or by HLASM.

Restrictions for debugging self-modifying code

z/OS Debugger cannot debug self-modifying code. If your program contains self-modifying code that modifies an instruction while the containing compilation unit is being debugged, the result can be unpredictable, including an abnormal termination (ABEND) of z/OS Debugger. If your program contains self-modifying code that completely replaces an instruction while the containing compilation unit is being debugged, the result might not be an ABEND. However, z/OS Debugger might miss a breakpoint on that instruction or display a message indicating an invalid hook address at delete.

The following coding techniques can be used to minimize problems debugging self-modifying code:

1. Do not modify part of an instruction. Instead, replace an instruction. The following table compares coding techniques:

Coding that modifies an instructions	Coding that replaces an instruction

`ModInst  BC 0,Target`
`   ...`
`   MVI ModInst+1,X'F0'`
	`ModInst  BC 0,Target`
	`   ...`
	`   MVC ModInst(4),NewInst`
	`   ...`
	`NewInst  BC 15,Target`

2. Define instructions to be modified by using DC instructions instead of executable instructions. For example, use the instruction `ModInst DC X'4700',S(Target)` instead of the instruction `MVC ModInst(4),NewInst`.

Displaying and modifying registers in the disassembly view

You can display the contents of all the registers by using the LIST REGISTERS command. To display the contents of an individual register, use the LIST Rx command, where x is the individual register number. You can also display the contents of an individual register by placing the cursor on the register and pressing the LIST function key. The default LIST function key is PF4. You can modify the contents of a register by using the assembler assignment statement.

Displaying and modifying storage in the disassembly view

You can display the contents of storage by using the LIST STORAGE command. You can modify the contents of storage by using the STORAGE command.

You can also use assembler statements to display and modify storage. For example, to set the four bytes located by the address in register 2 to zero, enter the following command:

```
R2-> <4>=0
```

To verify that the four bytes are set to zero, enter the following command:

```
LIST R2->
```

Changing the program displayed in the disassembly view

You can use the SET QUALIFY command to change the program that is displayed in the disassembly view. Suppose you are debugging program ABC and you need to set a breakpoint in program BCD.

1. Enter the command SET QUALIFY CU BCD on the command line. z/OS Debugger changes the Source window to display the disassembly instructions for program BCD.
2. Scroll through the Source window until you find the instruction where want to set a breakpoint.
3. To return to program ABC, at the point where the next instruction is to run, issue the SET QUALIFY RESET command.

Restrictions for the disassembly view

When you debug a disassembled program, the following restrictions apply:

- Applications that use the Language Environment XPLINK linking convention are not supported.
- The Dynamic Debug facility must be activated before you start debugging through the disassembly view.
- Debugging of assembler or disassembly code requires the use of the Dynamic Debug Facility. z/OS Debugger does not support the use of the Dynamic Debug Facility to debug code that is not known to the z/OS Contents Supervisor. This can occur in situations similar to the following situations:
  - Debugging load modules loaded by a directed LOAD.
  - Debugging segments of code which have been relocated. For example, a GETMAIN is used to obtain a new piece of storage. Then a section of code is moved into this new piece of storage and control is passed to it for execution.

When you debug a program through the disassembly view, z/OS Debugger cannot stop the application in any of the following situations:

- The program does not comply with the first three restrictions that are listed above.
- Between the following instructions:
  - After the LE stack extend has been called in the prologue code, and
  - Before R13 has been set with a savearea or DSA address and the backward pointer has been properly set.

The application runs until z/OS Debugger encounters a valid save area backchain.

Part 6. Debugging in different environments

© Copyright IBM Corp. 1992, 2023

Chapter 36. Debugging Db2 programs

While you debug a program containing SQL statements, remember the following behaviors:

- The SQL preprocessor replaces all the SQL statements in the program with host language code. The modified source output from the preprocessor contains the original SQL statements in comment form. For this reason, the source or listing view displayed during a debugging session can look very different from the original source.
- The host language code inserted by the SQL preprocessor starts the SQL access module for your program. You can halt program execution at each call to a SQL module and immediately following each call to a SQL module, but the called modules cannot be debugged.
- A host language SQL coprocessor performs Db2 precompiler functions at compile time and replaces the SQL statements in the program with host language code. However, the generated host language code is not displayed during a debug session; the original source code is displayed.

The topics below describe the steps you need to follow to use z/OS Debugger to debug your Db2 programs.

- Chapter 8, “Preparing a Db2 program,” on page 75
- “Processing SQL statements” on page 75
- “Linking Db2 programs for debugging” on page 76
- “Binding Db2 programs for debugging” on page 77
- “Debugging Db2 programs in batch mode” on page 333
- “Debugging Db2 programs in full-screen mode” on page 333

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 8, “Preparing a Db2 program,” on page 75
DB2 UDB for z/OS Application Programming and SQL Guide

Debugging Db2 programs in batch mode

In order to debug your program with z/OS Debugger while in batch mode, follow these steps:

1. Make sure the z/OS Debugger modules are available, either by STEPLIB or through the LINKLIB.
2. Provide all the data set definitions in the form of DD statements (example: Log, Preference, list, and so on).
3. Specify your debug commands in the command input file.
4. Run your program through the TSO batch facility.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 8, “Preparing a Db2 program,” on page 75

Debugging Db2 programs in full-screen mode

In full-screen mode, you can decide at debug time what debugging commands you want issued during the test.

Using z/OS Debugger Setup Utility (DTSU)

The z/OS Debugger Setup Utility is available through IBM z/OS Debugger Utilities.

© Copyright IBM Corp. 1992, 2023

1. Start DTSU by using the TSO command or the ISPF panel option, if available. Contact your system administrator to determine if the ISPF panel option is available.
2. Create a setup file. Remember to select the **Initialize New setup file for Db2** field.
3. Enter appropriate information for all the fields. Remember to enter the proper commands in the **DSN command options** and the **RUN command options** fields.
4. Enter the RUN command to run the Db2 program.

Using TSO commands

1. Ensure that either you or your system programmer has allocated all the required data sets through a CLIST or REXX EXEC.
2. Issue the DSN command to start Db2.
3. Issue the RUN subcommand to execute your program. You can specify the TEST runtime option as a parameter on the RUN subcommand. The following example starts a COBOL program:

```
RUN PROG(progname) PLAN(planname) LIB('user.library')
    PARMS('/TEST(,*,:,*)')
```

   The following example starts a non-Language Environment COBOL program:

```
RUN PROG(EQANMDBG) PLAN(planname) LIB('user.library')
    PARMS('progname,/TEST(,*,:,*)')
```

Using TSO/Call Access Facility (CAF)

1. Link-edit the CAF language interface module DSNALI with your program.
2. Ensure that the data sets required by z/OS Debugger and your program have been allocated through a CLIST or REXX procedure.
3. Enter the TSO CALL command CALL '*user.library*(name of your program)', to start your program. Include the TEST run-time option as a parameter in this command.

In full-screen mode using a dedicated terminal without Terminal Interface Manager

1. Specify the MFI%*LU_name* parameter as part of the TEST runtime option.
2. Follow the other requirements for debugging Db2 programs either under TSO or in batch mode.

In full-screen mode using the Terminal Interface Manager

1. Specify the VTAM%*userid* parameter as part of the TEST runtime option.
2. Follow the other requirements for debugging Db2 programs either under TSO or in batch mode.

After your program has been initiated, debug your program by issuing the required z/OS Debugger commands.

Note: If your source does not come up in z/OS Debugger when you launch it, check that the listing or source file name corresponds to the MVS library name, and that you have at least read access to that MVS library.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 8, “Preparing a Db2 program,” on page 75
“Starting z/OS Debugger for programs that start outside of Language Environment” on page 134

Related references

DB2 UDB for z/OS Administration Guide

Chapter 37. Debugging Db2 stored procedures

A Db2 stored procedure is a compiled high-level language (HLL) program that can run SQL statements. z/OS Debugger can debug any stored procedure written in assembler (if the program type is MAIN), C, C++, COBOL, and PL/I in any of the following debugging modes:

- remote debug mode
- full-screen mode using the Terminal Interface Manager
- batch mode

Before you begin, verify that you have completed all the tasks described in Chapter 9, "Preparing a Db2 stored procedures program," on page 79. The program resides in an address space that is separate from the calling program. The stored procedure can be called by another application or a tool such as the IBM Db2 Development Center.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 9, "Preparing a Db2 stored procedures program," on page 79
"Resolving some common problems while debugging Db2 stored procedures" on page 335

Related references

Db2 Application Programming and SQL Guide

Resolving some common problems while debugging Db2 stored procedures

This topic describes the messages you might receive and resolution to the problem described by those messages. This topic covers common problems.

Table 25. Common problems while debugging stored procedures and resolutions to those problems

Error code	Error message	Resolution
`SQLCODE = 471, SQLERRMC = 00E79001`	Stored procedure was stopped.	Start the stored procedure using Db2 Start Procedure command.
`SQLCODE = 471, SQLERRMC = 00E79002`	Stored procedure could not be started because of a scheduling problem.	Try using the Db2 Start Procedure command. If this does not work, contact the Db2 Administrator to raise the dispatching priority of the procedure.
`SQLCODE = 471, SQLERRMC = 00E7900C`	WLM application environment name is not defined or available.	Activate the WLM address space using the MVS `WLM VARY` command, for example: `WLM VARY APPLENV=applenv,RESUME` where *applenv* is the name of the WLM address space.
`SQLCODE = 444, SQLERRMC (none)`	Program not found.	Verify that the `LOADLIB` is in the `STEPLIB` for the WLM or Db2 address space JCL and has the appropriate RACF Read authorization for other applications to access it.

© Copyright IBM Corp. 1992, 2023

Table 25. Common problems while debugging stored procedures and resolutions to those problems (continued)

Error code	Error message	Resolution
`SQLCODE = 430, SQLERRMC (none)`	Abnormal termination in stored procedure	This can occur for many reasons. If the stored procedure abends without calling z/OS Debugger, analyze the Procedure for any logic errors. If the Procedure runs successfully without z/OS Debugger, there may a problem with how the stored procedure was compiled and linked. Be sure that the Procedure data set has the proper RACF authorizations. There may be a problem with the address space. Verify that the WLM or Db2 Address Space is correct. If there are any modifications, be sure the region is recycled.

Chapter 38. Debugging IMS programs

This topic describes the tasks involved in debugging IMS programs.

Using IMS Transaction Isolation to create a private message-processing region and select transactions to debug

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

z/OS Debugger's IMS Transaction Isolation facility allows you to debug IMS message processing programs (MPPs) in an environment that is isolated from other users of the same programs.

Using the IMS Transaction Isolation facility, you can do the following tasks:

1. Display a list of transactions available for a given IMS subsystem.
2. From that list of transactions, register to debug a specific transaction in a private message region that is created for your use.
3. For transactions you are registered to debug, specify other pattern-matching information, such as the content of messages that are sent to the transaction. This allows you to trap the transaction under specific conditions.
4. Start a private message-processing region based on the execution environment of a selected transaction. The private message-processing region is configured to use delay debug mode, and is hardcoded to read delay debug preferences from your delay debug profile data set.
5. Customize your private message region by supplying personal libraries for the STEPLIB concatenation.

To use the IMS Transaction Isolation facility, do the following tasks:

1. Start IBM z/OS Debugger Utilities. For more information, see “Starting IBM z/OS Debugger Utilities” on page 416.
2. In the **IBM z/OS Debugger Utilities** panel (EQA@PRIM), type 4 in the Option line and press Enter.
3. In the **Manage IMS Message Processing Programs** panel (EQAPRIS), type 5 in the Option line and press Enter.
4. The **IMS Transaction Isolation Facility** panel (EQAPMPSL) is displayed. The following screen highlights the fields in the panel.

```
--------------- IMS Transaction Isolation Facility ---------- Row 1 to 7 of 201
 Command ===>                                    Scroll ===> PAGE

IMS system . . . . . IMS1  1

 2  _ Manage additional libraries and delay debug options.
       Your region: @USRT001 Class 021 Stopped
       Delay debug data set: 'USRT001.DLAYDBG.EQAUOPTS'

  Filters:  3
   / Display full transaction list.
   _ Display only transactions you are registered to debug.
   _ Filter by name ==> dtmq___
__ 4 __50 Maximum number of transactions (0 - no limit)

(E) Edit  (S) Start Region  (P) Stop Region  (R) Register  (D) De-register
Sel   Transaction  PSB name  Reserved user    Region name    Status
_  5   ADDINV      DFSSAM04
_       ADDPART     DFSSAM04
_       APOL11      APOL1
_       APOL12      APOL1
_       APOL13      APOL1
_       APOL14      APOL1
 F1=Help      F3=Exit      F4=IMSIDLst  F7=Backward  F8=Forward  F12=Cancel
```

© Copyright IBM Corp. 1992, 2023

1 IMS System
Specify the IMS subsystem identifier where you debug.

Press F4 to receive a list of IMS subsystems that are set up for IMS Transaction Isolation.

2 Manage additional libraries and delay debug options
Place a forward slash (/) in the entry field and press Enter to display the **Manage Additional Libraries and Delay Debug** panel (EQAPMPRG).

3 Filters
You can use these selections to change the transactions that are displayed for the selected IMS subsystem.

4 Maximum number of transactions
This value limits the number of transactions displayed for the given filter. If there are more transactions matching the filter than the transactions that are displayed, a message is displayed.

Note: If you set too high of a limit or enter 0 to set no limit, the performance of this panel will be degraded considerably.

5 Transaction action character
The following actions can be performed for each transaction listed:

Action	Function	Description
E	Edit	Displays the **Edit pattern-matching parameters** panel (EQAPMPED).
S	Start Region	Starts a private message-processing region based on the current execution environment for the selected transaction. If you do not start the region, it will also register to debug the transaction.
P	Stop Region	Stops the private message-processing region that you started.
R	Register	Register to debug the selected transaction. When a message for the transaction is scheduled in the IMS subsystem, the message is routed to your private message-processing region if all pattern-matching parameters are satisfied.
D	De-register	Removes your registration to debug the selected transaction. Messages are no longer routed to your private message-processing region for this transaction.

5. In the **Manage Additional Libraries and Delay Debug** panel (EQAPMPRG), you can perform the following tasks:

   a. Edit the delay debug options data set.

   b. Specify Language Environment options for the private message region.

   c. Add data sets to the message region STEPLIB concatenation.

   When z/OS Debugger creates your private message-processing region, if you have a delay debug options data set allocated, the private message-processing region is in delay debug mode. This allows you to use the delay debug options data set to control the TEST option that is used and the programs that are trapped.

   If you do not have a delay debug data set allocated, z/OS Debugger creates the private message-processing region with a hardcoded CEEOPTS DD. The hardcoded CEEOPTS DD contains the string TEST (ALL, *,PROMPT,VTAM%userid:*), where userid is your TSO user ID.

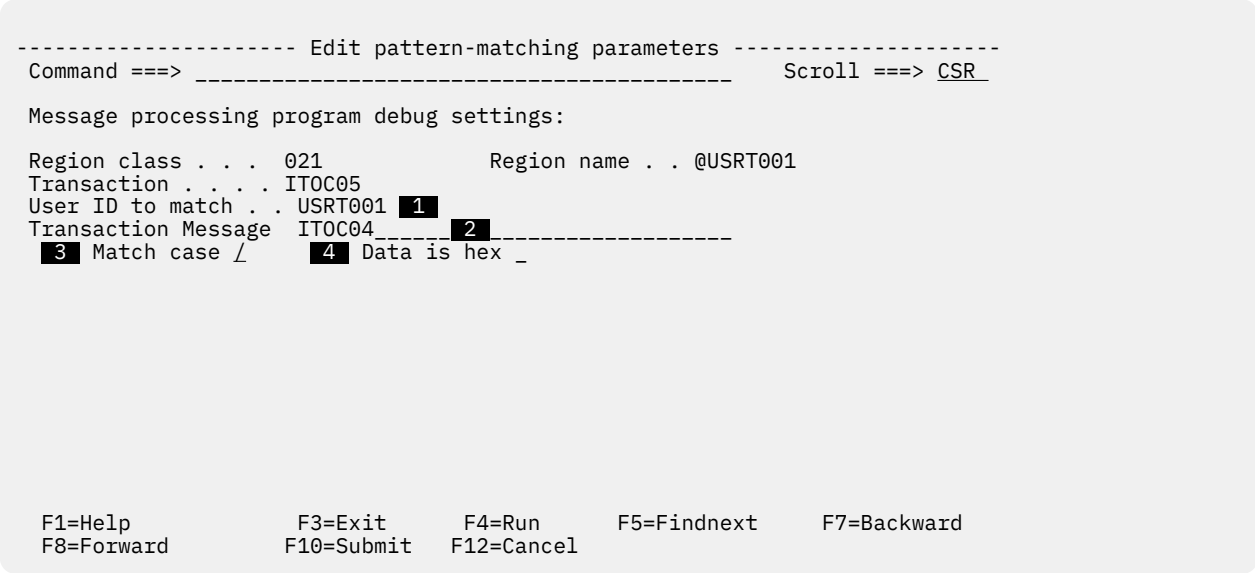
   All private message-processing regions started by IMS Isolation contain a CEEOPTS DD card. You can specify additional Language Environment options for this CEEOPTS DD by using the **Other run-time options** field.

   To add a data set to your private message-processing region's STEPLIB, type an I in the Cmd column of the data set table at the bottom of the panel. This adds an empty line to the table that you can complete with a data set name and a disposition.

Each data set in the table is added to the beginning of the STEPLIB concatenation for the private message-processing region, in the order that is specified in the table. You can change the relative position of the data sets in the table by modifying the values in the Seq column.

For more advanced manipulation of the DD card, you can type a forward slash (/) in the Cmd column for a DD card and press Enter. A menu is displayed where you can change the allocation parameters, the DCB parameters, and other characteristics that are specified on the DD card for a data set.

6. The following screen highlights the fields on the **Edit pattern-matching parameters** panel (EQAPMPED).



1 User ID to match
This field designates the user ID or pattern that is used to match against the user ID when a given instance of the selected transaction is run. The value may be a full user ID or a pattern that ends with the character '*'.

2 Transaction Message
Data that you enter in this field is used to match against all messages that are scheduled for the selected transaction. If the string you type is contained within the message, the message is considered a match, if the other pattern-matching parameters are also satisfied (see 3 and 4).

3 Match case
Place a forward slash (/) in the entry field to indicate that the string in "Transaction Message" is considered a match if all characters match, including their case.

4 Data is hex
Place a forward slash (/) in the entry field to indicate that the string in "Transaction Message" is a hexadecimal string.

Using IMS pseudo wait-for-input (PWFI) with IMS Transaction Isolation

When you debug an IMS application program with the IMS Transaction Isolation facility and the IMS region is using PWFI, z/OS Debugger might be unresponsive if the region is waiting for a message after a GetUnique (GU) call statement is done on the IOPCB.

With PWFI, when a GU call statement is done on the IOPCB with no messages, the IMS region goes into a wait state for a message and z/OS Debugger might appear unresponsive. If you use the **/DIS A** command, a status of WAIT-MESSAGE is displayed. When the status is WAIT-MESSAGE, you cannot stop the IMS region.

To stop the wait state and return to the caller of the program that did the GU on the IOPCB, issue the **IMS PSTOP** command.

```
/PST REG xx TRAN tttttttt
/PST REG JOBNAME jjjjjjjj TRAN tttttttt
```

After the **IMS PSTOP** command is issued, the control is then returned to the caller with a QC status code and you can continue with the program. When the program ends, you can stop the IMS region.

Debugging IMS batch programs interactively by running BTS in TSO foreground

If you want to debug an IMS batch program interactively, you can use full-screen mode using the Terminal Interface Manager or remote debug mode. This topic describes a third option, which is to run BTS in the TSO foreground, by doing the following steps:

1. Define a *dummy* transaction code on the `./'T` command to initiate your program
2. Include a *dummy* transaction in the BTS input stream
3. Start BTS in the TSO foreground.

   FSS is the default option when BTS is started in the TSO foreground, and is available only when you are running BTS in the TSO foreground. FSS can only be turned off by specifying TSO=NO on the ./O command. When running in the TSO foreground, all call traces are displayed on your TSO terminal by default. This can be turned off by parameters on either the `./'0` or `./'T` commands.

Note: If your source (C and C++) or listing (COBOL and PL/I) does not come up in z/OS Debugger when you launch it, check that the source or listing file name corresponds to the MVS library name, and that you have at least read access to that MVS library.

Debugging non-Language Environment IMS BTS programs

If you want to debug a non-Language Environment program that runs in IMS BTS, you can use the EQANIAFE application front-end program, along with the EQASET transaction, to start the debug session.

See the following example and steps to enable EQANIAFE and run EQASET:

```
//BTSITOC5 JOB ,'SYSADM',
//             CLASS=A,TIME=(3,14),MSGLEVEL=(1,1),REGION=128M,
//             NOTIFY=&SYSUID.,MSGCLASS=H
//G         EXEC PGM=BTSRC000,
//   PARM=(DLI,,0000,,0,,N,0,T,IMS1,,N,N,,N,,,'',,,,,,,'','')
//DFSRESLB  DD DISP=SHR,DSN=IMSBLD.I15RTSMM.SDFSRESL
//STEPLIB   DD DISP=SHR,DSN=USER.TEST.LOAD
//          DD DISP=SHR,DSN=EQAW.SEQAMOD
//          DD DISP=SHR,DSN=CEEV2R3Z.SCEERUN
//          DD DISP=SHR,DSN=CEEV2R3Z.SCEERUN2
//          DD DISP=SHR,DSN=IMSTOOL.BTS41.SBTSLMD0
//          DD DISP=SHR,DSN=IMSBLD.I15RTSMM.SDFSRESL
//*
//BTSIN   DD *
./E  APPLFE=EQANIAFE                                          1
./*  TC=BTSTERM MDL=P2
./D  LTERM=BTSTERM TYPE=3270-A2 SIZE=(24,80) LIMIT=0
./T  TC=EQASET MBR=EQANISET PSB=ITOC05 LANG=CBL TYPE=MSG      2
./T  TC=ITOC05 MBR=ITOC05 PSB=ITOC05 LANG=CBL TYPE=MSG
./* ESTABLISH A DEBUG ASSOCIATION FOR THIS TERMINAL
EQASET TCP=ON $                                               3
./* CLEAR THE TERMINAL SCREEN BEFORE YOUR TRAN
CLEAR                                                         4
ITOC05 ITOC05 $                                               5
./* REMOVE THE DEBUG ASSOCIATION
EQASET TCP= $                                                 6
/*
//EQANMDBG  DD *
ITOC05,TEST(ERROR,CMDS,PROMPT,TCPIP&9.85.213.175%8002:*)      7
/*
```

1. Identify an application front end for the BTS environment via the **./E APPLFE** command.
2. Add a transaction definition for EQASET.

   Note: EQANISET is normally defined as a GPSB application, so you must substitute one of your PSBs for the PSB value in the **./T** command.
3. Invoke EQASET to associate the BTS terminal with a debug preference. In the example above, the preference merely states that debugging is turned on. The actual debug preferences are supplied via the EQANMDBG DD card (7). This allows greater flexibility with the TEST runtime option, such as supplying an initial commands file (CMDS). In a simpler case, you can use the full syntax of EQASET to set up the destination for the debug session (TCP, VTAM, MFI) and let the rest of the TEST runtime options use the default values. For more information about the EQASET transaction, see "Syntax of the EQASET transaction for non-Language Environment MPPs" on page 342.
4. Clear the virtual screen by using the **CLEAR BTS** command.
5. Invoke your transaction. Debugging starts by using the TEST runtime options that you specified in the EQANMDBG DD card (7).
6. After debugging is complete, invoke EQASET to remove the debug association.

Debugging IMS batch programs in batch mode

You can use z/OS Debugger to debug IMS programs in batch mode. The debug commands must be predefined and included in one of the z/OS Debugger commands files, or in a command string. The command string can be specified as a parameter either in the TEST run-time option, or when `CALL CEETEST` or `__ctest` is used. Although batch mode consumes fewer resources, you must know beforehand exactly which debug commands you are going to issue. When you run BTS as a batch job, the batch mode of z/OS Debugger is the only mode available for use.

For example, you can allocate a data set, `userid.CODE.BTSINPUT` with individual members of test input data for IMS transactions under BTS.

Debugging non-Language Environment IMS MPPs

You can debug IMS message processing programs (MPPs) that do not run in Language Environment by doing the following tasks:

1. Verify that your system is configured correctly and start a new region. See "Verifying configuration and starting a region for non-Language Environment IMS MPPs" on page 341 for instructions.
2. Choose a debugging interface. See "Choosing an interface and gathering information for non-Language Environment IMS MPPs" on page 342 for instructions.
3. Run the EQASET transaction, which identifies the debugging interface you chose and enables debugging. See "Running the EQASET transaction for non-Language Environment IMS MPPs" on page 342.
4. Start the IMS transaction that is associated with the program you want to debug.

After you finish debugging your program, you can do one of the following:

- Continue debugging another program.
- Disable debugging and continue running the region for other tasks.
- Disable debugging and shut down the region. If you want to debug an IMS programs, you have to repeat tasks 2 to 4.

Verifying configuration and starting a region for non-Language Environment IMS MPPs

Before you debug an IMS MPP that does not run in Language Environment, do the following steps:

1. Consult with your system administrator and verify that your system has been configured to debug IMS programs that do not run in Language Environment. See the *IBM z/OS Debugger Customization Guide*

for instructions on how to include the APPLFE=EQANIAFE parameter string in the JCL that starts a region and EQANISET.

2. Start an IMS message processing region (MPR) that runs the EQANIAFE application front-end routine whenever a message processing program (MPP) is scheduled.

After you complete these steps, choose a debugging interface as described in "Choosing an interface and gathering information for non-Language Environment IMS MPPs" on page 342.

Choosing an interface and gathering information for non-Language Environment IMS MPPs

Choose from one of the following debugging interfaces and gather the indicated information:

- Use full-screen mode using a dedicated terminal without Terminal Interface Manager. Obtain the terminal LU for this terminal. For example, TRMLU001. If you are required to use the VTAM network identifier for the terminal LU, obtain this information from your system programmer.
- Use full-screen mode using the Terminal Interface Manager. Obtain the user ID. For example, USERABCD.
- Use remote debug mode. Obtain the IP address and port number that the remote debugger is listening to.

After you choose a debugging interface, run the EQASET transaction as described in "Running the EQASET transaction for non-Language Environment IMS MPPs" on page 342.

Running the EQASET transaction for non-Language Environment IMS MPPs

Running the EQASET transaction indicates to the EQANIAFE application front-end routine that you want to do one of the following functions:

- Enable a debugging session with the preferences you indicate
- Request information about your existing preferences
- Disable a debugging session

To enable a debugging session, select one of the following options:

- To debug in full-screen mode using a dedicated terminal without Terminal Interface Manager, enter the command EQASET MFI=*terminal_LU_name*. If you are required to specify a VTAM network identifier, enter the command EQASET MFI=*network_identifier.terminal_LU_name*.
- To debug in full-screen mode using the Terminal Interface Manager, enter the command EQASET VTAM=*user_ID*.
- To debug in remote debug mode, enter the command EQASET TCP=*IP_address%port_number*.

After you enter an EQASET command, on the same terminal, start the transaction that is associated with the application program that you want to debug.

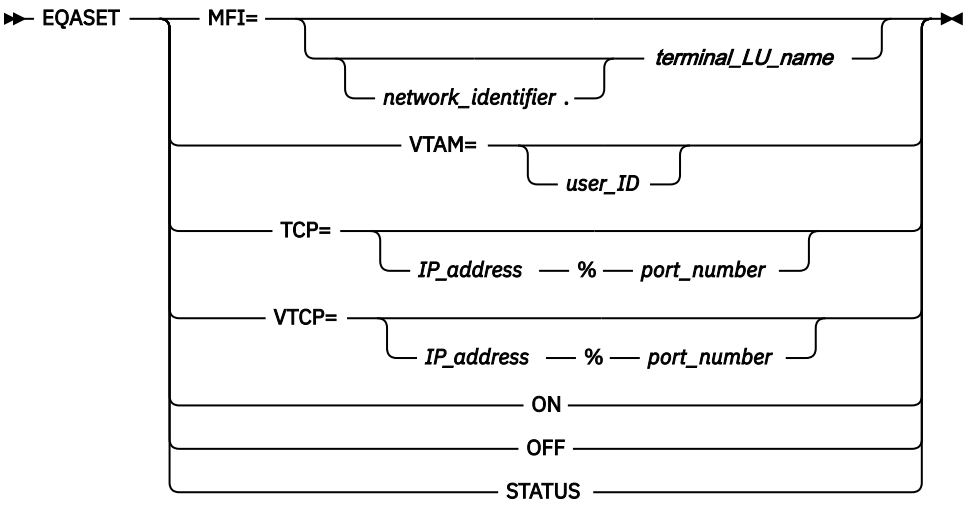
To request information about your existing preferences, enter the command EQASET STATUS.

To disable a debugging session, enter the command EQASET OFF.

To re-enable a debugging session after using EQASET OFF, enter the command EQASET ON.

Syntax of the EQASET transaction for non-Language Environment MPPs

The following diagram displays the syntax of the EQASET transaction for non-Language Environment MPPs:



The EQASET transaction manages a separate debugging setting for each user that runs the transaction. Each setting is identified by the user ID that is used to log on to the terminal where the transaction is run. For any user ID, only the last debugging preference (MFI, TCP, VTCP, or VTAM) entered is saved. You can use the STATUS option to see the current debugging preference.

The following TEST runtime option string is constructed with the debugging preference:

```
TEST(ALL,INSPIN,,debuggingPreference:*)
```

You cannot customize the other runtime options.

MFI=
Use full-screen mode using a dedicated terminal without Terminal Interface Manager. You must specify a dedicated terminal LU name for the debug session. If your site requires that you specify the VTAM network identifier, prefix the name of the VTAM network identifier to the terminal LU name. Without specifying the terminal LU name, debugging is turned off. No space is allowed after the equal sign (=). The preference implies debugging is turned on.

VTAM=
Use full-screen mode using the Terminal Interface Manager. You must specify the user ID that was used to log on to the Terminal Interface Manager. Without specifying the user ID, debugging is turned off. No space is allowed after the equal sign (=). The preference implies debugging is turned on.

TCP= or VTCP=
Use remote debug mode. Specify the TCP/IP address and port number of the workstation where the remote debug daemon is running. Without specifying the IP address and port number, debugging is turned off. No space is allowed after the equal sign (=). The preference implies debugging is turned on. You can specify the TCP/IP address in one of the following formats:

IPv4
You can specify the address as a symbolic address, such as `some.name.com`, or a numeric address, such as 9.112.26.333.

IPv6
You must specify the address as a numeric address, such as 1080:0:FF::0970:1A21. If you use IPv6 format, you must use the TCP= option; you cannot use the VTCP= option.

ON
Turn on debugging. This is valid only when a debugging preference (MFI, TCP, VTCP, or VTAM) has been set.

OFF
Turn off debugging.

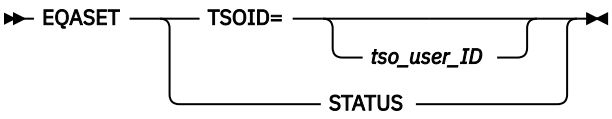
STATUS
Display the current debugging preference. The EQASET transaction displays only the first 25 characters of the IP address.

Debugging Language Environment IMS MPPs without issuing / SIGN ON

The Language Environment user exit for EQAD3CXT constructs the name of an MVS data set that contains the Language Environment runtime options, including the TEST runtime option. EQAD3CXT constructs the name of the MVS data set by assigning values to tokens that represent each qualifier in a data set name; it assigns the *IMS* user ID as the value for the &USERID token. However, if you do not sign on to IMS (by using `/SIGN ON`), the IMS user ID is either the same as the IMS LTERM ID or it is not defined. In either case, EQAD3CXT cannot locate the MVS data set. To specify that EQAD3CXT assigns a *TSO* user ID as the value for the &USERID token, run the EQASET transaction specifying the `TSOID` option. For a description of the EQASET transaction with the `TSOID` option, see "Syntax of the EQASET transaction for Language Environment MPPs" on page 344.

Syntax of the EQASET transaction for Language Environment MPPs

The following diagram displays the syntax of the EQASET transaction for Language Environment MPPs:



When you use the EQASET transaction for Language Environment MPPs, it associates the current IMS LTERM ID with the specified TSO user ID. EQADICXT can construct a valid name for the MVS data set using the TSO user ID for the &USERID token.

TSOID=
Identify a TSO user ID to use in place of the &USERID token in the Language Environment user exit. The TSO user ID must match the user ID used to create the data set name, as described in "Creating and managing the TEST runtime options data set" on page 101.

STATUS
Display the current value for `TSOID`.

This option might also display information about debugging preferences for non-Language Environment MPPs.

Creating setup file for your IMS program by using IBM z/OS Debugger Utilities

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can create setup files for your IMS Batch Messaging Process (BMP) program which describe how to create a custom region and defines the STEPLIB concatenation statements that reference the data sets for your IMS program's load module and the z/OS Debugger load module. You can also create and customize a setup file to create a private message region that you can use to test your IMS message processing program (MPP). Creating a private message region with class X allows you to test your IMS program run by transaction X and reduce the risk of interfering with other regions being used by other IMS programs.

To create a setup file for your IMS program by using IBM z/OS Debugger Utilities, do the following steps:

1. Start IBM z/OS Debugger Utilities. If you do not know how to start IBM z/OS Debugger Utilities, see "Starting IBM z/OS Debugger Utilities" on page 9.

2. In the IBM z/OS Debugger Utilities panel (EQA@PRIM), type 4 in the Option line and press Enter.
3. In the Manage IMS Programs panel (EQAPRIS), type 2 in the Option line and press Enter.
4. In the Create Private Message Regions - Edit Setup File panel (EQAPFORA), type in the information to create a new setup file or edit an existing setup file. Press Enter.

   Create a private message region to customize your application or z/OS Debugger libraries while you debug your application so that you do not impact other user's activities. Consult your system administrator for authorization and rules regarding the creation of private message regions.

   After you specify the setup information required to run your IMS program, you can specify the information needed to create a private message region you can use to test your IMS program or specify how to run a BMP program. To specify this setup information, do the following steps:
5. In the Edit Setup File panel (EQAPFORI), type in the information to start IMS batch processor. Type a forward slash (/) in the field Enter / to modify parameters, then press Enter to modify parameters for the batch processor.
6. In the Parameters for IMS Procedures panel (EQAPRIPM), use one of the following values in the TYPE field to indicate which action you want done:
   - MSG to start a private message region.
   - BMP to run a BMP program.

   Enter other parameters as needed. Press PF1 for information about the parameters.
7. After you type in the specifications, you can submit your job for processing by pressing PF10.

Using IMS message region templates to dynamically swap transaction class and debug in a private message region

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can use predefined IMS message region templates to debug a specific transaction in a private message region by using IBM z/OS Debugger Utilities option 4.3 Swap IMS Transaction Class and Run Transaction (panel EQAPMPRS). This panel and its sub-panels allow you to take the following actions:

1. Start a private message region from a predefined message region template. This template specifies a message class that is reserved for debug purposes.
2. Assign a transaction that you want to debug to the class for the private message region.
3. Schedule a message for the transaction.
4. After you have finished debugging the transaction and it completes, the transaction is assigned to its original class and the private message region is stopped.

To dynamically launch a private message region and run a specific transaction in that region, complete the following steps:

1. Start IBM z/OS Debugger Utilities. For detailed information, see “Starting IBM z/OS Debugger Utilities” on page 9.
2. In the IBM z/OS Debugger Utilities panel (EQA@PRIM), type 4 in the Option line and press Enter.
3. In the Manage IMS Programs panel (EQAPRIS), type 3 in the Option line and press Enter.
4. In the Debug IMS Transaction - Select Private Message Region panel (EQAPMPRS), type a forward slash (/) beside the template you want to use, and press Enter. You can choose from the following types of templates:
   - Predefined templates from a common z/OS Debugger Setup Utility data set
   - Templates previously customized and stored in a private z/OS Debugger Setup Utility data set

If you use a member from a private z/OS Debugger Setup Utility data set, you can see the Create Private Message Regions - Edit Setup File panel (EQAPFORA). Enter the information to edit an existing setup file.

5. In the Specify Transaction and Additional Test Libraries panel (EQAPMPRT), type the transaction name that you want to launch in your private message region. You also need to enter any additional information to send when the message is scheduled.

   You might want to add data sets to the message region STEPLIB concatenation. To add a data set, type an I in the Cmd column of the data set table at the bottom of the panel. This adds an empty line to the table that you can fill in with a data set name and a disposition.

   Each data set in the table is added to the beginning of the STEPLIB concatenation for the message region, in the order specified in the table. You might change the relative position of the data sets in the table by modifying the values in the Seq column.

   For more advanced manipulation of the DD card, you can type a forward slash (/) in the Cmd column for a DD card and press Enter. A menu is displayed where you can change the allocation parameters, the DCB parameters, and other characteristics that are specified on the DD card for a data set.

6. To start the private message region and schedule the transaction, run the z/OS Debugger IMS Transaction Swap Utility (the EQANBSWT Batch Message Program, hereafter referred to as EQANBSWT). This can be done in one of the two following ways:

   - Press PF4 to run the transaction. This starts EQANBSWT in the foreground of your TSO session.
   - Press PF10 to submit. This displays a JCL deck that runs the EQANBSWT program that you can submit to the Job Entry System by using the ISPF SUBMIT command.

   EQANBSWT will start the private message region. By default, the TEST parameter will be the following:

   ```
   TEST(ALL,*,PROMPT,VTAM%userid:*)
   ```

   The *userid* is your TSO user ID.

   If you want to use a different TEST parameter, type a forward slash (/) beside the **Enter / to modify parameters** field, and press Enter. The **EQAPFMTP** panel is displayed. Specify the TEST parameter sub-options and session type, and press PF3 to save.

   EQANBSWT will also start a second private message region, by using the NOTEST parameter, and serving the same class. This region allows additional messages scheduled for the transaction to be processed when the transaction is being debugged in the TEST region at the same time.

   EQANBSWT will then assign the transaction to the class served by the private message region and schedule the transaction.

   When the transaction completes, EQANBSWT stops the private message regions and assigns the transaction to the class to which it was initially assigned.

   The jobs that are started to run EQANBSWT and the two private message regions use the job card you specified in IBM z/OS Debugger Utilities option 0, Job Card. Each job name is replaced by values that you entered in Debug Utilities option 4.0, Set IMS Program Options. If you do not set personal defaults in option 4.0, system defaults are used.

   In certain circumstances, EQANBSWT does not complete normally. To interrupt EQANBSWT, take one of the following steps:

   - If you ran EQANBSWT in the foreground by using the Run command, press the ATTN or PA1 key and follow the prompts to stop the process.
   - If you ran EQANBSWT as a batch job by using the Submit command, issue the STOP *jobname* MVS command, for example, by typing /P *jobname* in the Spool Display and Search Facility (SDSF).

7. When you want to leave the Specify Transaction and Additional Test Libraries panel (EQAPMPRT), you can save any changes you have made into a private message region template.

- If you selected a predefined message template in step 4, type SAVE AS and press Enter. This displays the z/OS Debugger Foreground – Edit Setup File panel (EQAPFOR), where you can enter a data set name for your private copy of the template.
- Otherwise, press PF3 to Exit. Your changes are saved to the private template you opened in step 4.

Placing breakpoints in IMS applications to avoid the appearance of z/OS Debugger becoming unresponsive

When you debug an IMS application program, the way IMS manages resources might occasionally make z/OS Debugger appear unresponsive. To avoid this situation, set breakpoints as close as possible to the location that you need to debug or at the GetUnique (GU) call statement. The information in this topic helps you understand how IMS's management of resources might appear to make z/OS Debugger unresponsive and helps you determine the approximate location to set a breakpoint to avoid this situation.

After you start an IMS transaction, IMS loads and runs the application program associated with that transaction. IMS manages all the messages requested by and returned to that application program, along with all the messages requested by and returned to other application programs running at the same time. IMS uses the processing limit count (PLCT) and other tools to ensure that application programs get the appropriate share of resources. As long as your IMS application program does not exceed the PLCT[11], it continues running and processing messages or waiting for the next message. However, if you are trying to debug the application program, the continued message processing or waiting for messages might make z/OS Debugger appear unresponsive. To avoid this situation, try one of the following options at the beginning of your debug session, before you begin running the application program (for example, by entering the G0 command):

- Set a breakpoint as close as possible to the area you want to debug.
- Set a breakpoint at the GU call statement.

Related references

IMS System Definition Reference

[11] IMS Quick reschedule allows application programs to process more than the PLCT for each physical schedule. Quick reschedule helps eliminate processing overhead caused by unnecessary rescheduling and reloading of application programs.

Chapter 39. Debugging CICS programs

This topic describes tasks you can do while debugging CICS programs, and describes some restrictions.

Before you can debug your programs under CICS, verify that you have completed the following tasks:

- Ensured that all of the required installation and configuration steps for CICS Transaction Server, Language Environment, and z/OS Debugger have been completed. For more information, refer to the installation and customization guides for each product.
- Completed all the tasks in the following topics:
  - Chapter 4, “Planning your debug session,” on page 25
  - Chapter 5, “Updating your processes so you can debug programs with z/OS Debugger,” on page 59
  - Chapter 10, “Preparing a CICS program,” on page 81
  - Chapter 18, “Starting z/OS Debugger under CICS,” on page 139

Displaying the contents of channels and containers

You can display the contents of CICS channels by using the `DESCRIBE CHANNEL` command and the contents of a container by using the `LIST CONTAINER` command.

The section "Enhanced inter-program data transfer: channels as modern-day COMMAREAs" in the *CICS Application Programming Guide* describes the benefits of containers and channels and how to use them in your programs.

To display a list of containers in the current channel, enter the command `DESCRIBE CHANNEL`. To display a list of containers in another channel, enter the command `DESCRIBE CHANNEL` *`channel_name`*, where *`channel_name`* is the name of a specific channel. In either case, z/OS Debugger displays a list similar to the following list:

© Copyright IBM Corp. 1992, 2023

```
COBOL    LOCATION: ZCONPRGA :> 274
Command ===>                                                  Scroll ===> PAGE
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 2
****************************** TOP OF MONITOR *******************************
                                ----+----1----+----2----+----3----+----4----
0001  1 ********** AUTOMONITOR **********
0002    01 DFHC0160               'PrgA-ChanB-ContC'
***************************** BOTTOM OF MONITOR *****************************

SOURCE: ZCONPRGA -1----+----2----+----3----+----4----+----5--- LINE: 272 OF 307
     272       *              FLENGTH(LENGTH OF PrgA-ChanB-XXXXX)             .
     273       *              END-EXEC                                        .
     274            Move 'PrgA-ChanB-ContC' to dfhc0160                       .
     275            Move 'PrgA-CHANB' to dfhc0161                             .
     276            Call 'DFHEI1' using by content x'34167000072000000200000' .
     277       -    '00f0f0f0f5f3404040' by content x'0000' by reference      .
     278            PrgA-ChanB-XXXXX by reference dfhc0160 by content LENGTH  .
     279            PrgA-ChanB-XXXXX by content x'0000' by content x'0000' by .
     280            content x'0000' by content x'0000' by content x'0000' by  .
     281            content x'0000' by content x'0000' by content x'0000' by  .
     282            content x'0000' by content x'0000' by content x'0000' by  .
     283            content x'0000' by content x'0000' by content x'0000' by  .
LOG 0----+----1----+----2----+----3----+----4----+----5----+-- LINE: 147 OF 289
0147  DESCRIBE CHANNEL * ;
0148 CHANNEL                 PrgA-ChanB
0149 CONTAINER NAME                  SIZE
0150 ------------------------------------
0151 PrgA-ChanB-ContC                 21
0152 PrgA-ChanB-ContB                 21
0153 PrgA-ChanB-ContA                 21
0154 CHANNEL                 PRGA-CHANA
0155 CONTAINER NAME                  SIZE
0156 ------------------------------------
0157 PRGA-CHANA-CONTC                 21
PF  1:?          2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP         8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

To display the contents of a container in the current channel, enter the command `LIST CONTAINER` *`container_name`*, where *container_name* is the name of a particular channel. To display the contents of a container in another channel, enter the command `LIST CONTAINER` *`channel_name container_name`*, where *channel_name* is the name of another channel. In either case, z/OS Debugger displays the contents of the container in a format similar to the following diagram:

```
COBOL    LOCATION: ZCONPRGA :> 211.1
Command ===>                                                  Scroll ===> PAGE
MONITOR -+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 2
****************************** TOP OF MONITOR ********************************
                                ----+----1----+----2----+----3----+----4----
0001  1 ********** AUTOMONITOR **********
0002    01 DFHC0160               'PRGA-CHANA-CONTC'
***************************** BOTTOM OF MONITOR ******************************

SOURCE: ZCONPRGA -1----+----2----+----3----+----4----+----5--- LINE: 209 OF 307
     209       *              FLENGTH(LENGTH OF PrgA-ChanB-ContA)             .
     210       *              END-EXEC                                        .
     211            Move 'PrgA-ChanB-ContA' to dfhc0160                       .
     212            Move 'PrgA-ChanB' to dfhc0161                             .
     213            Call 'DFHEI1' using by content x'341670000720000002000000 .
     214       -    '00f0f0f0f3f5404040' by content x'0000' by reference      .
     215            PrgA-ChanB-ContA by reference dfhc0160 by content LENGTH  .
     216            PrgA-ChanB-ContA by content x'0000' by content x'0000' by .
     217            content x'0000' by content x'0000' by content x'0000' by  .
     218            content x'0000' by content x'0000' by content x'0000' by  .
     219            content x'0000' by content x'0000' by content x'0000' by  .
     220            content x'0000' by content x'0000' by content x'0000' by  .
LOG 0----+----1----+----2----+----3----+----4----+----5----+---- LINE: 15 OF 25
0015  STEP ;
0016  DESCRIBE CHANNEL * ;
0017 CHANNEL                 PRGA-CHANA
0018 CONTAINER NAME                SIZE
0019 ----------------------------------
0020 PRGA-CHANA-CONTC                21
0021 PRGA-CHANA-CONTB                21
0022 PRGA-CHANA-CONTA                21
0023  LIST CONTAINER PRGA-CHANA PRGA-CHANA-CONTC ;
0024 000C7F78  D7D9C7C1 60C3C8C1 D5C160C3 D6D5E3C3   *PRGA-CHANA-CONTC*
0025 000C7F88  60C4C1E3 C1                           *-DATA           *
PF  1:?         2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP        8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

Refer to the following topics for more information related to the material discussed in this topic.

- **Related references**
- DESCRIBE CHANNEL command in *IBM z/OS Debugger Reference and Messages*
- LIST CONTAINER command in *IBM z/OS Debugger Reference and Messages*

Controlling pattern-match breakpoints with the DISABLE and ENABLE commands

This topic describes how you can use the DISABLE and ENABLE commands to control pattern-match breakpoints. A *pattern-match* breakpoint is a breakpoint that is identified by the name, or part of the name, of a load module or compile unit specified in a DTCN or CADP profile.

The DISABLE command works with the debugging profile that started the current debugging session to prevent programs from being debugged. When you enter the DISABLE command, you specify the name, or part of the name, of a load module, compile unit, or both, that you do not want to debug. When z/OS Debugger finds a load module, compile unit, or both, whose name matches the name or part of the name (a pattern) that you specified, z/OS Debugger does not debug that program. When you enter the ENABLE command, you specify the pattern (the full name or part of a name of a load module, compile unit, or both) that you want to debug. The pattern must match the name of a load module, compile unit, or both, that you specified in a previously entered DISABLE command.

Before you begin, verify that you know which debugging profile started z/OS Debugger (DTCN or CADP) and the names you specified in the LoadMod::>CU field (for DTCN) or the Program field, Compile Unit field, or both (for CADP).

To use the DISABLE command to prevent z/OS Debugger from debugging a program, do the following steps:

1. If you don't remember what programs you might have disabled, enter the command `LIST DTCN` or `LIST CADP`. This command lists the programs you have already disabled. This step reminds you of the names of load modules, programs, or compile units you already disabled.

2. If you are running with a CADP profile, enter the command `DISABLE CADP PROGRAM program_name CU compile_unit_name`. *program_name* is the name of the program, or it matches the pattern of the name of a program, that you specified in the Program field and it is the program that you do not want to debug. *compile_unit_name* is the name of the compile unit, or it matches the pattern of the name of a compile unit, that you specified in the Compile Unit field and it is the compile unit that you do not want to debug. You can specify `PROGRAM program_name`, `CU compile_unit_name`, or both.

   For example, if you have the following circumstances, enter the command `DISABLE CADP PROGRAM ABD2` to prevent z/OS Debugger from debugging the program ABD2:

   - You specified ABD* in the Program field of the profile.
   - You have programs with the name ABD1, ABD2, ABD3, ABD4, and ABD5.

3. If you are running with a DTCN profile, enter the command `DISABLE DTCN LOADMOD load_module_name CU compile_unit_name`. *load_module_name* is the name of the load module, or it matches the pattern of the name of a load module, that you specified in the LoadMod field and it is the load module that you do not want to debug. *compile_unit_name* is the name of the compile unit, or it matches the pattern of the name of a compile unit, that you specified in the CU field and it is the compile unit that you do not want to debug. You can specify `LOADMOD load_module_name`, `CU compile_unit_name`, or both.

   For example, if you have the following circumstances, enter the command `DISABLE DTCN CU STAR2` to prevent z/OS Debugger from debugging the compile unit STAR2:

   - You specified STAR* in the CU field of the profile.
   - You have compile units with the names STAR1, STAR2, STAR3, STAR4, and STAR5.

To use the ENABLE command to allow a previously disabled program to be debugged, do the following steps:

1. If you don't remember the exact name of the disabled load module, program, or compile unit, enter the command `LIST DTCN` or `LIST CADP`. This command lists the programs you have disabled. Write down the name of the load module, program, or compile unit that you want to debug.
2. If you are running with a CADP profile, enter the command `ENABLE CADP PROGRAM program_name CU compile_unit_name`, where *program_name* is the name of the program and *compile_unit_name* is the name of the compile unit that you wrote down from step 1. If you only need to specify a program name, you do not have to type in the `CU compile_unit_name` portion of the command. If you only need to specify a compile unit name, you do not have to type in the `PROGRAM program_name` portion of the command.
3. If you are running with a DTCN profile, enter the command `ENABLE DTCN LOADMOD load_module_name CU compile_unit_name`, where *load_module_name* is the name of the load module and *compile_unit_name* is the name of the compile unit you wrote down from step 1. If you only need to specify a load module name, you do not have to type in the `CU compile_unit_name` portion of the command. If you only need to specify a compile unit name, you do not have to type in the `LOADMOD load_module_name` portion of the command.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

DISABLE command in *IBM z/OS Debugger Reference and Messages*

ENABLE command in *IBM z/OS Debugger Reference and Messages*

LIST CADP or DTCN command in *IBM z/OS Debugger Reference and Messages*

Preventing z/OS Debugger from stopping at EXEC CICS RETURN

z/OS Debugger stops at `EXEC CICS RETURN` and displays the following message:

```
CEE0199W The termination of a thread was signaled due to a STOP statement.
```

To prevent z/OS Debugger from stopping at every EXEC CICS RETURN statement in your application and suppress this message, set the TEST level to ERROR by using the SET TEST ERROR command.

Early detection of CICS storage violations

CICS can detect various types of storage violations. The *CICS Problem Determination Guide* describes the types of storage violations that CICS can detect and when CICS detects them automatically. You can request that z/OS Debugger detect one type of storage violation (whether the storage check zone of a user-storage element has been overlaid). You can make this request at any time.

To instruct z/OS Debugger to check for storage violations, enter the command CHKSTGV. z/OS Debugger checks the task that you are debugging for storage violations.

You can instruct z/OS Debugger to check for storage violations more frequently by including the command as part of a breakpoint. For example, the following commands check for a storage violation at each statement in a COBOL program and causes z/OS Debugger to stop if a violation is detected in the current procedure:

```
AT STATEMENT *
  PERFORM
    CHKSTGV ;
    IF %RC = 0 THEN
      GO ;
    END-IF ;
  END-PERFORM ;
```

If you plan on running a check at every statement, run it on as few statements as possible because the check causes overhead that can affect performance.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
CICS Problem Determination Guide

Saving settings while debugging a pseudo-conversational CICS program

If you change the z/OS Debugger display settings (for example, color settings) while you debug a pseudo-conversational CICS program, z/OS Debugger might restore the default settings. To ensure that your changes remain in effect every time your program starts z/OS Debugger, store your display settings in the preferences file or the commands file.

Saving and restoring breakpoints and monitor specifications for CICS programs

When you set any of the following specifications to AUTO, these specifications are used to control the saving and restoring of breakpoints and LOADDEBUGDATA specifications between z/OS Debugger settings:

- SAVE BPS
- SAVE MONITORS
- RESTORE BPS
- RESTORE MONITORS

You set switches by using the SET command. The SAVE BPS and SAVE MONITORS switches enable the saving of breakpoints and monitor specifications between debugging sessions. The RESTORE BPS and RESTORE MONITORS switches control the restoring of breakpoints and monitor specifications at the start of subsequent debugging sessions. Setting these switches to AUTO enables the automatic saving and restoring of this information. You must also enable the SAVE SETTING AUTO switch so that these settings are in effect at the start of subsequent debugging sessions.

While you run in CICS, consider the following requirements:

- You must log on as a user other than the default user.
- The CICS region must have update authorization to the SAVE SETTINGS and SAVE BPS data sets.

When you activate a DTCN profile for a full-screen debugging session and SAVE BPS, SAVE MONITORS, RESTORE BPS, and RESTORE MONITORS all specify NOAUTO, z/OS Debugger saves most of the breakpoint and LOADDEBUGDATA information for that session into the profile. When the DTCN profile is deleted, the breakpoint and LOADDEBUGDATA information is deleted.

See "Performance considerations in multi-enclave environments" on page 182 for information about performance savings and restoring settings, breakpoints, and monitors under CICS.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
IBM z/OS Debugger Reference and Messages

Restrictions when debugging under CICS

The following restrictions apply when debugging programs with the z/OS Debugger in a CICS environment.

- You can use CRTE terminals only in single terminal mode and screen control mode. You cannot use them in separate terminal mode.
- The `__ctest()` function with CICS does nothing.
- The CDT# transaction is a z/OS Debugger service transaction used during separate terminal mode debugging and is not intended for activation by direct terminal input. If CDT# is started via terminal entry, it will return to the caller (no function is performed).
- Applications that issue EXEC CICS POST cannot be debugged in separate terminal mode or screen control mode.
- References to DD names are not supported. All files, including the log file, USE files, and preferences file, must be referred to by their full data set names.
- The commands TSO, SET INTERCEPT, and SYSTEM cannot be used.
- CICS does not support an attention interrupt from the keyboard.
- The CICS region must have read authorization to the preferences and commands files.
- If the EQAOPTS LOGDSN command does not specify a naming pattern, z/OS Debugger does not automatically start the log file. You need to run the SET LOG ON *fileid* command.

  If the EQAOPTS LOGDSN command specifies a naming pattern, z/OS Debugger automatically starts the log file by running the SET LOG ON *fileid* command.

  If you are not logged into CICS or are logged in under the default user ID, z/OS Debugger does not run the EQAOPTS LOGDSN command; therefore, z/OS Debugger does not automatically start a log file.

  The CICS region must have update authorization to the log file.
- Ensure that you allocate a log file big enough to hold all the log output from a debug session, because the log file is truncated after it becomes full. (A warning message is not issued before the log is truncated.)
- z/OS Debugger disables Omegamon RLIM processing for any CICS task which is being debugged.
- You can start z/OS Debugger when a non-Language Environment assembler or non-Language Environment COBOL program under CICS starts by defining a debug profile by using CADP or DTCN. But z/OS Debugger will only start on a CICS Link Level boundary, such as when the first program of the task starts or for the first program to run at a new Link Level. For profiles defined in CADP or DTCN which list a non-Language Environment assembler or non-Language Environment COBOL program name that is dynamically called using EXEC CICS LOAD/CALL, z/OS Debugger will not start. Non-Language Environment assembler or non-Language Environment COBOL programs that are called in this way are identified by z/OS Debugger in an already-running debugging session and can be stopped by using a

command like AT APPEARANCE or AT ENTRY. However, they cannot be used to trigger a z/OS Debugger session initially.

Accessing CICS resources during a debugging session

You can gain access to CICS temporary storage and transient data queues during your debugging session by using the CALL %CEBR command. You can do all the functions you can currently do while in the CICS-supplied CEBR transaction. For access to general CICS resources (for example, information about the CICS system you are debugging on or opening and reading a VSAM file) you can use the CALL %CECI command. This command gives control to the CICS-supplied CECI transaction. Press PF3 from inside CEBR or CECI to return to the debug session. For more information about CEBR and CECI, see *CICS Supplied Transactions*.

Accessing CICS storage before or after a debugging session

You can uses the DTST transaction to display and modify CICS storage. See Appendix H, “Displaying and modifying CICS storage with DTST,” on page 497 for more information.

Chapter 40. Debugging ISPF applications

Debugging ISPF applications presents some challenges to the user because of the way ISPF application programs are invoked. The two main challenges are as follows:

- Providing TEST runtime options to the application.
- Choosing a display device for your z/OS Debugger session.

You need to provide TEST runtime options. This can be done in one of the following ways:

- Edit the exec or panel that invokes the application and change the parameter string that is passed to the program to add the TEST runtime options.
- Allocate a CEEOPTS DD that contains the TEST runtime options.
- Edit the application source code to add a call to CEETEST.

  This method provides the simplest way to debug only the ISPF application subroutine that you want to debug.

You need to select a display device for your z/OS Debugger session. This can be done in one of the following ways:

- Specify a display device by using the TEST runtime options.
  - Use the same 3270 terminal as ISPF is using. When you run your program, specify the `MFI` suboption of the TEST runtime option. The `MFI` suboption requires no additional values if you are going to use the same 3270 terminal as ISPF is using.

    ```
    TEST(ALL,*,PROMPT,MFI:*)
    ```

    PA2 refreshes the ISPF application panel and removes residual z/OS Debugger output from the emulator session. However, if z/OS Debugger sends output to the emulator session between displays of the ISPF application panels, you need to press PA2 after each ISPF panel displays.

    When you debug ISPF applications or applications that use line mode input and output, issue the SET REFRESH ON command. This command is executed and is displayed in the log output area of the Command/Log window.
  - Use a separate 3270 terminal using full-screen mode using the Terminal Interface Manager (TIM).

    When you run your program, specify the `VTAM` suboption of the TEST runtime option. The `VTAM` suboption requires that you specify your user ID, as in the following example:

    ```
    TEST(ALL,*,PROMPT,VTAM%user_id:*)
    ```
  - Use a separate 3270 terminal using full-screen mode using a dedicated terminal without Terminal Interface Manager.

    When you run your program, specify the `MFI` suboption of the TEST runtime option. The `MFI` suboption requires that you specify the VTAM LU name of the separate terminal that you started, as in the following example:

    ```
    TEST(ALL,*,PROMPT,MFI%terminal_id:*)
    ```
  - Use remote debug mode and a remote IDE.

    When you run your program, specify the `TCPIP` suboption of the TEST runtime option. The `TCPIP` suboption requires that you specify the TCP/IP address of your workstation, as in the following example:

    ```
    TEST(ALL,*,PROMPT,TCPIP&tcpip_id%8001:*)
    ```

    The 2nd, 3rd, and 4th options above support debugging a batch ISPF program.

© Copyright IBM Corp. 1992, 2023

- Specify a display device via a call to CEETEST.

  The 1st parameter to CEETEST test is a 'command string' where the first command in the string can be one of the following ones:

  – A null command. In this case, z/OS Debugger will use the same display as ISPF is using.

    ```
     ;
    ```

  – A parameter that indicates you want to use full-screen mode using the Terminal Interface Manager (TIM) and the ID you logged on to TIM with.

    ```
    VTAM%USERIBM:*;
    ```

  – A parameter that indicates that you want to use remote debug mode and provides the TCP/IP address of the workstation.

    ```
    TCPIP&9.51.66.92%8001:*;
    ```

  The 2nd and 3rd options above support debugging a batch ISPF program.

Here is an example of using CEETEST in a COBOL program to provide both the TEST runtime options and the display device information.

This declaration in the DATA DIVISION indicates using the same 3270 terminal that ISPF is using.

```
01 COMMAND-STRING.
      05 AA PIC 99 Value 1 USAGE IS COMPUTATIONAL.
      05 BB PIC x(60) Value ';'.
```

This declaration in the DATA DIVISION indicates using full-screen mode using the Terminal Interface Manager.

```
01 COMMAND-STRING.
      05 AA PIC 99 Value 14 USAGE IS COMPUTATIONAL.
      05 BB PIC x(60) Value 'VTAM%USERIBM:*;'.
```

This declaration in the DATA DIVISION indicates using remote debug mode.

```
 01 COMMAND-STRING.
      05 AA PIC 99 Value 24 USAGE IS COMPUTATIONAL.
      05 BB PIC x(60) Value 'TCPIP&9.51.66.92%8001:*;'.
```

The 2nd and 3rd options above are needed if you are debugging a batch ISPF program.

These are the declarations needed in the DATA DIVISION for the 2nd parameter to CEETEST.

```
01 FC.
      02 CONDITION-TOKEN-VALUE.
         COPY CEEIGZCT.
             03 CASE-1-CONDITION-ID.
                04 SEVERITY PIC S9(4) BINARY.
                04 MSG-NO PIC S9(4) BINARY.
             03 CASE-2-CONDITION-ID
                      REDEFINES CASE-1-CONDITION-ID.
                04 CLASS-CODE PIC S9(4) BINARY.
                04 CAUSE-CODE PIC S9(4) BINARY.
             03 CASE-SEV-CTL PIC X.
             03 FACILITY-ID PIC XXX.
      02 I-S-INFO PIC S9(9) BINARY.
```

Here is the call to CEETEST that goes in the PROCEDURE DIVISION.

```
 CALL "CEETEST" USING COMMAND-STRING FC.
```

Related concepts
z/OS Debugger runtime options in *IBM z/OS Debugger Reference and Messages*
“Starting z/OS Debugger with CEETEST” on page 119

Chapter 41. Debugging programs in a production environment

Programs in a production environment have any of the following characteristics:

- The programs are compiled without hooks.
- The programs are compiled with the optimization compiler option, usually the OPT compiler option.
- The programs are compiled with COBOL compilers that support the SEPARATE suboption of the TEST compiler option.

This section helps you determine how much of z/OS Debugger's testing functions you want to continue using after you complete major testing of your application and move into the final tuning phase. Included are discussions of program size and performance considerations; the consequences of removing hooks, the statement table, and the symbol table; and using z/OS Debugger on optimized programs.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Fine-tuning your programs for z/OS Debugger" on page 359
"Debugging without hooks, statement tables, and symbol tables" on page 360
"Debugging optimized COBOL programs" on page 362

Fine-tuning your programs for z/OS Debugger

After initial testing, you might want to consider the following options available to improve performance and reduce size:

- Compile your COBOL programs with optimization compiler options, as described in "Debugging optimized COBOL programs" on page 362. You cannot debug PL/I and C/C++ programs that are optimized.
- Removing the hooks, which can improve the performance of your program.
- Removing the statement and symbol tables, which can reduce the size of your program.

Removing hooks

One option for increasing the performance of your program is to compile with a minimum of hooks or with no hooks.

- For C programs, compiling with the option `TEST(NOLINE,BLOCK,NOPATH)` causes the compiler to insert a minimum number of hooks while still allowing you to perform tasks at block boundaries.
- For COBOL programs, compiling with the following compiler suboptions creates programs that do not have hooks:
  - `TEST(NONE)` for any release of the Enterprise COBOL for z/OS Version 3, or COBOL OS/390 & VM, Version 2, compiler
  - `TEST(NOHOOK)` for Enterprise COBOL for z/OS Version 4
  - TEST for Enterprise COBOL for z/OS Version 5

  Using the Dynamic Debug facility, z/OS Debugger inserts hooks while debugging the program, allowing you to perform almost any debugging task.

Independent studies show that performance degradation is negligible because of hook-overhead for PL/I programs. Also, in the event you need to request an attention interrupt, z/OS Debugger is not able to regain control without compiled-in hooks. In such a case you can request an interrupt three times. After

© Copyright IBM Corp. 1992, 2023

the third time, z/OS Debugger is able to stop program execution and prompt you to enter QUIT or GO. If you enter QUIT, your z/OS Debugger session ends. If you enter GO, control is returned to your application.

Programs compiled with certain suboptions of the TEST compiler option have hooks inserted at compile time. However, if the Dynamic Debug facility is activated (which is the default, unless altered by the DYNDEBUG EQAOPTS command) and the programs are compiled with certain compilers, the compiled-in hooks are replaced with runtime hooks. This replacement is done to improve the performance of z/OS Debugger. Certain path hook functions are limited when you use the Dynamic Debug facility. To enable these functions, enter the SET DYNDEBUG OFF command, which deactivates the Dynamic Debug facility. See *IBM z/OS Debugger Reference and Messages* for a description of these commands.

It is a good idea to examine the benefits of maintaining hooks in light of the performance overhead for that particular program.

Removing statement and symbol tables

If you are concerned about the size of your program, you can remove the symbol table, the statement table, or both, after the initial testing period. For C and PL/I programs, compiling with the option TEST(NOSYM) inhibits the creation of symbol tables.

Before you remove them, however, you should consider their advantages. The statement table allows you to display the execution history with statement numbers rather than offsets, and error messages identify statement numbers that are in error. The symbol table enables you to refer to variables and program control constants by name. Therefore, you need to look at the trade-offs between the size of your program and the benefits of having symbol and statement tables.

For programs that are compiled with the following compilers and with the SEPARATE suboption of the TEST compiler option, the symbol tables are saved in a separate debug file. This arrangement lets you to retain the symbol table information and have a smaller program:

- Enterprise COBOL for z/OS, Version 6 Release 2 and later
- Enterprise COBOL for z/OS, Version 4
- Enterprise COBOL for z/OS and OS/390, Version 3
- COBOL for OS/390 & VM, Version 2 Release 2
- COBOL for OS/390 & VM, Version 2 Release 1, with APAR PQ40298
- Enterprise PL/I for z/OS, Version 3.5 or later

For C and C++ programs compiled with the C/C++ compiler of z/OS, Version 1.6 or later, you can compile with the FORMAT(DWARF) suboption of the DEBUG compiler option to save debug information in a separate debug file. This produces a smaller program.

Programs compiled with the Enterprise COBOL for z/OS Version 5 compiler, Version 6 Release 1 compiler, or Version 6 Release 2 and later compiler with the TEST(NOSEPARATE) compiler option have all of their debug information (including the symbol table) stored in a NOLOAD segment of the program object. This segment is only loaded into memory when you are debugging the program object.

Debugging without hooks, statement tables, and symbol tables

z/OS Debugger can gain control at program initialization by using the PROMPT suboption of the TEST run-time option. Even when you have removed all hooks and the statement and symbol tables from a production program, z/OS Debugger receives control when a condition is raised in your program if you specify ALL or ERROR on the TEST run-time option, or when a __ctest(), CEETEST, or PLITEST is executed.

When z/OS Debugger receives control in this limited environment, it does not know what statement is in error (no statement table), nor can it locate variables (no symbol table). Thus, you must use addresses and interpret hexadecimal data values to examine variables. In this limited environment, you can:

- Determine the block that is in control:

```
list (%LOAD, %CU, %BLOCK);
or
list (%LOAD, %PROGRAM, %BLOCK);
```

- Determine the address of the error and of the compile unit:

```
list (%ADDRESS, %EPA);  (where %EPA is allowed)
```

- Display areas of the program in hexadecimal format. Using your listing, you can find the address of a variable and display the contents of that variable. For example, you can display the contents at address 20058 in a C and C++ program by entering:

```
LIST STORAGE (0x20058);
```

  To display the contents at address 20058 in a COBOL or PL/I program, you would enter:

```
LIST STORAGE (X'20058');
```

- Display registers:

```
LIST REGISTERS;
```

- Display program characteristics:

```
DESCRIBE CU;  (for C)

DESCRIBE PROGRAM;  (for COBOL)
```

- Display the dynamic block chain:

```
LIST CALLS;
```

- Request assistance from your operating system:

```
SYSTEM ...;
```

- Continue your program processing:

```
GO;
```

- End your program processing:

```
QUIT;
```

If your program does not contain a statement or symbol table, you can use session variables to make the task of examining values of variables easier.

Even in this limited environment, HLL library routines are still available.

Programs that are compiled with the following combination of compilers and compiler options can have the best performance and smallest module size, while retaining full debugging capabilities:

- Enterprise COBOL for z/OS Version 5 and Version 6, with the TEST compiler option.

  Note: For Version 5, Version 6 Release 1, and Version 6 Release 2 and later with the TEST(NOSEPARATE) compiler option, the debug information in this case is kept in a NOLOAD segment in the program object that is only loaded when the debugger is active.
- Enterprise COBOL for z/OS Version 4, with the TEST(NOHOOK,SEPARATE) compiler option.
- Enterprise COBOL for z/OS and OS/390 Version 3, with the TEST(NONE,SYM,SEPARATE) compiler option.
- COBOL for OS/390 & VM Version 2, with the TEST(NONE,SYM,SEPARATE) compiler option.
- Enterprise PL/I for z/OS Version 3.5 or later, with the TEST(ALL,SYM,NOHOOK,SEPARATE) compiler option.

Debugging optimized COBOL programs

Before you debug an optimized COBOL program, you must compile it with the correct compiler options. See “Choosing TEST or NOTEST compiler suboptions for COBOL programs” on page 27.

The following list describes the tasks that you can do when you debug optimized COBOL programs:

- You can set breakpoints. If the optimizer moves or removes a statement, you cannot set a breakpoint at that statement.
- You can display the value of a variable by using the LIST or LIST TITLED commands. z/OS Debugger displays the correct value of the variable.
- You can step through programs one statement at a time, or run your program until you encounter a breakpoint.
- You can use the SET AUTOMONITOR and PLAYBACK commands.
- You can modify variables in an optimized program that was compiled with one the following compilers:
  - Enterprise COBOL for z/OS, Version 4 and 5
  - Enterprise COBOL for z/OS and OS/390, Version 3 Release 2 or later
  - Enterprise COBOL for z/OS and OS/390, Version 3 Release 1 with APAR PQ63235 installed
  - COBOL for OS/390 & VM, Version 2 Release 2
  - COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ63234 installed

  However, results might be unpredictable. To obtain more predictable results, compile your program with Enterprise COBOL for z/OS, Version 4, and specify the EJPD suboption of the TEST compiler option. However, variables that are declared with the VALUE clause to initialize them cannot be modified.
- If you are using Enterprise COBOL for z/OS, Version 4 and 5, and specify the EJPD suboption of the TEST compiler option, the JUMPTO and GOTO commands are fully enabled by the compiler for use in a debugging session.
- If you are using Enterprise COBOL for z/OS Version 4 and using OPT and the NOHOOK or NONE, and NOEJPD suboptions of the TEST compiler option, the GOTO and JUMPTO commands are not enabled by the compiler. In this case, there is limited support for GOTO and JUMPTO when you run the commands with SET WARNING OFF. However, the results of using GOTO or JUMPTO in this case might be unpredictable and any problems encountered are not investigated by IBM service.
- If you are using Enterprise COBOL for z/OS Version 5 or later, and the programs are complied with OPT and NOEJPD of the TEST compiler option or optimized by Automatic Binary Optimizer for z/OS, the GOTO and JUMPTO are still allowed but you need to first execute the SET WARNING OFF command. However, the results of using GOTO or JUMPTO in this case might be unpredictable and any problems encountered are not investigated by IBM service.
- If you are using Enterprise COBOL for z/OS Version 4 or earlier and using OPT, the information about section and paragraph names is not predictable. Commands LIST NAMES LABELS and AT LABEL are affected, as well as the ability of the remote debugger to list the section and paragraph names when the module is expanded in the **Modules** view.

The enhancements to the compilers help you create programs that can be debugged in the same way that you debug programs that are not optimized, with the following exceptions

- You cannot change the flow of your program.
- You cannot use the AT CALL *entry_name* command. Instead, use the AT CALL * command.
- If the optimizer discarded a variable, you can refer to the variable only by using the DESCRIBE ATTRIBUTES command. If you try to use any other command, z/OS Debugger displays a message indicating that the variable was discarded by the optimization techniques of the compiler.
- If you use the AT command, the following restrictions apply:
  - You cannot specify a line number where all the statements have been removed.
  - You cannot specify a range of line numbers where all the statements have been removed.

– You cannot specify a range of line numbers where the beginning point or ending point specifies a line number where all the statements have been removed.

The Source window does display the variables and statements that the optimizer removed, but you cannot use any z/OS Debugger commands on those variables or statements. For example, you cannot list the value of a variable removed by the optimizer.

Chapter 42. Debugging UNIX System Services programs

You must debug your UNIX System Services programs in one of the following debugging modes:

- remote debug mode
- full-screen mode using the Terminal Interface Manager

If your program spans more than one process, you must debug it in remote debug mode.

If one or more of the programs you are debugging are in a shared library and you are using dynamic debugging, you need to assign the environment variable `_BPX_PTRACE_ATTACH` a value of YES. This enables z/OS Debugger to set hooks in the shared libraries. Programs that have a `.so` suffix are programs in a shared library. For more information about how to set environment variables, see your UNIX System Services documentation.

Debugging MVS POSIX programs

You can debug MVS POSIX programs, including the following types of programs:

- Programs that store source in HFS or zFS
- Programs that use POSIX multithreading
- Programs that use fork/exec
- Programs that use asynchronous signals that are handled by the Language Environment condition handler

To debug MVS POSIX programs in full screen mode or batch mode, the program must run under TSO or MVS batch. If you want to run your program under the UNIX SHELL, you must debug in full-screen mode using the Terminal Interface Manager or remote debug mode.

To debug any MVS POSIX program that spans more than one process, you must debug the program in remote debug mode. To customize the behavior of z/OS Debugger when a new process is created by fork or exec, use the EQAOPTS `MULTIPROCESS` command. For more information about EQAOPTS, see *IBM z/OS Debugger Reference and Messages*.

© Copyright IBM Corp. 1992, 2023

Chapter 43. Debugging non-Language Environment programs

There are several considerations that you must make when you debug programs that do not run under the Language Environment. Some of these are unique to programs that contain no Language Environment routines, others pertain only when the initial program does not execute under control of the Language Environment, and still others apply to all programs that have mixtures of non-Language Environment and Language Environment programs.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
Chapter 14, “Starting z/OS Debugger from the IBM z/OS Debugger Utilities,” on page 115

Debugging exclusively non-Language Environment programs

When Language Environment is not active, you can debug only assembler, disassembly, or non-Language Environment COBOL programs. Debugging programs written in other languages requires the presence of an active Language Environment.

Debugging MVS batch or TSO non-Language Environment initial programs

If the initial program that is invoked does not run under Language Environment, and you want to begin debugging before Language Environment is initialized, you must use the EQANMDBG program to start both z/OS Debugger and your user program.

You do not have to use EQANMDBG to initiate a z/OS Debugger session if the initial user program runs under control of the Language Environment, even if other parts of the program do not run under the Language Environment.

When you use EQANMDBG to debug an assembler program that creates a COBOL reusable runtime environment, z/OS Debugger is not able to debug any COBOL programs. You can create a COBOL reusable runtime environment in one of the following ways:

- Calling the preinitialization routine ILBOSTP0
- Calling the preinitialization routine IGZERRE
- Specifying the runtime option RTEREUS.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
Chapter 17, “Starting z/OS Debugger for batch or TSO programs,” on page 131
z/OS Language Environment Debugging Guide

Debugging CICS non-Language Environment assembler or non-Language Environment COBOL initial programs

The non-Language Environment assembler or non-Language Environment COBOL program that you specify in a DTCN or CADP profile that starts a debugging session must be one of the following:

- The first program started for the CICS transaction.
- The first program that runs for an `EXEC CICS LINK` or `XCTL` statement.

© Copyright IBM Corp. 1992, 2023

Part 7. Debugging complex applications

© Copyright IBM Corp. 1992, 2023

Chapter 44. Debugging multilanguage applications

To support multiple high-level programming languages (HLL), z/OS Debugger adapts its commands to the HLLs, provides *interpretive subsets* of commands from the various HLLs, and maps common attributes of data types across the languages. It evaluates HLL expressions and handles constants and variables.

The topics below describe how z/OS Debugger makes it possible for you to debug programs consisting of different languages, structures, conventions, variables, and methods of evaluating expressions.

A general rule to remember is that z/OS Debugger tries to let the language itself guide how z/OS Debugger works with it.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Qualifying variables and changing the point of view" on page 373
"Debugging multilanguage applications" on page 376
"Handling conditions and exceptions in z/OS Debugger" on page 375

Related references

"z/OS Debugger evaluation of HLL expressions" on page 371
"z/OS Debugger interpretation of HLL variables and constants" on page 371
"z/OS Debugger commands that resemble HLL commands" on page 372
"Coexistence with other debuggers" on page 379
"Coexistence with unsupported HLL modules" on page 379

z/OS Debugger evaluation of HLL expressions

When you enter an expression, z/OS Debugger records the programming language in effect at that time. When the expression is run, z/OS Debugger passes it to the language run time in effect when you entered the expression. This run time might be different from the one in effect when the expression is run.

When you enter an expression that will not be run immediately, you should fully qualify all program variables. Qualifying the variables assures that proper context information (such as load module and block) is passed to the language run time when the expression is run. Otherwise, the context might not be the one you intended when you set the breakpoint, and the language run time might not evaluate the expression.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"z/OS Debugger evaluation of C and C++ expressions" on page 300
"z/OS Debugger evaluation of COBOL expressions" on page 273
"z/OS Debugger evaluation of PL/I expressions" on page 289

z/OS Debugger interpretation of HLL variables and constants

z/OS Debugger supports the use of HLL variables and constants, both as a part of evaluating portions of your test program and in declaring and using session variables.

Three general types of variables supported by z/OS Debugger are:

- Program variables defined by the HLL compiler's symbol table
- z/OS Debugger variables denoted by the percent (**%**) sign
- Session variables declared for a given z/OS Debugger session and existing only for the session

© Copyright IBM Corp. 1992, 2023

HLL variables

Some variable references require language-specific evaluation, such as pointer referencing or subscript evaluation. Once again, the z/OS Debugger interprets each case in the manner of the HLL in question. Below is a list of some of the areas where z/OS Debugger accepts a different form of reference depending on the current programming language:

- Structure qualification

  C and C++ and PL/I: dot (.) qualification, high-level to low-level
  COBOL: `IN` or `OF` keyword, low-level to high-level

- Subscripting

  C and C++: `name [subscript1][subscript2]...`
  COBOL and PL/I: `name(subscript1,subscript2,...)`

- Reference modification

  COBOL `name(left-most-character-position: length)`

HLL constants

You can use both string constants and numeric constants. z/OS Debugger accepts both types of constants in C and C++, COBOL, and PL/I.

z/OS Debugger commands that resemble HLL commands

To allow you to use familiar commands while in a debug session, z/OS Debugger provides an *interpretive subset* of commands for each language. This consists of commands that have the same syntax, whether used with z/OS Debugger or when writing application programs. You use these commands in z/OS Debugger as though you were coding in the original language.

Use the `SET PROGRAMMING LANGUAGE` command to set the current programming language to the desired language. The current programming language determines how commands are parsed. If you `SET PROGRAMMING LANGUAGE` to `AUTOMATIC`, every time the current qualification changes to a module in a different language, the current programming language is automatically updated.

The following types of z/OS Debugger commands have the same syntax (or a subset of it) as the corresponding statements (if defined) in each supported programming language:

Assignment
These commands allow you to assign a value to a variable or reference.

Conditional
These commands evaluate an expression and control the flow of execution of z/OS Debugger commands according to the resulting value.

Declarations
These commands allow you to declare session variables.

Looping
These commands allow you to program an iterative or logical loop as a z/OS Debugger command.

Multiway
These commands allow you to program multiway logic in the z/OS Debugger command language.

In addition, z/OS Debugger supports special kinds of commands for some languages.

Related references
“z/OS Debugger commands that resemble C and C++ commands” on page 293
“z/OS Debugger commands that resemble COBOL statements” on page 267

Qualifying variables and changing the point of view

Each HLL defines a concept of name scoping to allow you, within a single compile unit, to know what data is referenced when a name is used (for example, if you use the same variable name in two different procedures). Similarly, z/OS Debugger defines the concepts of qualifiers and point of view for the run-time environment to allow you to reference all variables in a program, no matter how many subroutines it contains. The assignment x = 5 does not appear difficult for z/OS Debugger to process. However, if you declare x in more than one subroutine, the situation is no longer obvious. If x is not in the currently executing compile unit, you need a way to tell z/OS Debugger how to determine the proper x.

You also need a way to change the z/OS Debugger's point of view to allow it to reference variables it cannot currently see (that is, variables that are not within the scope of the currently executing block or compile unit, depending upon the HLL's concept of name scoping).

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks
“Qualifying variables” on page 373
“Changing the point of view” on page 374

Qualifying variables

Qualification is a method you can use to specify to what procedure or load module a particular variable belongs. You do this by prefacing the variable with the block, compile unit, and load module (or as many of these labels as are necessary), separating each label with a colon (or double colon following the load module specification) and a greater-than sign (:>), as follows:

```
load_name::>cu_name:>block_name:>object
```

This procedure, known as *explicit qualification*, lets z/OS Debugger know precisely where the variable is.

If required, *load_name* is the load module name. It is required only when the program consists of multiple load modules and when you want to change the qualification to other than the current load module. *load_name* can be the z/OS Debugger variable %LOAD.

If required, *cu_name* is the compile unit name. The *cu_name* is required only when you want to change the qualification to other than the currently qualified compile unit. *cu_name* can be the z/OS Debugger variable %CU.

If required, *block_name* is the program block name. The *block_name* is required only when you want to change the qualification to other than the currently qualified block. *block_name* can be the z/OS Debugger variable %BLOCK.

For PL/I only:

- In PL/I, the primary entry name of the external procedure is the same as the compile unit name. When qualifying to the external procedure, the procedure name of the top procedure in a compile unit fully qualifies the block. Specifying both the compile unit and block name results in an error. For example:

  ```
  LM::>PROC1:>variable
  ```

  is valid.

  ```
  LM::>PROC1:>PROC1:>variable
  ```

  is not valid.

For C++ only:

- You must specify the full function qualification including formal parameters where they exist. For example:

1. For function (or block) ICCD2263() declared as void ICCD2263(void) within CU "USERID.SOURCE.LISTING(ICCD226)" the correct block specification for C++ would include the parenthesis () as follows:

   ```
   qualify block %load::>"USERID.SOURCE.LISTING(ICCD226)":>ICCD2263()
   ```

2. For CU ICCD0320() declared as int ICCD0320(signed long int SVAR1, signed long int SVAR2) the correct qualification for AT ENTRY is:

   ```
   AT ENTRY "USERID.SOURCE.LISTING(ICCD0320)":>ICCD0320(long,long)
   ```

   Use the z/OS Debugger command DESCRIBE CUS to give you the correct BLOCK or CU qualification needed.

   Use the LIST NAMES command to show all polymorphic functions of a given name. For the example above, LIST NAMES "ICCD0320*" would list all polymorphic functions called ICCD0320.

You do not have to preface variables in the currently executing compile unit. These are already known to z/OS Debugger; in other words, they are *implicitly* qualified.

In order for attempts at qualifying a variable to work, each block must have a name. Blocks that have not received a name are named by z/OS Debugger, using the form: %BLOCK*nnn*, where *nnn* is a number that relates to the position of the block in the program. To find out the name of z/OS Debugger for the current block, use the DESCRIBE PROGRAMS command.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
“Qualifying variables and changing the point of view in C and C++” on page 307
“Qualifying variables and changing the point of view in COBOL” on page 275

Changing the point of view

The point of view is usually the currently executing block. You can get to inaccessible data by changing the point of view using the SET QUALIFY command with the following operand.

```
load_name::>cu_name:>block_name
```

Each time you update any of the three z/OS Debugger variables %CU, %PROGRAM, or %BLOCK, all four variables (%CU, %PROGRAM, %LOAD, and %BLOCK) are automatically updated to reflect the new point of view. If you change %LOAD using SET QUALIFY LOAD, only %LOAD is updated to the new point of view. The other three z/OS Debugger variables remain unchanged. For example, suppose your program is currently suspended at loadx::>cux:>blockx. Also, the load module loadz, containing the compile unit cuz and the block blockz, is known to z/OS Debugger. The settings currently in effect are:

```
%LOAD = loadx
%CU = cux
%PROGRAM = cux
%BLOCK = blockx
```

If you enter any of the following commands:

```
SET QUALIFY BLOCK blockz;

SET QUALIFY BLOCK cuz:>blockz;

SET QUALIFY BLOCK loadz::>cuz:>blockz;
```

the following settings are in effect:

```
%LOAD = loadz
%CU = cuz
%PROGRAM = cuz
%BLOCK = blockz
```

If you are debugging a program that has multiple enclaves, SET QUALIFY can be used to identify references and statement numbers in any enclave by resetting the point of view to a new block, compile unit, or load module.

Related tasks

Chapter 46, “Debugging across multiple processes and enclaves,” on page 383
“Changing the point of view in C and C++” on page 308
“Changing the point of view in COBOL” on page 276

Handling conditions and exceptions in z/OS Debugger

To suspend program execution just before your application would terminate abnormally, start your application with the following runtime options:

```
TRAP(ON)
TEST(ALL,*,NOPROMPT,*)
```

When a condition is signaled in your application, z/OS Debugger prompts you and you can then *dynamically* code around the problem. For example, you can initialize a pointer, allocate memory, or change the course of the program with the GOTO command. You can also indicate to Language Environment's condition handler, that you have handled the condition by issuing a GO BYPASS command. Be aware that some of the code that follows the instruction that raised the condition might rely on data that was not properly stored or handled.

When debugging with z/OS Debugger, you can (depending on your host system) either instruct the debugger to handle program exceptions and conditions, or pass them on to your own exception handler. Programs also have access to Language Environment services to deal with program exceptions and conditions.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Handling conditions in z/OS Debugger” on page 375
“Handling exceptions within expressions (C and C++ and PL/I only)” on page 376

Handling conditions in z/OS Debugger

You can use either or both of the two methods during a debugging session to ensure that z/OS Debugger gains control at the occurrence of HLL conditions.

If you specify TEST(ALL) as a run-time option when you begin your debug session, z/OS Debugger gains control at the occurrence of most conditions.

Note: z/OS Debugger recognizes all Language Environment conditions that are detected by the Language Environment error handling facility.

You can also direct z/OS Debugger to respond to the occurrence of conditions by using the AT OCCURRENCE command to define breakpoints. These breakpoints halt processing of your program when a condition is raised, after which z/OS Debugger is given control. It then processes the commands you specified when you defined the breakpoints.

There are several ways a condition can occur, and several ways it can be handled.

When a condition can occur

A condition can occur during your z/OS Debugger session when:

- A C++ application throws an exception.
- A C and C++ application program executes a `raise` statement.
- A PL/I application program executes a SIGNAL statement.
- The z/OS Debugger command TRIGGER is executed.

- Program execution causes a condition to exist. In this case, conditions are not raised at consistency points (the operations causing them can consist of several machine instructions, and consistency points usually occur at the beginnings and ends of statements).
- The setting of WARNING is OFF (for C and C++ and PL/I).

When a condition occurs

When an HLL condition occurs and you have defined a breakpoint with associated actions, those actions are first performed. What happens next depends on how the actions end.

- Your program's execution can be terminated with a QUIT command. If you are debugging a CICS non-Language Environment assembler or non-Language Environment COBOL programs, QUIT ends z/OS Debugger and the task ends with an ABEND 4038.
- Control of your program's execution can be returned to the HLL exception handler, using the GO command, so that processing proceeds as if z/OS Debugger had never been invoked (even if you have perhaps used it to change some variable values, or taken some other action).
- Control of your program's execution can be returned to the program itself, using the GO BYPASS command, bypassing any further processing of this exception either by the user program or the environment.
- PL/I allows GO TO out of block;, so execution control can be passed to some other point in the program.
- If no circumstances exist explicitly directing the assignment of control, your primary commands file or terminal is queried for another command.

If, after the execution of any defined breakpoint, control returns to your program with a GO, the condition is raised again in the program (if possible and still applicable). If you use a GOTO to bypass the failing statement, you also bypass your program's error handling facilities.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

“Language Environment conditions and their C and C++ equivalents” on page 300
“PL/I conditions and condition handling” on page 285
z/OS Language Environment Programming Guide
Enterprise COBOL for z/OS Language Reference

Handling exceptions within expressions (C and C++ and PL/I only)

When an exception such as division by zero is detected in a z/OS Debugger expression, you can use the z/OS Debugger command SET WARNING to control z/OS Debugger and program response. During an interactive z/OS Debugger session, such exceptions are sometimes due to typing errors and so are probably not intended to be passed to the program. If you do not want errors in z/OS Debugger expressions to be passed to your program, use SET WARNING ON. Expressions containing such errors are terminated, and a warning message is displayed.

However, you might want to pass an exception to your program, perhaps to test an error recovery procedure. In this case, use SET WARNING OFF.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

“Using SET WARNING PL/I command with built-in functions” on page 290

Debugging multilanguage applications

Language Environment simplifies the debugging of multilanguage applications by providing a single run-time environment and interlanguage communication (ILC).

When the need to debug a multilanguage application arises, you can find yourself facing one of the following scenarios:

- You need to debug an application written in more than one language, where each language is supported by Language Environment and can be debugged by z/OS Debugger.
- You need to debug an application written in more than one language, where not all of the languages are supported by Language Environment, nor can they be debugged by z/OS Debugger.

When writing a multilanguage application, a number of special considerations arise because you must work outside the scope of any single language. The Language Environment initialization process establishes an environment tailored to the set of HLLs constituting the main load module of your application program. This removes the need to make explicit calls to manipulate the environment. Also, termination of the Language Environment environment is accomplished in an orderly fashion, regardless of the mixture of HLLs present in the application.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Debugging an application fully supported by Language Environment" on page 377
"Using session variables across different programming languages" on page 377

Debugging an application fully supported by Language Environment

If you are debugging a program written in a combination of languages supported by Language Environment and compiled by supported compilers, very little is required in the way of special actions. z/OS Debugger normally recognizes a change in programming languages and automatically switches to the correct language when a breakpoint is reached. If desired, you can use the `SET PROGRAMMING LANGUAGE` command to stay in the language you specify; however, you can only access variables defined in the currently set programming language.

When defining session variables you want to access from compile units of different languages, you must define them with compatible attributes.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Using session variables across different programming languages" on page 377

Related references

z/OS Language Environment Programming Guide

Using session variables across different programming languages

While working in one language, you can declare session variables that you can continue to use after calling in a load module of a different language. The table below shows how the attributes of session variables are mapped across programming languages. Session variables with attributes not shown in the table cannot be accessed from other programming languages. (Some attributes supported for C and C++ or PL/I session variables cannot be mapped to other languages; session variables defined with these attributes cannot be accessed outside the defining language. However, all of the supported attributes for COBOL session variables can be mapped to equivalent supported attributes in C and C++ and PL/I, so any session variable that you declare with COBOL can be accessed from C and C++ and PL/I.)

Machine attributes	PL/I attributes	C and C++ attributes	COBOL attributes	Assembler, disassembly, and LangX COBOL attributes
byte	`CHAR(1)`	`unsigned char`	`PICTURE X`	`DS X` or `DS C`

Machine attributes	PL/I attributes	C and C++ attributes	COBOL attributes	Assembler, disassembly, and LangX COBOL attributes
byte string	`CHAR(j)`	`unsigned char[j]`	`PICTURE X(j)`	`DS XLj` or `DS CLj`
halfword	`FIXED BIN(15,0)`	`signed short int`	`PICTURE S9(j≤4) USAGE BINARY`	`DS H`
fullword	`FIXED BIN(31,0)`	`signed long int`	`PICTURE S9(4<j≤9) USAGE BINARY`	`DS F`
floating point	`FLOAT BIN(21)` or `FLOAT DEC(6)`	`float`	`USAGE COMP-1`	`DS E`
long floating point	`FLOAT BIN(53)` or `FLOAT DEC(16)`	`double`	`USAGE COMP-2`	`DS D`
extended floating point	`FLOAT BIN(109)` or `FLOAT DEC(33)`	`long double`	n/a	`DS L`
fullword pointer	`POINTER`	*	`USAGE POINTER`	`DS A`

Note: When registering session variables in PL/I, the `DECIMAL` type is always the default. For example, if C declares a `float`, PL/I registers the variable as a `FLOAT DEC(6)` rather than a `FLOAT BIN(21)`.

When declaring session variables, remember that C and C++ variable names are case-sensitive. When the current programming language is C and C++, only session variables that are declared with uppercase names can be shared with COBOL or PL/I. When the current programming language is COBOL or PL/I, session variable names in mixed or lowercase are mapped to uppercase. These COBOL or PL/I session variables can be declared or referenced using any mixture of lowercase and uppercase characters and it makes no difference. However, if the session variable is shared with C and C++, within C and C++, it can only be referred to with all uppercase characters (since a variable name composed of the same characters, but with one or more characters in lowercase, is a different variable name in C and C++).

Session variables with incompatible attributes cannot be shared between other programming languages, but they do cause session variables with the same names to be deleted. For example, COBOL has no equivalent to PL/I's `FLOAT DEC(33)` or C's `long double`. With the current programming language COBOL, if a session variable X is declared `PICTURE S9(4)`, it will exist when the current programming language setting is PL/I with the attributes `FIXED BIN(15,0)` and when the current programming language setting is C with the attributes `signed short int`. If the current programming language setting is changed to PL/I and a session variable X is declared `FLOAT DEC(33)`, the X declared by COBOL will no longer exist. The variable X declared by PL/I will exist when the current programming language setting is C with the attributes `long double`.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

“z/OS Debugger interpretation of HLL variables and constants” on page 371

Creating a commands file that can be used across different programming languages

If you want to create a commands file to use across different programming languages, "Creating a commands file" on page 171 describes some guidelines you should follow to ensure that the commands files works correctly.

Coexistence with other debuggers

Coexistence with other debuggers cannot be guaranteed because there can be situations where multiple debuggers might contend for use of storage, facilities, and interfaces that are intended for only one requester.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"Coexistence with unsupported HLL modules" on page 379

Coexistence with unsupported HLL modules

Compile units or program units written in unsupported high- or low-level languages, or in older releases of HLLs, are tolerated. See *Using CODE/370 with VS COBOL II and OS PL/I* for information about two unsupported HLLs that can be used with z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"Coexistence with other debuggers" on page 379

Chapter 45. Debugging multithreading programs

You can run your multithreading programs with z/OS Debugger when POSIX pthread_create is used to create new threads under Language Environment. When more than one thread is involved in your program, z/OS Debugger might be started by any or all of them. Because conflicting use of the terminal or log file, for example, could occur if z/OS Debugger is operating on multiple threads, its use is single-threaded. So, if your program runs as two threads (thread A and thread B) and thread A calls z/OS Debugger, z/OS Debugger accepts the request and begins operating on behalf of thread A. If, during that period, thread B calls z/OS Debugger, the request from thread B is held until the request from thread A is complete (for example, you issued a STEP or GO command). z/OS Debugger is then released and can accept any pending invocation.

Restrictions when debugging multithreading applications

- Debugging applications that create another thread is constrained because both threads compete for the use of the terminal.
- Only the variables and symbol information for compile units in the thread that is being debugged are accessible.
- The `LIST CALL` command provides a traceback of the compile units only in the current thread.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
z/OS Language Environment Programming Guide

© Copyright IBM Corp. 1992, 2023

Chapter 46. Debugging across multiple processes and enclaves

There is a single z/OS Debugger session across all enclaves in a process. Breakpoints set in one process are restored when the new process begins in the new session.

In full-screen mode or batch mode, you can debug a non-POSIX program that spans more than one process, but z/OS Debugger can be active in only one process. In remote debug mode, you can debug a POSIX program that spans more than one process. The remote debugger can display each process.

When you are recording the statements that you run, data collection persists across multiple enclaves until you stop recording. When you replay your statements, the data is replayed across the enclave boundaries in the same order as they were recorded.

A commands file continues to execute its series of commands regardless of what level of enclave is entered.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Starting z/OS Debugger within an enclave" on page 383
"Viewing z/OS Debugger windows across multiple enclaves" on page 383
"Ending a z/OS Debugger session within multiple enclaves" on page 383
"Using z/OS Debugger commands within multiple enclaves" on page 384

Starting z/OS Debugger within an enclave

After an enclave in a process activates z/OS Debugger, it remains active throughout subsequent enclaves in the process, regardless of whether the run-time options for the enclave specify TEST or NOTEST. z/OS Debugger retains the settings specified from the TEST run-time option for the enclave that activated it, until you modify them with SET TEST. If your z/OS Debugger session includes more than one process, the settings for TEST are reset according to those specified on the TEST run-time option of the first enclave that activates z/OS Debugger in each new process.

If z/OS Debugger is first activated in a nested enclave of a process, and you step or go back to the parent enclave, you can debug the parent enclave. However, if the parent enclave contains COBOL but the nested enclave does not, z/OS Debugger is not active for the parent enclave, even upon return from the child enclave.

Upon activation of z/OS Debugger, the initial commands string, primary commands file, and the preferences file are run. They run only once, and affect the entire z/OS Debugger session. A new primary commands file cannot be started for a new enclave.

Viewing z/OS Debugger windows across multiple enclaves

When an enclave starts another enclave, all compile units in the first enclave are hidden. You can change the point of view to a new compile unit (by using the SET QUALIFY command) only if that compile unit is in the current enclave.

Ending a z/OS Debugger session within multiple enclaves

If you specify the NOPROMPT suboption of the TEST runtime option for the next process on the host, z/OS Debugger restores the saved breakpoints after it gains control of that next process. However, z/OS Debugger might gain control of the process after many statements have been run. Therefore, z/OS Debugger might not run some or all of the following breakpoints:

- STATEMENT/LINE

© Copyright IBM Corp. 1992, 2023

- ENTRY
- EXIT
- LABEL

If you have not used these breakpoint types, you can specify NOPROMPT.

In a single enclave, QUIT closes z/OS Debugger. For CICS non-Language Environment programs (assembler or non-Language Environment COBOL), QUIT closes z/OS Debugger and the task ends with an ABEND 4038, regardless of the link level.

In a nested enclave, however, QUIT causes z/OS Debugger to signal a severity 3 condition that corresponds to Language Environment message CEE2529S. The system is trying to cleanly terminate all enclaves in the process. This applies to any Language Environment programs on CICS, IMS and MVS.

Normally, the condition causes the current enclave to terminate. Then, the same condition will be raised in the parent enclave, which will also terminate. This sequence continues until all enclaves in the process have been terminated. As a result, you will see a CEE2529S message for each enclave that is terminated.

For CICS and MVS only: Depending on Language Environment run-time settings, the application might be terminated with an ABEND 4038. This termination is normal and should be expected.

Using z/OS Debugger commands within multiple enclaves

Some z/OS Debugger commands and variables have a specific scope for enclaves and processes. The table below summarizes the behavior of specific z/OS Debugger commands and variables when you are debugging an application that consists of multiple enclaves.

z/OS Debugger command	Affects current enclave only	Affects entire z/OS Debugger session	Comments
%CAADDRESS	X		
AT GLOBAL		X	
AT TERMINATION		X	
CC START		X	
CC STOP		X	
CLEAR AT	X	X	In addition to clearing breakpoints set in the current enclave, CLEAR AT can clear global breakpoints.
CLEAR DECLARE		X	
CLEAR LDD		X	
CLEAR VARIABLES		X	
Declarations		X	Session variables are cleared at the termination of the process in which they were declared.
DISABLE	X	X	In addition to disabling breakpoints set in the current enclave, DISABLE can disable global breakpoints.
ENABLE	X	X	In addition to enabling breakpoints set in the current enclave, ENABLE can enable global breakpoints.

z/OS Debugger command	Affects current enclave only	Affects entire z/OS Debugger session	Comments
LIST AT	X	X	In addition to listing breakpoints set in the current enclave, LIST AT can list global breakpoints.
LIST CALLS	X		Applies to all systems except MVS batch and MVS with TSO. Under MVS batch and MVS with TSO, LIST CALLS lists the call chain for the current active thread in the current active enclave. For programs containing interlanguage communication (ILC), routines from previous enclaves are only listed if they are coded in a language that is active in the current enclave. **Note:** Only compile units in the current thread will be listed for PL/I multitasking applications.
LIST CC		X	Only source statements for the current enclave will be displayed.
LIST EXPRESSION	X		You can only list variables in the currently active thread.
LIST LAST		X	
LIST LDD		X	
LIST NAMES CUS		X	Applies to compile unit names. In the Debug Frame window, compile units in parent enclaves are marked as deactivated.
LIST NAMES LABELS	X		You can only list variables in the currently active thread.
LIST NAMES TEST		X	Applies to z/OS Debugger session variable names.
MONITOR GLOBAL		X	Applies to Global monitors.
PLAYBACK ENABLE		X	The PLAYBACK command that informs z/OS Debugger to begin the recording session.
PLAYBACK DISABLE		X	The PLAYBACK command that informs z/OS Debugger to stop the recording session.
PLAYBACK START		X	The PLAYBACK command that suspends execution of the program and indicates to z/OS Debugger to enter replay mode.
PLAYBACK STOP		X	The PLAYBACK command that terminates replay mode and resumes normal execution of z/OS Debugger.
PLAYBACK BACKWARD		X	The PLAYBACK command that indicates to z/OS Debugger to perform STEP and RUNTO commands backward, starting from the current point and going to previous points.

z/OS Debugger command	Affects current enclave only	Affects entire z/OS Debugger session	Comments
PLAYBACK FORWARD		X	The PLAYBACK command that indicates to z/OS Debugger to perform STEP and RUNTO commands forward, starting from the current point and going to the next point.
PROCEDURE		X	
SET AUTOMONITOR[1]	X		Controls the monitoring of data items at the currently executing statement.
SET COUNTRY[1]	X		This setting affects both your application and z/OS Debugger. At the beginning of an enclave, the settings are those provided by Language Environment or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.
SET EQUATE[1]		X	
SET INTERCEPT[1]		X	For C, intercepted streams or files cannot be part of any C I/O redirection during the execution of a nested enclave. For example, if `stdout` is intercepted in program A, program A cannot then redirect `stdout` to `stderr` when it does a `system()` call to program B. Also, not supported for PL/I.
SET NATIONAL LANGUAGE[1]	X		This setting affects both your application and z/OS Debugger. At the beginning of an enclave, the settings are those provided by Language Environment or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.
SET PROGRAMMING LANGUAGE[1]	X		Applies only to programming languages in which compile units known in the current enclave are written (a language is "known" the first time it is entered in the application flow).
SET QUALIFY[1]	X		Can only be issued for load modules, compile units, and blocks that are known in the current enclave.
SET TEST[1]		X	
TRIGGER condition[2]	X		Applies to triggered conditions.[2] Conditions can be either an Language Environment symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.

z/OS Debugger command	Affects current enclave only	Affects entire z/OS Debugger session	Comments
TRIGGER AT	X	X	In addition to triggering breakpoints set in the current enclave, TRIGGER AT can trigger global breakpoints.

Note:

1. SET commands other than those listed in this table affect the entire z/OS Debugger session.
2. If no active condition handler exists for the specified condition, the default condition handler can cause the program to end prematurely.

Chapter 47. Debugging a multiple-enclave interlanguage communication (ILC) application

When you debug a multiple-enclave ILC application with z/OS Debugger, use the `SET PROGRAMMING LANGUAGE` to change the current programming language setting. The programming language setting is limited to the languages currently known to z/OS Debugger (that is, languages contained in the current load module).

Command lists on monitors and breakpoints have an implied programming language setting, which is the language that was in effect when the monitor or breakpoint was established. Therefore, if you change the language setting, errors might result when the monitor is refreshed or the breakpoint is triggered.

© Copyright IBM Corp. 1992, 2023

Chapter 48. Debugging programs called by Java native methods

This topic describes how to debug, with z/OS Debugger, Java native methods and the programs they call that are running in Language Environment. By inserting calls to the Language Environment CWI service CEE3CBTS and callable service CEETEST into your Java native method or program and compiling your methods or programs with the H00K suboption of the TEST compiler option, you can debug your application. These instructions describe how to insert the calls to CEE3CBTS and CEETEST directly into your method or program.

These instructions assume you understand the following items:

- You understand Java JNI interface.
- You have configured a remote debugger to debug the Java native method and the programs it calls. You need to know the IP address and port ID of the remote debugger.
- You can modify the compilation parameters of the Java native method and the programs it calls.

Do the following steps:

1. Review the description of the Language Environment CWI service CEE3CBTS in *Language Environment Vendor Interfaces*. For this situation, specify the following values for the elements in the structure:
   - TCP/IP address, as described in the *Language Environment Vendor Interfaces*
   - Debugger port ID, as described in the *Language Environment Vendor Interfaces*
   - Client Process ID, assign a value of 0
   - Client Thread ID, assign a value of 0
   - Client IP address, assign a value of 0
   - Debug Flow, assign a value of 1
2. Choose which programs that the native method calls to debug. Decide where you want to start and stop debugging.
3. In the Java native method, add a call to CEE3CBTS with the *AttachDebug* function code and assign values to the debug context parameters.
4. In the Java native method or the programs it calls, add a call to CEETEST. CEETEST is the way you start z/OS Debugger for this situation.
5. In the Java native method, add a call to CEE3CBTS with the *StopDebug* function code to stop the debug session.
6. Run the JCL for your programs. Your remote debugging session starts.

After you finish debugging your Java native method and the programs called by the Java native method, remove the modifications done in these steps before moving your application to a production environment.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

"Starting z/OS Debugger with CEETEST" on page 119
"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27
"Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 34
"Choosing TEST or DEBUG compiler suboptions for C programs" on page 39
"Choosing TEST or DEBUG compiler suboptions for C++ programs" on page 44

Related references

Language Environment Vendor Interfaces

© Copyright IBM Corp. 1992, 2023

Chapter 49. Solving problems in complex applications

This section describes some problems you might encounter while you try to debug complex applications and some possible solutions.

Debugging programs loaded from library lookaside (LLA)

z/OS Debugger obtains information about programs in memory by using binder APIs. The binder APIs must access information stored in the data set containing the load module or program object. In most cases, z/OS can provide z/OS Debugger the data set name from which the program was loaded so z/OS Debugger can pass it to the binder APIs. However, z/OS does not have this information for programs loaded from LLA.

When z/OS Debugger attempts to debug a program loaded from LLA, z/OS Debugger does the following steps:

- z/OS Debugger checks for the allocation of DD name EQALOAD and checks that it contains a member name that matches the program that LLA loaded.
- If z/OS Debugger does not find a program by the specified name in EQALOAD, z/OS Debugger checks for the allocation of DD name STEPLIB and checks that it contains a member name that matches the program that LLA loaded.
- If z/OS Debugger does find a program by the specified name in one of the previous steps and the length of this program matches the length of the program in memory, z/OS Debugger passes the data set name from the corresponding DD statement to the binder APIs to use it to obtain the information.

The following restrictions apply:

- z/OS Debugger cannot always determine the exact length of the program in memory. In certain situations, the length might be rounded to a multiple of 4K. Therefore, the length checking is not always exact and programs that might appear the same length are not.
- The copy of the program found in DD name EQALOAD or DD name STEPLIB must exactly match the copy in memory. If the program found does not exactly match the copy loaded from LLA (even though the lengths match), unpredictable problems, including abends, might occur.
- If you are running DTSU in foreground, you must use DD name EQALOAD. When a DD name of STEPLIB is specified when DTSU is running in the TSO foreground, DTSU uses a different DD name and, therefore, z/OS Debugger cannot find STEPLIB.

Debugging user programs that use system prefixed names

z/OS Debugger assumes that load module and compile unit names that begin with specific prefixes are system components. For example, EQAxxxxx is a z/OS Debugger module, CEExxxxx is a Language Environment module, and IGZxxxxx is a COBOL module.

z/OS Debugger does not try to debug load modules or compile units that have these prefixes for the following reasons:

- z/OS Debugger might perform improper recursions that might result in abnormally endings (ABENDs) or other erroneous behavior.
- z/OS Debugger assumes users do not have access to the source for these load modules and library routines.
- Creating debug information for these routines would waste significant amounts of memory and other resources.

© Copyright IBM Corp. 1992, 2023

If you have named a user load module or compile unit with a prefix that conflicts with one of these system prefixes, you can use the NAMES INCLUDE command and the instructions described in this section to debug this load module or compile unit.

IMPORTANT: Do **not** use the NAMES INCLUDE command to debug system components (for example, z/OS Debugger, Language Environment, CICS, IMS, or compiler run-time modules). If you attempt to do debug these system components, you might experience unpredictable failures. Only use this command to debug *user* programs that are named with prefixes that z/OS Debugger recognizes as system components.

Displaying system prefixes

You can display a list of prefixes that z/OS Debugger recognizes as system prefixes by using the following commands:

```
NAMES DISPLAY ALL EXCLUDED LOADMODS;
NAMES DISPLAY ALL EXCLUDED CUS;
```

These commands display a list of names currently excluded at your request (by using the NAMES EXCLUDE command), followed by a section displaying a list of names excluded by z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

NAMES command in *IBM z/OS Debugger Reference and Messages*

Debugging programs with names similar to system components

If the name of your program begins with one of the prefixes excluded by z/OS Debugger, use the NAMES INCLUDE command to indicate to z/OS Debugger that your program is a user load module or compile unit, not a system program.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"NAMES command" in *IBM z/OS Debugger Reference and Messages*

Debugging programs containing data-only modules

Some programs contain load modules or compile units that have no executable code. These modules are known as data-only modules. These modules are prevalent in assembler programs. In some situations, z/OS Debugger might not recognize that these modules contain no executable instructions and attempt to set a breakpoint, which means overlaying the contents of these modules.

In these situations, you can use the NAMES EXCLUDE command to indicate to z/OS Debugger that these are data-only modules that contain no executable code. z/OS Debugger will not attempt to set breakpoints in these data-only modules. If the NAMES EXCLUDE command is used to exclude a module that contains executable instructions, the module might still appear in z/OS Debugger and z/OS Debugger might still attempt to set breakpoints in the modules.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"NAMES command" in *IBM z/OS Debugger Reference and Messages*

Optimizing the debugging of large applications

z/OS Debugger is designed to implicitly load the debug data for any compile units compiled with the TEST or DEBUG compiler option. However, some very large applications can contain a large number of load modules or compile units that you do not need to debug. In some cases, the creation and manipulation of debug data for these load modules or compile units can consume a significant amount of memory, CPU time, and other resources.

You can handle this situation in one of the following ways:

- Change the default behavior so that z/OS Debugger loads debug data only for modules that you indicate that you want to debug.
- Indicate to z/OS Debugger that you do not want to debug certain modules.

In some instances, even when the OPT compile option is not used, the compiler can optimize the code:

- By collapsing several statements into one single statement.
- By not creating the object code for a specific statement and instead using a NOOP instruction.
- By removing the duplicate code.

Because the program object signature area does not specify the use of the OPT compile option, the debugger is not aware of such optimization and does not take any measures. To make the behavior more predicable when you debug such applications, use any of the following approaches:

- Compile with TEST(EJPD) to reduce optimization.
- Compile with a combination of OPT(x) and TEST(EJPD/NOEJPD) to make the debugger be aware of the optimization and control the optimization performed by the compiler.

Even when you use the suggested approach, the behavior can still become unpredictable in some instances.

Using explicit debug mode to load debug data for only specific modules

By default, z/OS Debugger automatically loads debug data whenever it encounters a high-level language compile unit compiled with the TEST or DEBUG compiler option. In most cases, this is the most convenient mode of operation because you do not have to decide in advance which load modules and compile units you want to debug. However, in some complex applications, manipulating this data might cause a significant performance impact. In this case, you can use explicit debug mode to load debug data only for compile units that you indicate you want to debug.

You enable explicit debug mode by entering the SET EXPLICITDEBUG ON command or by specifying the EQAOPTS EXPLICITDEBUG command. By default, this mode is OFF. In explicit debug mode, (except for cases described below) you must use the LOADDEBUGDATA (LDD) command to cause z/OS Debugger to load the debug data for the compile units that you want to debug.

In most cases, you can use the SET EXPLICTDEBUG command to enable explicit debug mode; however, in some cases you might need to use the EQAOPTS EXPLICITDEBUG command. Because z/OS Debugger does not process commands until after it processes the initial load module and all the compile units it contains, if you want z/OS Debugger to *not* load debug data for compile units in the initial load module, use the EQAOPTS EXPLICITDEBUG command.

When explicit debug mode is active, z/OS Debugger loads debug data in only the following cases:

- For the compile unit where z/OS Debugger first became active and the first compile unit in each enclave. In most cases, this is the entry compile unit for the initial load module.
- Whenever z/OS Debugger loads a load module and you previously entered a LOADDEBUGDATA (LDD) command for that load module and compile unit or when you enter an LDD command for a compile unit in the current load module.
- Whenever z/OS Debugger processes a load module for any of the following reasons and you previously specified the compile unit on a NAMES INCLUDE CU command:
  - It is the initial load module.
  - When z/OS Debugger loads a load module that you previously specified on an LDD command.
  - When z/OS Debugger loads a load module that you previously specified on a NAMES INCLUDE LOADMOD command.
  - It is a load module for which z/OS Debugger generated an implicit NAMES INCLUDE LOADMOD command.

- For the target of a deferred AT ENTRY which specifies both load module and compile unit names and in which the compile unit name is not a source file name enclosed in quotation marks (").
- For the entry point compile unit of a load module that you specified in an AT LOAD command.
- In CICS, for the load module and compile units you specified in DTCN or the Program and Comp Unit you specified in CADP, unless they contain an asterisk (*).

z/OS Debugger does not support the SET DISASSEMBLY ON command in explicit debug mode. When explicit debug mode is active, z/OS Debugger forces SET DISASSEMBLY OFF and you will not be able to set it back to ON while in explicit debug mode.

Excluding specific load modules and compile units

In some cases, you might know that there are certain load modules or compile units that you do not want to debug. In this case, you can improve performance by informing z/OS Debugger to not load debug data for these load modules or compile units.

You can use the NAMES EXCLUDE command to indicate to z/OS Debugger that it does not need to maintain debug data for these modules. When you use the NAMES EXCLUDE command to exclude executable modules, there are situations where z/OS Debugger requires debug data for the excluded modules. The following list, while not comprehensive, describes some of the possible situations:

- The entry point of an enclave.
- The next higher-level compile unit when a STEP RETURN command is executing.
- Compile units that appear in the call chain of a compile unit where z/OS Debugger has suspended execution.
- The next higher-level compile unit when the user-program has been suspended at an AT EXIT breakpoint.

Also, when you enter a deferred AT ENTRY command, z/OS Debugger generates an implicit NAMES INCLUDE command for the load module and compile unit that is the target of the deferred AT ENTRY. If these names appear later in the program, z/OS Debugger will not exclude them even if you specified them in a previous NAMES EXCLUDE command.

In all of the above situations, z/OS Debugger loads debug data as required and these modules might become known to z/OS Debugger.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
"NAMES command" in *IBM z/OS Debugger Reference and Messages*

Displaying current NAMES settings

Use the NAMES DISPLAY command to display the current settings of the NAMES command.

Refer to the following topics for more information related to the material discussed in this topic.

Related references
"NAMES command" in *IBM z/OS Debugger Reference and Messages*

Using the EQAOPTS NAMES command to include or exclude the initial load module

You cannot use the NAMES command on load modules or compile units that are already known to z/OS Debugger; therefore, you cannot use the NAMES command to indicate to z/OS Debugger that you want to include or exclude the initial load module or the compile units contained in the initial load module. If you want to do this, you must specify the EQAOPTS NAMES command either at run time or through the EQAOPTS load module. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS

commands" in the *IBM z/OS Debugger Reference and Messages* or *IBM z/OS Debugger Customization Guide*.

Refer to the following topics for more information related to the material discussed in this topic.

Related references

"NAMES command" in *IBM z/OS Debugger Reference and Messages*

Using delay debug mode to delay starting of a debug session

By default, z/OS Debugger starts a debug session at the first entry compile unit of the initial load module of an application. However, there are cases where the problem is in some compile unit (for example, `prog1`) inside the application that needs debugging.

Currently, you enter AT ENTRY `prog1` and GO commands when the debug session starts.

However, in some complex applications, it can take some significant time before `prog1` appears. In this case, you can use delay debug mode to delay the starting of the debug session until z/OS Debugger recognizes `prog1`.

z/OS Debugger is in the dormant state during the delay debug mode and monitors only a few events. When z/OS Debugger recognizes `prog1`, z/OS Debugger comes out of delay debug mode, completes the initialization, and starts the debug session.

Delay debug mode uses a delay debug profile data set that contains the program list and TEST runtime option. This profile is used by z/OS Debugger to match the program name or C function name (compile unit) (and optionally a load module name) with the names in the program list. If there is a match, z/OS Debugger comes out of the delay debug mode and uses the TEST runtime to complete the initialization. This data set is a physical sequential data set that is created and edited by using one of the following ways:

- Option B of the IBM z/OS Debugger Utilities: Delay Debug Profile
- The **z/OS Debugger Profiles** view in a remote IDE
- Any application that uses the Debug Profile Service API for profile management

You can enable delay debug mode by using one of the following ways:

- Using the EQAOPTS DLAYDBG command
- Specifying a simple TEST runtime option for Language Environment programs[12]

By default, delay debug mode is NO. When delay debug mode is enabled, you can specify these additional commands:

DLAYDBGCND

You can use this command to indicate whether you want z/OS Debugger to monitor condition events in the delay debug mode.

The default is `DLAYDBGCND,ALL`.

DLAYDBGDSN

Delay debug profile data set naming pattern.

The default is *userid*.DLAYDBG.EQAUOPTS.

This command is ignored if delay debug mode is turned on using a simple TEST runtime option. In that case, the delay debug profile data set naming pattern is set with a parameter to the Debug Profile Service API.

[12] The simple TEST option turns on delay debug if the application environment is not a foreground TSO application, and TEST suboptions are not supplied using #`pragma runopts` or `PLIXOPT`.

DLAYDBGTRC
Delay debug pattern match trace message level.

This message level is used to generate error and informational messages for debugging purposes.

The default is `0`, which indicates no trace messages.

DLAYDBGXRF

You can use this command to indicate that you want z/OS Debugger to use the cross reference file or the Terminal Interface Manager repository to find the user ID when it constructs the delay debug profile data set name.

This command can be used in the IMS environment when an IMS transaction is started with a generic ID. With the RESPOSITORY option, the command can also be used in the DB/2 stored procedures environment when a stored procedure runs under a generic ID.

See “Debugging tasks running under a generic user ID by using Terminal Interface Manager ” on page 399 for a description of the steps required to use the REPOSITORY option of DLAYDBGXRF

Once z/OS Debugger completes the initialization, the delay debug mode cannot be reactivated.

Usage notes

- The delay debug mode applies to non-CICS environments only.
- The delay debug mode applies to programs compiled with the Enterprise COBOL for z/OS and Enterprise PL/I for z/OS compilers, C functions compiled with the z/OS V2.1 XL C/C++ compiler and non-Language Environment programs. Non-Language Environment compile units must be the initial program in a task or the target of a LINK or LINKX macro to be eligible for delay debug pattern matching.

  For compile time and run time requirements of C functions, see “Delay debug mode for C requires the FUNCEVENT(ENTRYCALL) compiler suboption” on page 43.
- The main program of the application must be a Language Environment program, or a non-Language Environment program that is started by using EQANMDBG.
- The TEST runtime option used to start z/OS Debugger at the beginning of the application must have PROMPT in the third suboption, for example, `TEST(ALL,*,PROMPT,*)`.
- If the user exit method is used to start z/OS Debugger at the beginning of the application, the user exit data set should have a '*' as one of the names in the program name list name list, so that the pattern matching always succeeds and the TEST runtime option is returned to Language Environment.

  In addition, the name of the user exit data set must be different from the name of the delay debug profile data set.
- Use Delay Debug Profile to set up the delay debug profile data set. You can find this tool under option B in IBM z/OS Debugger Utilities.
- The TEST runtime option in the delay debug profile must have PROMPT in the third sub option, for example, `TEST(ALL,*,PROMPT,*)`.
- To interoperate between 31-bit and 64-bit COBOL programs, specify load module and compile unit pairs for both 31-bit and 64-bit programs in the debug profile.

Debugging subtasks created by the ATTACH assembler macro

Under certain circumstances, you can debug multi-tasked applications that create their subtasks by using the ATTACH assembler macro. To debug subtasks, the following conditions must be met:

- SVC screening must be in effect. For information about enabling SVC screening for your task, see SVCSCREEN in Chapter 15. EQAOPTS commands in the *IBM z/OS Debugger Customization Guide*.
- Delay debug mode must be active or INSPPREF DD must be allocated.

  - For information about setting delay debug mode, see “Using delay debug mode to delay starting of a debug session ” on page 397.
  - Without delay debug mode, INSPPREF DD can be used to supply commands for a batch debug session on a subtask.
- If the main program of the top-level task is not Language Environment-compliant, you must start the program by using EQANMDBG. For information about using EQANMDBG to start z/OS Debugger for non-Language Environment-compliant programs, see “Starting z/OS Debugger for programs that start outside of Language Environment” on page 134.

To debug a program in the main task or in a subtask of the main task, you must enter pattern matching arguments in the delay debug profile data set that matches one of the programs that executes in the subtask.

Usage notes:

- Each task to be debugged in an address space uses an entirely separate copy of z/OS Debugger. Therefore, commands such as `LIST CALLS` provide information only about the current task. z/OS Debugger does not provide information about any tasks that precede the current task in the parent chain.
- For remote debug users, each task is displayed in the Debug view as a separate "Incoming debug session".
- 3270 users can log on to the Terminal Interface Manager on multiple terminals using the same user ID. For each task to be debugged, a z/OS Debugger session starts on an available terminal if the following conditions are met:
  - The user chose full-screen mode using the Terminal Interface Manager in the delay debug profile.
  - The terminal that the user logged on is available, and is not in a z/OS Debugger session.

Debugging tasks running under a generic user ID by using Terminal Interface Manager

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

If you use Terminal Interface Manager and want to debug an IMS transaction or a DB/2 stored procedure that runs under a generic user ID, take the following setup steps first:

1. The Terminal Interface Manager stared task must be started in repository mode by using the -r startup option. See *Starting the Terminal Interface Manager* in the *IBM z/OS Debugger Customization Guide* for more information about this task.
2. The IMS message region or DB/2 stored procedure WLM started task where the task will execute must be running in delay debug mode. At a minimum this requires that the TEST option be specified, and that an EQAOPTS load module containing the DLAYDBG,YES command be present in the environment's search path.
3. You must have authority to debug tasks started by the given generic user ID. This authority is controlled by the EQADTOOL.GENERICID.generic-user-ID RACF facility.

Note: For the setup items, you may need to confer with your system programmer to ensure that the steps have been performed.

To debug a task stared by a generic user ID, take the following steps:

1. Make a connection to the Terminal Interface Manager.
2. Log on to Terminal Interface Manager with your login credentials. The following panel is displayed:

```
                        z/OS DEBUGGER TERMINAL INTERFACE MANAGER

EQAY001I Terminal TRMLU004 connected for user USRT001
EQAY001I Ready for z/OS Debugger

EQAY001I Data set name : 'USRT001.DLAYDBG.EQAUOPTS' (default -- not loaded)



              PF3=EXIT  PF10=Edit LE options data set  PF12=LOGOFF
```

3. Press PF10 to edit your LE options data set. Ensure that you select the delay debug options data set. The default name for this data set is *userid*.DLAYDBG.EQAUOPTS, but it might have been customized for your site to use a different naming pattern.
4. On the **Edit TEST runtime options data set** panel, press PF8 to access the **Db2/IMS information** panel. Fill in the proper information for the IMS transaction or the DB/2 stored procedure that you want to debug. The following example shows how to debug IMS transaction "DTMQBR" defined in IMS system "IMS1":

```
                     z/OS DEBUGGER TERMINAL INTERFACE MANAGER
                         * Supply additional options *

 Enter IMS identifiers for IMS generic ID support:

IMS Subsystem ID                   : IMS1
IMS Transaction ID                 : DTMQBR

 Enter Db2 identifiers for Db2 generic ID support:

Db2 Stored Procedure Schema        :


Db2 Stored Procedure External Name :




                        PF1=Help  PF4=Save  PF12=Cancel
```

5. Press PF4 to save the IMS or DB/2 information, and then press PF4 again to save the delay debug preferences.
6. When you no longer want to debug the given task, you can deregister for the task by using Terminal Interface Manager to edit the delay debug preferences data set and remove the task information from the IMS/Db2 options panel. You can also log off of Terminal Interface Manager.

Note: When the IMS transaction or DB/2 stored procedure executes under a generic user ID in delay debug mode, z/OS Debugger communicates with Terminal Interface Manager to determine whether a user has signed on and wants to debug the current task.

If the IMS or DB/2 information matches, the TIM user's TSO user ID is returned to z/OS Debugger. z/OS Debugger uses the TSO user ID to open the associated delay debug preferences data set. If the pattern-matching arguments in the delay debug preferences data set match, the debug preference will be used to start the z/OS Debugger user interface.

Appendix A. Data sets used by z/OS Debugger

z/OS Debugger uses the following data sets:

C and C++ source
This data set is used as input to the compiler, and must be kept in a permanent PDS member, sequential file, or HFS or zFS file. The data set must be a single file, not a concatenation of files. z/OS Debugger uses the data set to show you the program as it is executing.

The C and C++ compilers store the name of the source data set inside the load module. z/OS Debugger uses this data set name to access the source.

This data set might not be the original source; for example, the program might have been preprocessed by the CICS translator. If you use a preprocessor, you must keep the data set input to the compiler in a permanent data set for later use with z/OS Debugger.

As this data set might be read many times by z/OS Debugger, we recommend that you do one of the following:

- Define it with the largest block size that your DASD can hold.
- Instruct the system to compute the optimum block size, if your system has that capability.

If you manage your source code with a library system that requires you to specify the SUBSYS=*ssss* parameter when you allocate a data set, you or your site need to specify the EQAOPTS SUBSYS command, which provides the value for *ssss*. This support is not available when debugging a program under CICS. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or the *IBM z/OS Debugger Customization Guide*.

If the following conditions apply to your situation, you do not need access to the source because the .mdbg file has a copy of the source:

- You are compiling with z/OS XL C/C++, Version 1.10 or later.
- You compile your program with the `FORMAT(DWARF)` and `FILE` suboptions of the `DEBUG` compiler option.
- You create an .mdbg file and save (capture) the source with either of the following commands:
  - the dbgld command with the -c option
  - the CDADBGLD command with the CAPSRC option
- You or your site specified YES for the EQAOPTS `MDBG` command[13], which requires z/OS Debugger to search for the .dbg and source file in a .mdbg file.

COBOL listing
This data set is produced by the compiler and must be kept in a permanent PDS member, sequential file, or an HFS or zFS file. z/OS Debugger uses it to show you the program as it is executing.

The COBOL compiler stores the name of the listing data set inside the load module. z/OS Debugger uses this data set name to access the listing.

z/OS Debugger does not use the output that is created by the COBOL `LIST` compiler option.

COBOL programs that have been compiled with the SEPARATE suboption do not need to save the listing file. Instead, you must save the separate debug file SYSDEBUG.

For Enterprise COBOL for z/OS Version 5 and Version 6 Release 1, program listings do not need to be saved. The debug data and the source code is saved in a NOLOAD segment in the program object.

[13] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS `MDBG` command.

© Copyright IBM Corp. 1992, 2023

For Enterprise COBOL for z/OS Version 6 Release 2 and later, program listings do not need to be saved. The debug data and the source code are saved in a NOLOAD segment of the program object if you specified TEST or TEST(NOSEPARATE,SOURCE), and in a separate debug file if you specified TEST(SEPARATE,SOURCE).

The VS COBOL II compilers do not store the name of the listing data set. z/OS Debugger creates a name in the form `userid.cuname.LIST` and uses that name to find the listing.

Because this data set might be read many times by z/OS Debugger, we recommend that you do one of the following:

- Define it with the largest block size that your DASD can hold.
- Instruct the system to compute the optimum blocksize, if your system has that capability.

EQALANGX file

z/OS Debugger uses this data set to obtain debug information about assembler and LangX COBOL source files. It can be a permanent PDS member or sequential file. You must create it before you start z/OS Debugger. You can create it by using the EQALANGX program. Use the SYSADATA output from the High Level assembler or the listing from the IBM OS/VS COBOL, IBM VS COBOL II, or Enterprise COBOL compiler as input to the EQALANGX program.

PL/I source (Enterprise PL/I only)

This data set is used as input to the compiler, and must be kept in a permanent PDS member, sequential file, or HFS or zFS file. z/OS Debugger uses it to show you the program as it is executing.

The Enterprise PL/I compiler stores the name of the source data set inside the load module. z/OS Debugger uses this data set name to access the source.

This data set might not be the original source; for example, the program might have been preprocessed by the CICS translator. If you use a preprocessor, you must keep the data set input to the compiler in a permanent data set, for later use with z/OS Debugger.

Because this data set might be read many times by z/OS Debugger, we recommend that you do one of the following:

- Define it with the largest block size that your DASD can hold.
- Instruct the system to compute the optimum block size, if your system has that capability.

If you manage your source code with a library system that requires you to specify the SUBSYS=*ssss* parameter when you allocate a data set, you or your site need to specify the EQAOPTS SUBSYS command, which provides the value for *ssss*. This support is not available when debugging a program under CICS. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or the *IBM z/OS Debugger Customization Guide*.

PL/I listing (all other versions of PL/I compiler)

This data set is produced by the compiler and must be kept in a permanent file. z/OS Debugger uses it to show you the program as it is executing.

The PL/I compiler does not store the name of the listing data set. z/OS Debugger looks for the listing in a data set with the name in the form of `userid.cuname.LIST`.

z/OS Debugger does not use the output that is created by the PL/I compiler LIST option; performance improves if you specify NOLIST.

Because this data set might be read many times by z/OS Debugger, we recommend that you do one of the following:

- Define it with the largest block size that your DASD can hold.
- Instruct the system to compute the optimum block size, if your system has that capability.

Separate debug file

This data set is produced by the compiler and it stores information used by z/OS Debugger. To produce this file, you must compile your program with the following compiler options:

- The SEPARATE compiler suboption of the TEST compiler option, which is available on the following compilers:
  - Enterprise COBOL for z/OS, Version 6 Release 2 or later. For Enterprise COBOL for z/OS Version 6 Release 2 with APAR PH04485 installed or later, you can specify SEPARATE(DSNAME) to provide the side file location.
  - Enterprise COBOL for z/OS, Version 4
  - Enterprise COBOL for z/OS and OS/390, Version 3
  - COBOL for OS/390 & VM, Version 2 Release 2
  - COBOL for OS/390 & VM, Version 2 Release 1 with APAR PQ40298
  - Enterprise PL/I for z/OS, Version 3.5 or later

  The compiler uses the SYSDEBUG DD statement to specify the separate debug file.
- The FORMAT(DWARF) suboption of the DEBUG compiler option with z/OS C/C++, Version 1.6 or later. The compiler uses one of the following methods to specify the separate debug file (also known as a .dbg file):
  - You specify a name with the FILE suboption
  - You specify a name with the SYSCDBG DD statement
  - If you do not specify a name, the compiler generates a name as described in *z/OS XL C/C++ User's Guide*.

  The file does not contain source. You must also save the source file.

Save the file in any of the following formats:

- a permanent PDS member
- a sequential file
- for COBOL or PL/I, an HFS or zFS file
- for C or C++, a z/OS UNIX System Services file

The compiler stores the data set name of the separate debug file inside the load module. z/OS Debugger uses this data set name to access the debug information, unless you provide another data set name as described in Appendix B, "How does z/OS Debugger locate source, listing, or separate debug files?," on page 407.

Because this data set might be read many times by z/OS Debugger, do one of the following steps to improve efficiency:

- Define it with the largest block size that your DASD can hold.
- Instruct the system to compute the optimum block size, if your system has that capability.

.mdbg file

The .mdbg file is created by the dbgld command or CDADBGLD utility. It contains all the .dbg files for all the programs in a load module or DLL. Beginning with z/OS XL C/C++, Version 1.10, z/OS Debugger can obtain information from this file if it also stores (captures) the source files. Create an .mdbg file with captured source by using the dbgld command with the -c option or the CDADBGLD utility with the CAPSRC option.

To learn how to use these commands, see *z/OS XL C/C++ User's Guide*.

Preferences file

This data set contains z/OS Debugger commands that customize your session. You can use it, for example, to change the default screen colors set by z/OS Debugger. Store this file in a permanent PDS member or a sequential file.

You can specify a preferences file directly (for example, through the TEST runtime option) or through the EQAOPTS PREFERENCESDSN command. For instructions, see "Creating a preferences file" on page 156.

A CICS region must have read authorization to the preferences file.

The preferences file specifications need to be one of the following options:

- RECFM(F) or RECFM(FB) and 32<=LRECL<=256
- RECFM(V) or RECFM(VB) and 40<=LRECL<=264

For more information, see “Creating a preferences file” on page 156.

Global preferences file

This is a preferences file generally available to all users. It is specified through the EQAOPTS GPFDSN command. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or the *IBM z/OS Debugger Customization Guide*. If a global preferences file exists, z/OS Debugger runs the commands in the global preferences file before commands found in the preferences file.

A CICS region must have read authorization to the global preferences file.

The global preferences file specifications need to be one of the following options:

- RECFM(F) or RECFM(FB) and 32<=LRECL<=256
- RECFM(V) or RECFM(VB) and 40<=LRECL<=264

For more information, see command GPFDSN in *IBM z/OS Debugger Customization Guide*.

Commands file

This data set contains z/OS Debugger commands that control the debug session. You can use it, for example, to set breakpoints or set up monitors for common variables. Store it in a permanent PDS member or a sequential file.

If you specify a preferences file, z/OS Debugger runs the commands in the commands file *after* the commands specified in the preferences file.

You can specify a commands file directly (for example, through the TEST runtime option) or through the EQAOPTS COMMANDSDSN command. If it is specified through the EQAOPTS COMANDSDSN command, it must be in a PDS or PDSE and the member name must match the name of the initial load module in the first enclave. For instructions on creating a commands files, see “Creating a commands file” on page 171.

A CICS region must have read authorization to the commands file.

The commands file specifications need to be one of the following options:

- RECFM(F) or RECFM(FB) and 32<=LRECL<=256
- RECFM(V) or RECFM(VB) and 40<=LRECL<=264

For more information, see “Creating a commands file” on page 171 and “Entering multiline commands in a commands file” on page 263.

EQAOPTS file

This data set contains EQAOPTS commands that control initial settings and options for the debug session. Store it in a permanent PDS member or a sequential file. To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *IBM z/OS Debugger Reference and Messages* or the *IBM z/OS Debugger Customization Guide*.

The RECFM must be either F or FB and the LRECL must be 80.

A CICS region must have read authorization to the EQAOPTS file.

EQAUOPTS file

This data set is used to hold parameters for the z/OS Debugger Language Environment user exit or for the delay debug processing.

The EQAUOPTS data set must be a sequential data set with a RECFM of FB or VB, and an LRECL of 80 to 256.

For more information about this data set, see Chapter 12, “Specifying the TEST runtime options through the Language Environment user exit,” on page 97 and “Using delay debug mode to delay starting of a debug session ” on page 397.

Log file

z/OS Debugger uses this file to record the progress of the debugging session. z/OS Debugger stores a copy of the commands you entered along with the results of the execution of commands. The results are stored as comments. This allows you to use the log file as a commands file in subsequent debugging sessions. Store the log file in a permanent PDS member or a sequential file. Because z/OS Debugger writes to this data set, store the log file as a sequential file to relieve any contention for this file.

z/OS Debugger does not use log files in remote debug mode.

The log file specifications need to be one of the following options:

- RECFM(F) or RECFM(FB) and 32<=LRECL<=256
- RECFM(V) or RECFM(VB) and 40<=LRECL<=264

You can specify a log file directly (for example, the INSPLOG DD or the SET LOG command) or through the EQAOPTS LOGDSN command. For instructions, see "Creating the log file" on page 172.

For Db2 stored procedures, to prevent multiple users from trying to use the same log, do not use the EQAOPTS LOGDSN command.

For CICS, review the special circumstances described in "Restrictions when debugging under CICS" on page 354.

Save settings file (SAVESETS)

z/OS Debugger uses this file to save and restore, between z/OS Debugger sessions, the settings from the SET command. A sequential file with RECFM of VB and LRECL>=3204 must be used.

The default name for this data set is *userid*.DBGTOOL.SAVESETS. However, you can change this default by using the EQAOPTS SAVESETDSN command. In non-interactive mode (MVS batch mode without using a full-screen terminal), the DD name used to locate this file is INSPSAFE.

You can not save the settings information in the same file that you save breakpoint and monitor specifications information.

Save settings files are not used for remote debug sessions.

Automatic save and restore of the settings is not supported under CICS if the current user is not logged-in or is logged in under the default user ID. If you are running in CICS, the CICS region must have update authorization to the save settings file.

Save settings files are not supported automatically when debugging Db2 stored procedures.

You or your site can direct z/OS Debugger to create this file and enable saving and restoring settings through the EQAOPTS SAVESETDSNALLOC command. For instructions, see "Saving and restoring settings, breakpoints, and monitor specifications" on page 178.

Save breakpoints and monitor specifications file (SAVEBPS)

z/OS Debugger uses this file to save and restore, between z/OS Debugger sessions, the breakpoints, monitor specifications, and LDD specifications. A PDSE or PDS data set with RECFM of VB and LRECL >= 3204 must be used. (We recommend you use a PDSE.)

The default name for this data set is *userid*.DBGTOOL.SAVEBPS. However, you can change this default by using EQAOPTS SAVEBPDSN command. In non-interactive mode (MVS batch mode without using a full-screen terminal), the DD name used to locate this file is INSPBPM.

You can not save the breakpoint and monitor specifications information in the same file that you save settings information.

Save breakpoints and monitor specifications files are not used for remote debug sessions.

Automatic save and restore of the breakpoints and monitor specifications is not supported under CICS if the current user is not logged-in or is logged in under the default user ID. If you are running in CICS, the CICS region must have update authorization to the save breakpoints and monitor specifications file.

Save breakpoints and monitor specifications files are not supported automatically when debugging Db2 stored procedures.

You or your site can direct z/OS Debugger to create this file and enable saving and restoring breakpoints and monitor specifications through the EQAOPTS SAVEBPDSNALLOC command. For instructions, see “Saving and restoring settings, breakpoints, and monitor specifications” on page 178.

Appendix B. How does z/OS Debugger locate source, listing, or separate debug files?

z/OS Debugger obtains information (called debug information) it needs about a compilation unit (CU) by searching through the following sources.

Remote debugging in standard mode

z/OS Debugger obtains information (called debug information) it needs about a compilation unit (CU) by searching through the following sources:

For remote debugging in standard mode:

- To debug Enterprise COBOL for z/OS V3.4 and V4 programs, specify the location of the listing files by adding the EQAV4LST DD name or the EQA_DBG_V4LIST environment variable. Adding either of those values makes the listing files available to the debugger. For example:

  ```
  //EQAV4LST DD DSN=USER123.COBOL.LISTING,DISP=SHR
  ```

  or

  ```
  ENVAR("EQA_DBG_V4LIST=//'USER123.COBOL.LISTING'")
  ```

  Note: EQAV4LST is only supported in batch.
- To debug Enterprise COBOL for z/OS V6.2 and later programs with the debug information in side files, complete the following steps:

  1. To generate the side files, compile the COBOL program with the TEST(SEPARATE) compiler option, and specify the SYSDEBUG DD name to store the generated side files.
  2. Make the side files available to the debugger by specifying the location of the side files with the EQADEBUG DD name or the EQA_DBG_SYSDEBUG environment variable. For example:

     ```
     //EQADEBUG DD DSN=USER123.COBOL.SYSDEBUG,DISP=SHR
     ```

     or

     ```
     ENVAR("EQA_DBG_SYSDEBUG=USER123.COBOL.SYSDEBUG")
     ```

  If you are using the Source view instead of the default Expanded Source view, make the source files available to the debugger by specifying the location of the source files with the EQASRCE DD name or the EQA_DBG_SRCE environment variable. This is used with data sets that contain only source files. For example:

  ```
  //EQASRCE DD DSN=USER123.COBOL.SOURCE,DISP=SHR
  ```

  or

  ```
  ENVAR("EQA_DBG_SRCE=USER123.COBOL.SOURCE")
  ```
- To debug z/OS C and C++, and 31-bit Enterprise PL/I for z/OS programs with separate side files, make the side files, which contain debugging information, available to the debugger by adding the EQADEBUG DD name to specify the location of the side files. For example:

  ```
  //EQADEBUG DD DSN=USER123.PLI.DEBUG,DISP=SHR
  ```

© Copyright IBM Corp. 1992, 2023

To locate source files for C or C++ programs, add the EQASRCE DD name to specify the location of the application source files. For example:

```
//EQASRCE DD DSN=USER123.CPP.SOURCE,DISP=SHR
```

z/OS Debugger uses the source code compiled by 31-bit Enterprise PL/I for z/OS to display the source. Thus, that source code needs to be in a cataloged data set accessible to the debugger.

To debug 64-bit Enterprise PL/I for z/OS programs, z/OS debugger also uses the source code compiled by Enterprise PL/I for z/OS to display the source. If the source file no longer exists in the location that was provided as the input to the compiler, add the EQADEBUG DD to specify the location of the application source files. For example:

```
//EQADEBUG DD DSN=USER123.PLI64.SOURCE,DISP=SHR
```

- To debug HLASM programs, specify the location of the SYSADATA files by adding the EQAADATA DD name or the EQA_DBG_SYSADATA environment variable. Adding either of those values makes the debugging information available to the debugger. For example:

  ```
  //EQAADATA DD DSN=USER123.HLASM.SYSADATA,DISP=SHR
  ```

  or

  ```
  ENVAR("EQA_DBG_SYSADATA=//'USER123.HLASM.SYSADATA'")
  ```

  z/OS Debugger uses the source code assembled by HLASM to display the source. Thus, that source code needs to be in a cataloged data set accessible to the debugger.
- In the CICS subsystem, you can specify an environment variable using ENVAR (as shown in the previous text, except remove any //) in one of the following ways:
  - In DTCN, specify the environment variable in the **Any other valid Language Environment options** field of the **z/OS Debugger CICS Control - Menu 2** menu.
  - In CADP, specify the environment variable in the **Other Language Environment Options** section.
- If you are using DBM to establish the connection with a remote system, you can modify the timeout value used by Debug Probe to communicate with DBM by specifying an environment variable EQA_DBG_DBMTIMEOUT. For example:

  ```
  ENVAR("EQA_DBG_DBMTIMEOUT=200")
  ```

  The default is 300 seconds.

Non-remote debugging and remote debugging in Debug Tool compatibility mode

z/OS Debugger obtains information (called debug information) it needs about a compilation unit (CU) by searching through the following sources:

- In some cases, the debug information is stored in the load module. z/OS Debugger uses this information, along with the source or listing file, to display source code on the screen.
- For IBM Enterprise COBOL for z/OS, Version 5, Version 6 Release 1, and Version 6 Release 2 or later with the TEST(NOSEPARATE) compiler option programs, z/OS Debugger uses the debug information and the source files that are in a NOLOAD segment in the program object.
- For COBOL and PL/I CUs compiled with the SEPARATE suboption of the TEST compiler option, z/OS Debugger uses the information stored in a separate file (called a separate debug file) that contains both the debug information and the information needed to display source code on the screen.
- For C and C++ CUs created and debugged under the following conditions, z/OS Debugger uses the debug information stored in the .dbg file along with the source file to display code on the screen:
  - Compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option

  - Specified or defaulted to NO for the EQAOPTS MDBG[1] command
- For C and C++ CUs created and debugged under the following conditions, z/OS Debugger uses debug information and source code stored in the .mdbg file to display source code on the screen:
  - Compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option
  - Compiled with z/OS XL C/C++, Version 1.10 or later
  - Created an .mdbg file with saved (captured) source for the load module or DLL by using the -c option of the dbgld command or CAPSRC option of the CDADBGLD utility.
  - Specified YES for the EQAOPTS MDBG[1] command (which requires z/OS Debugger to search for .dbg and source files in a .mdbg file)
- For assembler and LangX COBOL CUs, z/OS Debugger uses the information stored in a separate file (called an EQALANGX file) that contains both the debug information and the information needed to display source code on the screen.

In all of these cases, with the exception of Enterprise COBOL for z/OS Version 6 Release 2 or later compiled with TEST(SEPARATE) or TEST(SEPARATE(NODSNAME)), there is a default data set name associated with each CU, load module, or DLL. The way this default name is generated differs depending on the source language and compiler used. To learn how each compiler generates the default name, see the compiler's programming guide or user's guide.

z/OS Debugger obtains the source or listing data, separate debug file data, or EQALANGX data from one of the following sources:

- the default data set name
- the SET SOURCE command
- the SET DEFAULT LISTINGS command
- the EQADEBUG DD statement

For C and C++ CUs, z/OS Debugger obtains the source data and separate debug file data from different sources, depending on how you created the CU and what value you specified for the EQAOPTS MDBG[1] command. For CUs created and debugged under the following conditions, z/OS Debugger obtains the source data from the source file and separate debug file data from the .dbg file:

- Compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option
- Specified NO for the EQAOPTS MDBG[1] command

z/OS Debugger obtains the source file from one of the following sources:

- the default data set name
- the SET SOURCE command
- the SET DEFAULT LISTINGS command
- the EQAUEDAT user exit (specifying function code 3)
- The EQADEBUG DD name
- the EQA_SRC_PATH environment variable

z/OS Debugger obtains the .dbg file from one of the following sources:

- the default data set name
- the SET DEFAULT DBG command
- the EQAUEDAT user exit (specifying function code 35)
- the EQADBG DD name
- the EQA_DBG_PATH environment variable

Note that these lists do show only what can be processed, not the processing order.

For C and C++ CUs created and debugged under the following conditions, z/OS Debugger obtains the source data and separate debug file data from the .mdbg file:

- Compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option
- Compiled with z/OS XL C/C++, Version 1.10 or later
- Created an .mdbg file with saved (captured) source for the load module or DLL by using the -c option of the dbgld command or CAPSRC option of the CDADBGLD utility.
- Specified YES for the EQAOPTS MDBG[1] command (which requires z/OS Debugger to search for a .dbg file in a .mdbg file)

z/OS Debugger obtains the .mdbg file from one of the following sources:

- the default data set name
- the SET MDBG command
- the SET DEFAULT MDBG command
- the EQAUEDAT user exit (specifying function code 37)
- the EQAMDBG DD statement
- the EQA_MDBG_PATH environment variable

For each type of file (source, listing, separate debug file, .dbg, or .mdbg), z/OS Debugger searches through the sources in different order. The rest of the topics in this chapter describe the order.

If you are using the EQAUEDAT user exit in your environment, the name provided in the user exit takes precedence if z/OS Debugger finds that file.

For .dbg and .mdbg files, z/OS Debugger does not search for the source until it finds a valid .dbg or .mdbg file.

Notes:

1. In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

How does z/OS Debugger locate source and listing files?

z/OS Debugger reads the source or listing file for a CU each time it needs to display information about that CU. While you are debugging your CU, the data set from which the file is read can change. Each time z/OS Debugger needs to read a source or listing file, it searches for the data set in the following order:

1. SET SOURCE command
2. SET DEFAULT LISTINGS command. If the EQAUEDAT user exit is implemented and a EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
3. if present, the EQADEBUG DD statement
4. default data set name. If a data set with the default data set name cannot be located, and if the EQAUEDAT user exit is implemented and a EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.

How does z/OS Debugger locate COBOL source during code coverage

Notes:

- Only programs compiled with IBM Enterprise COBOL for z/OS 6.2 and later are supported.
- Currently, only source files on the host can be located.

z/OS Debugger searches for the source file in the following order:

1. Default data set name
2. EQAUEDAT user exit (specifying function code 44)
3. EQASRCE DD name
4. EQA_SRC_PATH environment variable

The search for the separate debug file is independent and not changed, and follows the order in "How does z/OS Debugger locate COBOL and PL/I separate debug files" on page 411.

How does z/OS Debugger locate COBOL and PL/I separate debug files

z/OS Debugger might read from an Enterprise COBOL for z/OS Version 4 compiler or earlier, Enterprise COBOL for z/OS Version 6 Release 2 or later, or PL/I separate debug file more than once but it always reads the separate debug file from the same data set. After z/OS Debugger locates a valid separate debug file, you cannot direct z/OS Debugger to a different separate debug file. When the CU first appears, z/OS Debugger looks for the separate debug file in the following order:

1. SET SOURCE command
2. default data set name, with the exception of Enterprise COBOL for z/OS Version 6 Release 2 or later compiled with TEST(SEPARATE) or TEST(SEPARATE(NODSNAME)). If a data set with the default data set name cannot be located, and if the EQAUEDAT user exit is implemented and an EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
3. SET DEFAULT LISTINGS command. If the EQAUEDAT user exit is implemented and a EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
4. if present, the EQADEBUG DD name
5. For Enterprise COBOL for z/OS Version 6 Release 2 and later, you can also provide an environment variable EQA_DBG_SYSDEBUG to look for a separate debug file.

For Enterprise COBOL for z/OS Version 6 Release 2 or later, if you use a SET DEFAULT LISTINGS command, EQADEBUG DD name, or EQA_DBG_SYSDEBUG environment variable, and if the separate debug file is not found because the file name does not match the CU name, z/OS Debugger will do an exhaustive search of the data sets specified by the same method to locate the matching debug file. The exhaustive search might be slow.

The Enterprise COBOL for z/OS Version 5 compiler does not create a separate debug file and the commands in this section do not apply.

The SET SOURCE command can be entered only after the CU name appears as a CU and the separate debug file is not found in any of the other locations. The SET DEFAULT LISTINGS command can be entered at any time before the CU name appears as a CU or, if the separate debug file is not found in any of the other possible locations, it can be entered later.

How does z/OS Debugger locate EQALANGX files

An EQALANGX file, which contains debug information for an assembler or LangX COBOL program, might be read more than once but it is always read from the same data set. After z/OS Debugger locates a valid EQALANGX file, you cannot direct z/OS Debugger to a different EQALANGX file. After you enter the LOADDEBUGDATA (LDD) command (which is run immediately or run when the specified CU becomes known to z/OS Debugger), z/OS Debugger looks for the EQALANGX file in the following order:

1. SET SOURCE command
2. a previously loaded EQALANGX file that contains a CSECT that matches the name and length of the program
3. default data set name. If a data set with the default data set name cannot be located, and if the EQAUEDAT user exit is implemented and a EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
4. SET DEFAULT LISTINGS command. If the EQAUEDAT user exit is implemented and a EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
5. the EQADEBUG DD statement

Note: If z/OS Debugger detects a Language Environment-enabled EQAUEDAT when Language Environment is not active, the exit will not be started.

The SET SOURCE command can be entered during any of the following situations:

- Any time after the CU name appears as a disassembly CU.
- If the CU is known when the LDD command is entered but then z/OS Debugger does not find the EQALANGX file.
- If the CU is not known to z/OS Debugger when the LDD command is entered and then z/OS Debugger runs the LDD after the CU becomes known to z/OS Debugger.

The SET DEFAULT LISTINGS command can be entered any time before you enter the LDD command or, if the EQALANGX file is not found by the LDD command, after you enter the LDD command.

How does z/OS Debugger locate the C/C++ source file and the .dbg file?

If you compile with the FORMAT(DWARF) and FILE suboptions of the DEBUG compiler option and specify NO for the EQAOPTS MDBG command[14], z/OS Debugger needs the source file and the .dbg file. The following list describes how z/OS Debugger searches for those files:

- z/OS Debugger reads the source files for a CU each time it needs to display the source code. z/OS Debugger searches for the source file by using the name the compiler saved in the load module or DLL. If you move the source files to a different location, z/OS Debugger searches for the source file based on the input from the following commands, user exit, or environment variable, in the following order:
  1. In full screen mode, the SET SOURCE command.
  2. In remote debug mode, the EQA_SRC_PATH environment variable or what you enter in the **Change Text File** action from the editor view.
  3. The EQADEBUG DD statement.
  4. The EQAUEDAT user exit, specifying function code 3. If you specify the EQADEBUG DD statement, the EQAUEDAT user exit is not run.
  5. The SET DEFAULT LISTINGS command.
- z/OS Debugger might read the .dbg file more than once, but it always reads this file from the same data set. After z/OS Debugger locates this file and validates its contents with the load module being debugged, you cannot redirect z/OS Debugger to search a different file. z/OS Debugger searches for the .dbg file by using the name the compiler saved in the load module or DLL. If you move the .dbg file to a different location, z/OS Debugger searches for the .dbg file based on the input from the following commands, user exit, or environment variable, in the following order:
  1. In remote debug mode, the EQA_DBG_PATH environment variable.
  2. The EQADBG DD statement.
  3. The EQAUEDAT user exit, specifying function code 35. If you specify the EQADBG DD statement, the EQAUEDAT user exit is not run.
  4. The SET DEFAULT DBG command.

To learn more about the DEBUG compiler option, the dbgld command, and the CDADBGLD utility, see *z/OS XL C/C++ User's Guide*.

How does z/OS Debugger locate the C/C++ .mdbg file?

For the following conditions, z/OS Debugger can obtain debug information and source from a module map (.mdbg) file:

- You do one of the following tasks:
  - You or your site specifies YES for the EQAOPTS MDBG command and, for environments that support environment variables, you do not set the environment variable EQA_USE_MDBG to NO.

[14] In situations where you can specify environment variables, you can set the environment variable EQA_USE_MDBG to YES or NO, which overrides any setting (including the default setting) of the EQAOPTS MDBG command.

  - You or your site specifies or defaults to NO for the EQAOPTS MDBG command but, for environments that support environment variables, you override that option by setting the environment variable EQA_USE_MDBG to YES.
- You compile your programs with z/OS XL C/C++, Version 1.10 or later

You use the dbgld command with the -c option or the CDADBGLD utility with the CAPSRC option to save (capture) the source files, as well as all the .dbg files, belonging to the programs that make up a single load module or DLL into one module map file (.mdbg file). Create an .mdbg file with captured source for any load module or DLL that you want to debug because the .mdbg file makes it easier for you to debug the load module or DLL. For example, if your load module is consists of 10 programs and you do not create a module map file, you would need to keep track of 10 .dbg files and 10 source files. If you create a module map file for that load module, you would need to keep track of just one .mdbg file.

z/OS Debugger might read the .mdbg file more than once, but it always reads this file from the same data set. After z/OS Debugger locates this file and validates its contents with the load module being debugged, you cannot redirect z/OS Debugger to search a different file. z/OS Debugger searches for the .mdbg file based on the input from the following commands, user exit, or environment variable, in the following order:

1. The EQAUEDAT user exit, specifying function code 37.
2. If you do not write the EQAUEDAT user exit or the user exit cannot find the file, the default data set name, which is *userid*.mdbg(*load_module_or_DLL_name*), or, in UNIX System Services, ./*load_module_or_DLL_name*.mdbg.

If z/OS Debugger cannot find the .mdbg file, then it searches for the .mdbg file based on the input from the following commands, DD statement, or environment variable, in the following order:

1. The SET MDBG command
2. The SET DEFAULT MDBG command
3. The EQAMDBG DD statement.
4. The EQA_MDBG_PATH environment variable.

To learn more about the DEBUG compiler option, the dbgld command, and the CDADBGLD utility, see *z/OS XL C/C++ User's Guide*.

Appendix C. Examples: Preparing programs and modifying setup files with IBM z/OS Debugger Utilities

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

These examples show you how to use IBM z/OS Debugger Utilities to prepare your programs and how to create, manage, and use a setup file. The examples guide you through the following tasks:

1. Creating personal data sets with the correct attributes.
2. Starting IBM z/OS DebuggerUtilities.
3. Compiling or assembling your program by using IBM z/OS Debugger Utilities. If you do not use IBM z/OS Debugger Utilities, you can build your program through your usual methods and resume this example with the next step.
4. Modifying and using a setup file to run your program in the foreground or in batch.

Creating personal data sets

Create the data sets with the names and attributes described below. Allocate 5 tracks for each of the data sets. Partitioned data sets should be specified with 5 blocks for the directory.

Table 26. Names and attributes to use when you create your own data sets.

Data set name	LRECL	BLKSIZE	RECFM	DSORG
prefix.SAMPLE.COBOL	80	*	FB	PO
prefix.SAMPLE.PLI	80	*	FB	PO
prefix.SAMPLE.C	80	*	FB	PO
prefix.SAMPLE.ASM	80	*	FB	PO
prefix.SAMPLE.DTSF	1280	*	VB	PO
* You can use any block size that is valid.				

Copy the following members of the *hlq*.SEQASAMP data set into the personal data sets you just created:

SEQASAMP member name	Your sample data set	Description of member
EQAWPP1	*prefix*.SAMPLE.COBOL(WPP1)	COBOL source code
EQAWPP3	*prefix*.SAMPLE.PLI(WPP3)	PL/I source code
EQAWPP4	*prefix*.SAMPLE.C(WPP4)	C source code
EQAWPP5	*prefix*.SAMPLE.ASM(WPP5)	Assembler source code
EQAWSU1	*prefix*.SAMPLE.DTSF(WSU1)	setup file for EQAWPP1
EQAWSU3	*prefix*.SAMPLE.DTSF(WSU3)	setup file for EQAWPP3
EQAWSU4	*prefix*.SAMPLE.DTSF(WSU4)	setup file for EQAWPP4
EQAWSU5	*prefix*.SAMPLE.DTSF(WSU5)	setup file for EQAWPP5

© Copyright IBM Corp. 1992, 2023

Starting IBM z/OS Debugger Utilities

To start IBM z/OS Debugger Utilities, do one the following options:

- If IBM z/OS Debugger Utilities was installed as an option on an existing ISPF panel, then select that option.
- If IBM z/OS Debugger Utilities data sets were installed as part of your log on procedure, enter the following command from ISPF option 6:

```
EQASTART
```

- If IBM z/OS Debugger Utilities was installed as a separate application, enter the following command from ISPF option 6:

```
EX 'hlq.SEQAEXEC(EQASTART)'
```

The IBM z/OS Debugger Utilities primary panel (EQA@PRIM) is displayed. On the command line, enter the PANELID command. This command displays the name of each panel on the upper left corner of the screen. These names are used as navigation aids in the instructions provided in this section. After you complete these examples, you can stop the display of these names by entering the PANELID command.

Compiling or assembling your program by using IBM z/OS Debugger Utilities

To compile your program, do the following steps:

1. In panel EQA@PRIM, select 1. Press Enter.
2. In panel EQAPP, select one of the following option and then press Enter.
   - 1 to compile a COBOL program.
   - 3 to compile a PL/I program
   - 4 to compile a C or C++ program
   - 5 to assemble an assembler program
3. One of the following panels is displayed, depending on the language you selected in step 2:
   - EQAPPC1 for COBOL programs. Enter the following information in the fields indicated:
     - Project = *prefix*
     - Group= SAMPLE
     - Type=COBOL
     - Member=WPP1
   - EQAPPC3 for PL/I programs.
     - Project = *prefix*
     - Group= SAMPLE
     - Type=PLI
     - Member=WPP3
   - EQAPPC4 for C and C++ programs.
     - Project = *prefix*
     - Group= SAMPLE
     - Type=C
     - Member=WPP4
   - EQAPPC5 for assembler programs.

- Project = *prefix*
- Group= SAMPLE
- Type=ASM
- Member=WPP5

4. If you are preparing an assembler program, enter the location of your CEE library in the Syslib data set Name field. For example: 'CEE.SCEEMAC'
5. Enter '/' to edit options and specify a naming pattern for the output data sets in the field `Data set naming pattern`. Press Enter.
6. One of the following panels is displayed, depending on the language you selected in step 2:

   - EQAPPC1A for COBOL programs.
   - EQAPPC3A for PL/I programs.
   - EQAPPC4A for C and C++ programs.
   - EQAPPC5A for assembler programs.

   Look at the panel to review the following information:

   - test compiler options
   - naming patterns for output data sets

   Press PF3 (Exit).
7. One of the following panels is displayed, depending on the language you selected in step 2:

   - EQAPPC1 for COBOL programs.
   - EQAPPC3 for PL/I programs.
   - EQAPPC4 for C and C++ programs.
   - EQAPPC5 for assembler programs.

   Select "F" to process these programs in the foreground. Specify "N" for CICS translator and "N" for Db2 precompiler. None of these programs contain CICS or Db2 instructions. Press Enter.
8. One of the following panels is displayed, depending on the language you selected in step 2:

   - EQAPPC1B for COBOL programs.
   - EQAPPC3B for PL/I programs.
   - EQAPPC4B for C and C++ programs.
   - EQAPPC5B for assembler programs.

   Make a note of the data set name for Object compilation output. For a COBOL program, the data set name will look similar to the following name: *prefix*`.SAMPLE.OBJECT(WPP1)`. You will use this name when you link your object modules. Press Enter.
9. If panel EQAPPA1 is displayed, press Enter.
10. One of the following panels is displayed, depending on the language you selected in step 2:

    - EQAPPC1C for COBOL programs.
    - EQAPPC3C for PL/I programs.
    - EQAPPC4C for C and C++ programs.
    - EQAPPC5C for assembler programs.

    Check for a 0 or 4 return code. Type a "b" in the Listing field. Press Enter.
11. In panel ISRBROBA, browse the file to review the messages. When you are done reviewing the messages, press PF3 (Exit).
12. One of the following panels is displayed, depending on the language you selected in step 2:

    - EQAPPC1C for COBOL programs.

- EQAPPC3C for PL/I programs.
- EQAPPC4C for C and C++ programs.
- EQAPPC5C for assembler programs.

Press PF3 (Exit).

13. One of the following panels is displayed, depending on the language you selected in step 2:
    - EQAPPC1B for COBOL programs.
    - EQAPPC3B for PL/I programs.
    - EQAPPC4B for C and C++ programs.
    - EQAPPC5B for assembler programs.

    Press PF3 (Exit).
14. One of the following panels is displayed, depending on the language you selected in step 2:
    - EQAPPC1 for COBOL programs.
    - EQAPPC3 for PL/I programs.
    - EQAPPC4 for C and C++ programs.
    - EQAPPC5 for assembler programs.

    Press PF3 (Exit).
15. In panel EQAPP, press PF3 (Exit) to return to EQA@PRIM panel.

To link your object modules, do the following steps:

1. In panel EQA@PRIM, select 1. Press Enter.
2. In panel EQAPP, select L. Press Enter.
3. In panel EQAPPCL, specify "F" to process the programs in the foreground. Then, choose one of the following options, depending on the language you selected in step 2
    - For the COBOL program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=OBJECT, Member=WPP1
    - For the PL/I program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=OBJECT, Member=WPP3
    - For the C program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=OBJECT, Member=WPP4
    - For the assembler program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=OBJECT, Member=WPP5
4. In panel EQAPPCL, specify the name of the other libraries you need to link to your program. For example, in the field Syslib data set Name, specify the prefix of your CEE library: `'CEE.SCEELKED'`. Press Enter.
5. In panel EQAPPCLB, make a note of the data set name in the Load link-edit output field. You will use this name when you modify a setup file. Press Enter.
6. If panel EQAPPA1 is displayed, press Enter.
7. In panel EQAPPCLC, check for a 0 return code. Type a "V" in the Listing field. Press Enter.
8. In panel ISREDDE2, review the messages. After you review the messages, press PF3 (Exit).
9. In panel EQAPPCLC, press PF3 (Exit).
10. In panel EQAPPCLB, press PF3 (Exit).
11. In panel EQAPPCL, press PF3 (Exit).
12. In panel EQAPP, press PF3 (Exit) to return to EQA@PRIM panel.

Modifying and using a setup file

This example describes how to modify a setup file and then use it to run the examples in the TSO foreground or run the examples in the background by submitting a MVS batch job.

Run the program in foreground

To modify and run the setup file so your program runs in the foreground, do the following steps:

1. In panel EQA@PRIM, select 2. Press Enter.
2. In panel EQAPFOR, select one of the following choices, depending on which language you selected in step 2 in topic "Compiling or assembling your program by using IBM z/OS Debugger Utilities" on page 416:
   - For the COBOL program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=DTSF, Member = WSU1
   - For the PL/I program, use the following values for each field: Project = *prefix*, Group = SAMPLE, Type=DTSF, Member=WSU3
   - For the C program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=DTSF, Member=WSU4
   - For the assembler program, use the following values for each field: Project = *prefix*, Group= SAMPLE, Type=DTSF, Member=WSU5

   Press Enter.
3. In panel EQAPFORS, do the following steps:
   a. Replace &LOADDS. with the name of the load data set from step 5 in topic "Compiling or assembling your program by using IBM z/OS Debugger Utilities" on page 416:
   b. Replace &EQAPRFX. with the prefix your EQAW (z/OS Debugger) library.
   c. Replace &CEEPRFX. with the prefix your CEE (Language Environment) library.
   d. Enter "e" in Cmd field next to CMDS DD name. In the window that is displayed, if there is a QUIT ; statement at the end of the data set, remove it. Press PF3 (Exit).
   e. Type "run" in command line. Press Enter.
4. z/OS Debugger is started and the z/OS Debugger window is displayed. Enter any valid z/OS Debugger commands to verify that you can debug the program. Enter "qq" in the command line to stop z/OS Debugger and close the z/OS Debugger window.
5. In panel EQAPFORS, check the return code message:
   - For the COBOL program, the return code (RC) is 0.
   - For the PL/I program, the return code (RC) is 1000.
   - For the C program, the return code (RC) is 0.
   - For the assembler program, the return code (RC) is 0.

   Press PF3 (Exit). All the changes made to the setup file are saved.
6. In panel EQAPFOR, press PF3 (Exit) to return to the panel EQA@PRIM.

Run the program in batch

To modify and run the setup file so that the program runs in batch, do the following steps:

1. In panel EQA@PRIM, select 0. Press Enter.
2. In panel EQAPDEF, review the job card information. If there are any changes that need to be made, make them. Press PF3 (Exit).
3. In panel EQA@PRIM, select 2. Press Enter.

4. In panel EQAPFOR, select one of the following choices, depending on which language you selected in step 2 in topic "Compiling or assembling your program by using IBM z/OS Debugger Utilities" on page 416:
   - For the COBOL program, use the following values for each field: Project = *prefix*, Group = SAMPLE, Type = DTSF, Member =WSU1
   - For the PL/I program, use the following values for each field: Project = *prefix*, Group = SAMPLE, Type = DTSF, Member =WSU3
   - For the C program, use the following values for each field: Project = *prefix*, Group = SAMPLE, Type = DTSF, Member = WSU4
   - For the assembler program, use the following values for each field: Project = *prefix*, Group = SAMPLE, Type = DTSF, Member = WSU5

   Press Enter.
5. If you ran the steps beginning in step 1 of topic "Run the program in foreground" on page 419 you can skip this step. In panel EQAPFORS, do the following steps:
   a. Replace `&LOADDS.` with the name of the load data set from step 5 in topic "Compiling or assembling your program by using IBM z/OS Debugger Utilities" on page 416.
   b. Replace `&EQAPRFX.` with the prefix your EQAW (z/OS Debugger) library.
   c. Replace `&CEEPRFX.` with the prefix your CEE (Language Environment) library.
6. Enter "e" in the Cmd field next to CMDS DD name. If there is not `'QUIT ;'` statement at the end of the data set, then add the statement. Press PF3 (Exit).
7. Type `submit` in command line. Press Enter.
8. In panel ISREDDE2, type `submit` in the command line. Press Enter. Make a note of the job number that is displayed.
9. In panel ISREDDE2, press PF3 (Exit).
10. In panel EQAPFORS, press PF3 (Exit). The changes you made to the setup file are saved.
11. In panel EQAPFOR, press PF3 (Exit) to return to EQA@PRIM panel. locate the job output using the job number recorded. Check for zero return code and the command log output at the end of the job output.

Appendix D. IBM z/OS Debugger JCL Wizard

By using the EQAJCL ISPF edit macro, you can modify a JCL or procedure member and create statements to invoke z/OS Debugger in various environments.

You can build control statements to complete the following tasks:

- Invoke z/OS Debugger by using the Terminal Interface Manager (**T**).
- Invoke z/OS Debugger by using the remote debugger in an Eclipse IDE, either with Debug Manager (**G2**) or the workstation TCP/IP address (**G1**).
- Invoke z/OS Debugger for Language Environment (LE) or non-Language Environment (non-LE) programs.
- Request z/OS Debugger Code Coverage invocation with an interactive debug session (**Code Coverage** in the parameters panel for Terminal Interface Manager).
- Request z/OS Debugger Code Coverage invocation without an interactive debug session (**C**).
- Define the libraries to search for z/OS Debugger source and debug information (**Debugger Libs** in the parameters panel).

  Note: If the program name or CSECT name for assembler is not the member name of the debug file, the wizard presents a list of members for each debug file, and then users can select the corresponding member name.
- Enter the program names that require LDD statements (**LDD Programs** in the parameters panel).
- Set AT ENTRY breakpoints at subprograms (**At Entry** in the parameters panel).
- Invoke the automonitor to show variables as you step through lines of code (**Automonitor on** in the parameters panel).
- Show COBOL DISPLAY statements on the IBM z/OS Debugger log or Debug Console in the Eclipse IDE (**Intercept on** in the parameters panel).
- Allow variable changes for optimized programs (**Warning off** in the parameters panel).
- Enable SVC screening for batch non-Language Environment programs (**SVC Screening** in the parameters panel).
- Add a DD statement to capture the IBM z/OS Debugger log (**z/OS Debugger Log** in the parameters panel for Terminal Interface Manager).
- Show comments that depict how to access subprogram source and debug information before the program is loaded into storage (**Show Comments** in the parameters panel).
- Request a delayed debug session (**D**).
- Remove z/OS Debugger control statements (**R**).

For a list of all the EQAJCL commands and options in the parameters panel, see “Commands and parameters in IBM z/OS Debugger JCL Wizard” on page 425.

The IBM z/OS Debugger JCL Wizard can create in-stream JCL statements. For procedures or included members, in-stream JCL statements, such as `//SYSIN DD *`, are valid only in JES2 systems with z/OS 1.13 or later, or JES3 systems with z/OS 2.1 or later. If you do not run IBM z/OS Debugger JCL Wizard in one of the environments that are described above, submitting JCL invoking procedures with in-stream control statements fails.

Generation of the EQAMDBG DD statement for C/C++ mdbg files is not supported.

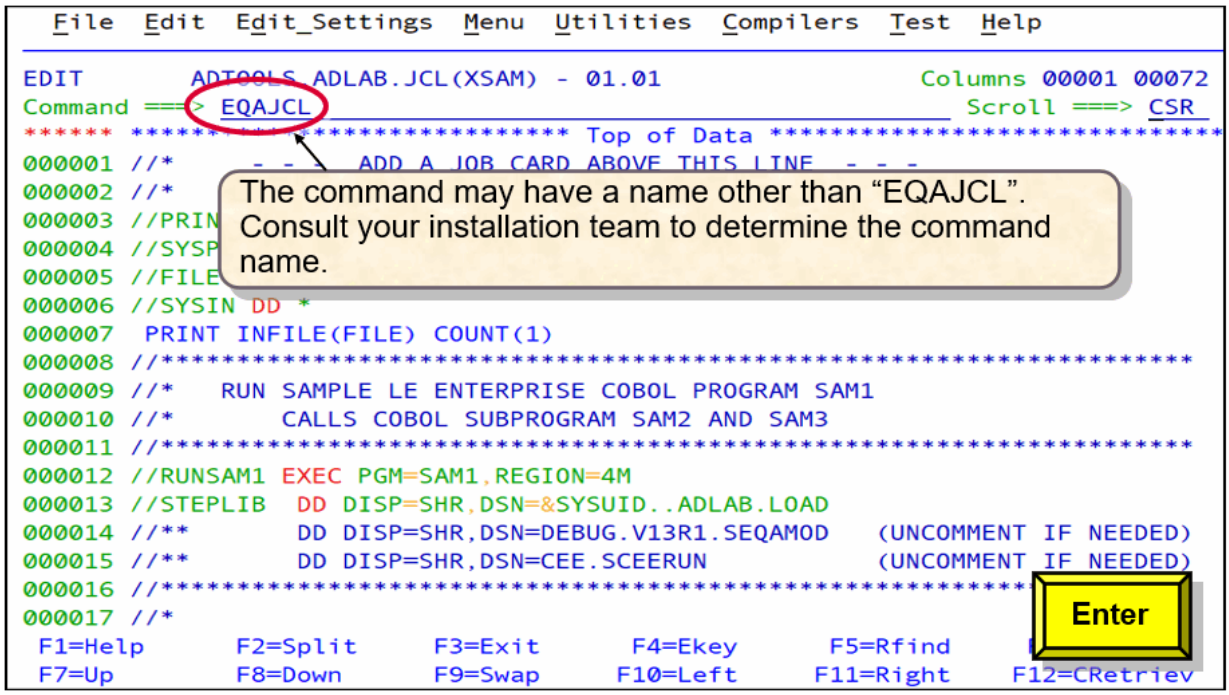
© Copyright IBM Corp. 1992, 2023

Invoking the IBM z/OS Debugger JCL Wizard

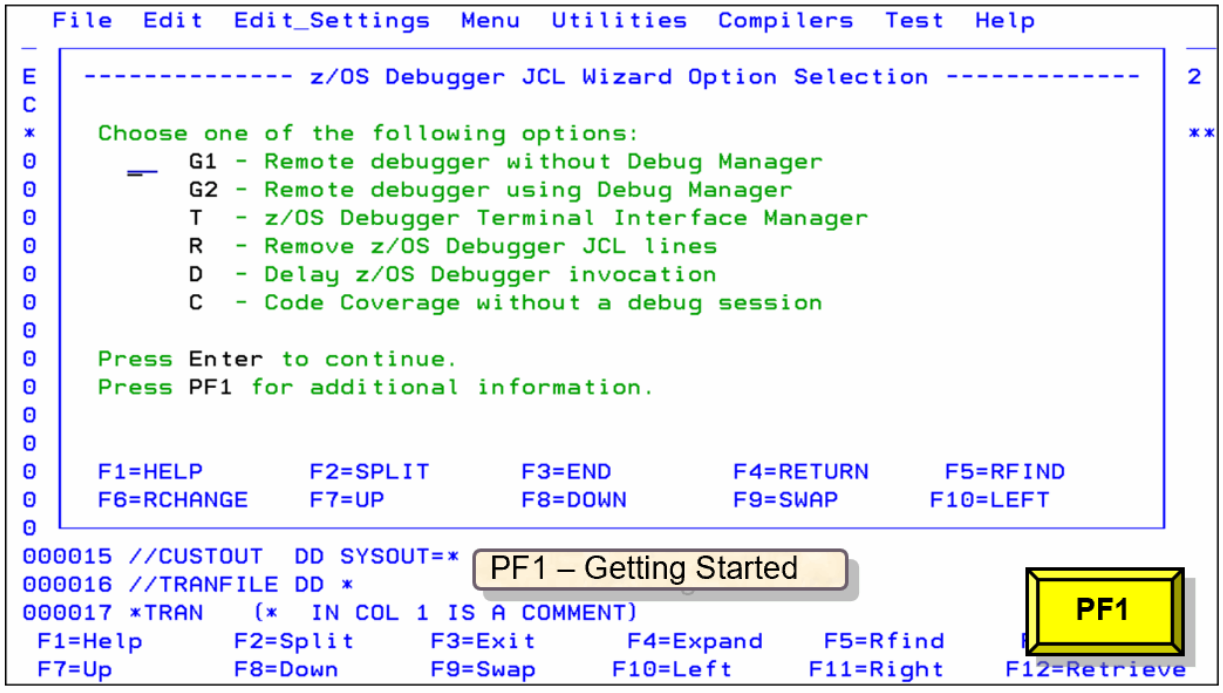
You can invoke the IBM z/OS Debugger JCL Wizard when you are editing or viewing JCL, a procedure, or an include member in ISPF.

Your environment might be customized to use another name rather than EQAJCL, such as DEBUG. In this documentation, the name EQAJCL is used to invoke the wizard.

1. In ISPF Edit or Browse, enter **EQAJCL** and press **Enter** to continue. In the IBM z/OS Debugger JCL Wizard, always press **ENTER** to save your selection and continue to the next panel.



2. The **z/OS Debugger JCL Wizard Option Selection** panel is displayed for you to choose an option.



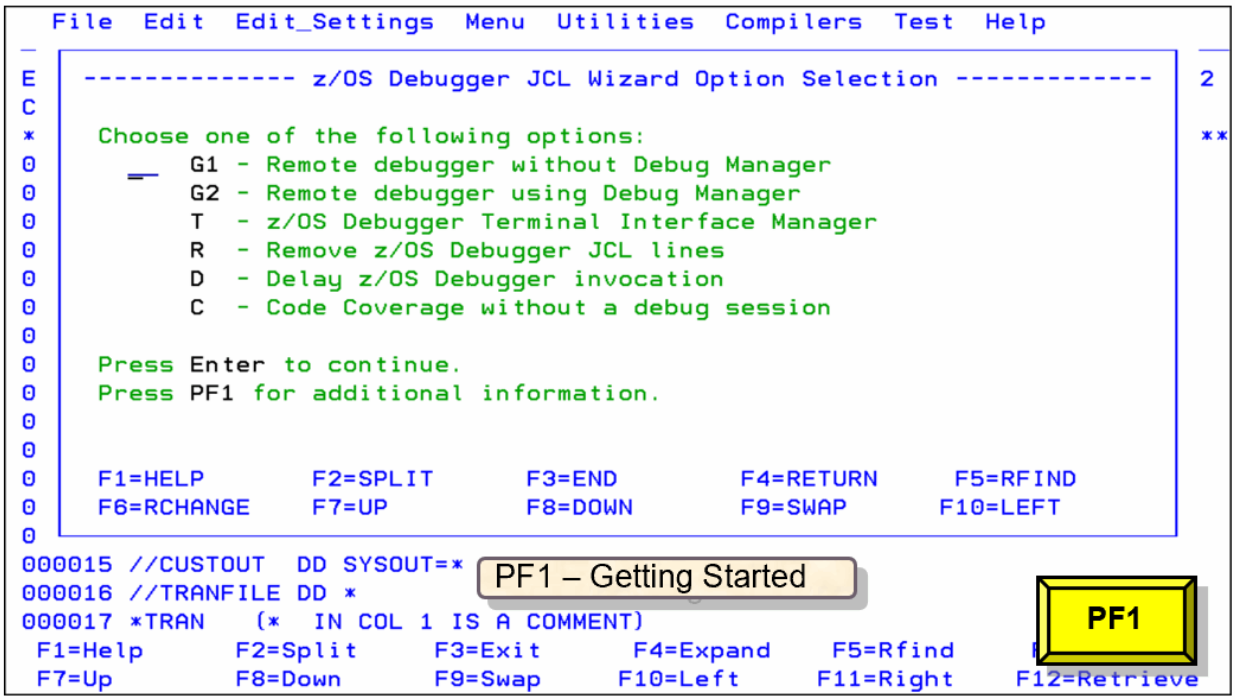
3. Choose the option that you want. For example, if you want to create JCL or procedure lines to invoke a debug session on a remote debugger with Debug Manager, enter **G2**. Alternatively, you can enter

EQAJCL G2 in ISPF Edit or Browse to skip the **z/OS Debugger JCL Wizard Option Selection** panel and open the parameters panel for the selected option directly.

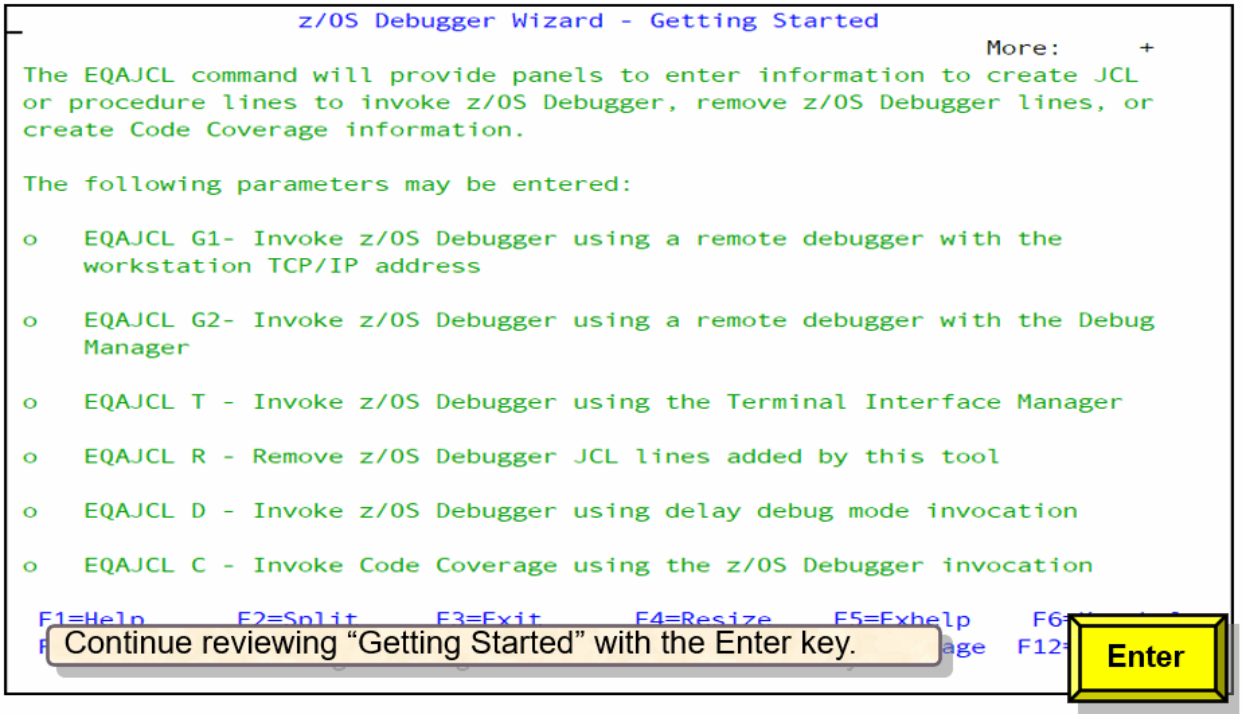
Viewing help in IBM z/OS Debugger JCL Wizard panels

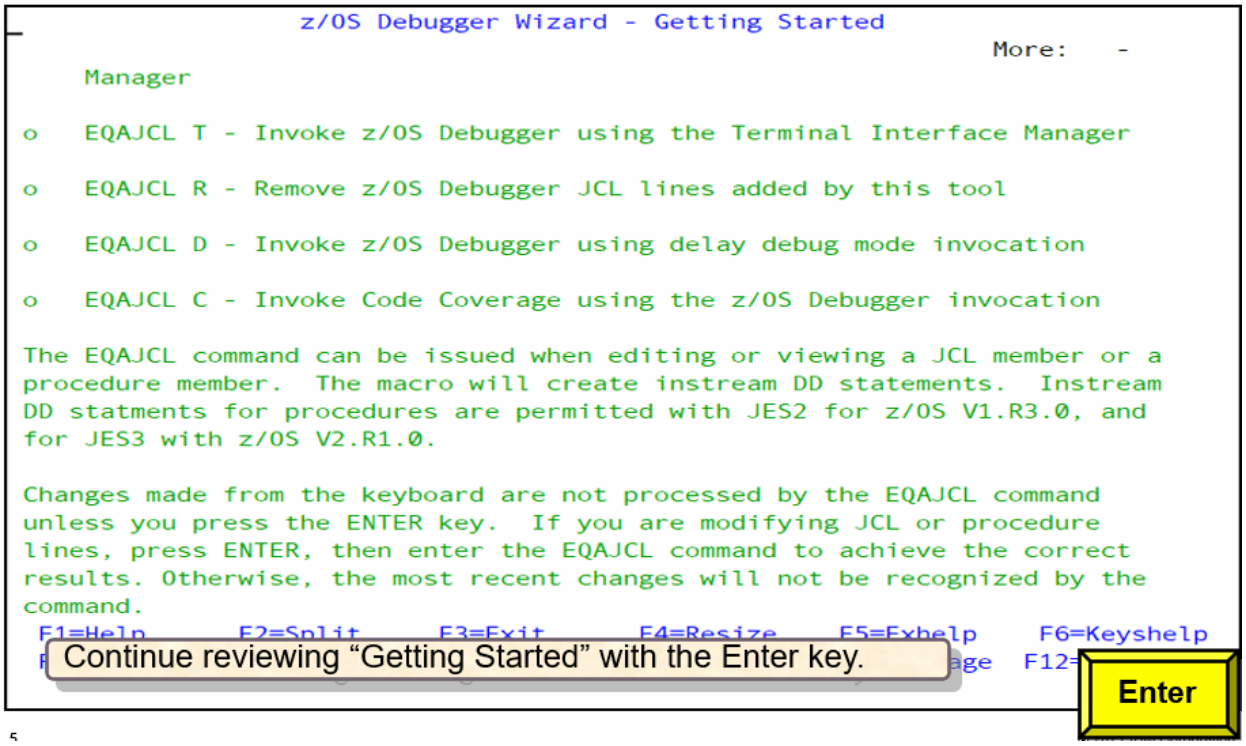
You can get the help information for the wizard or a field by using **PF1**.

- To get the help information for the wizard, select **PF1** in the IBM z/OS Debugger JCL Wizard panels.

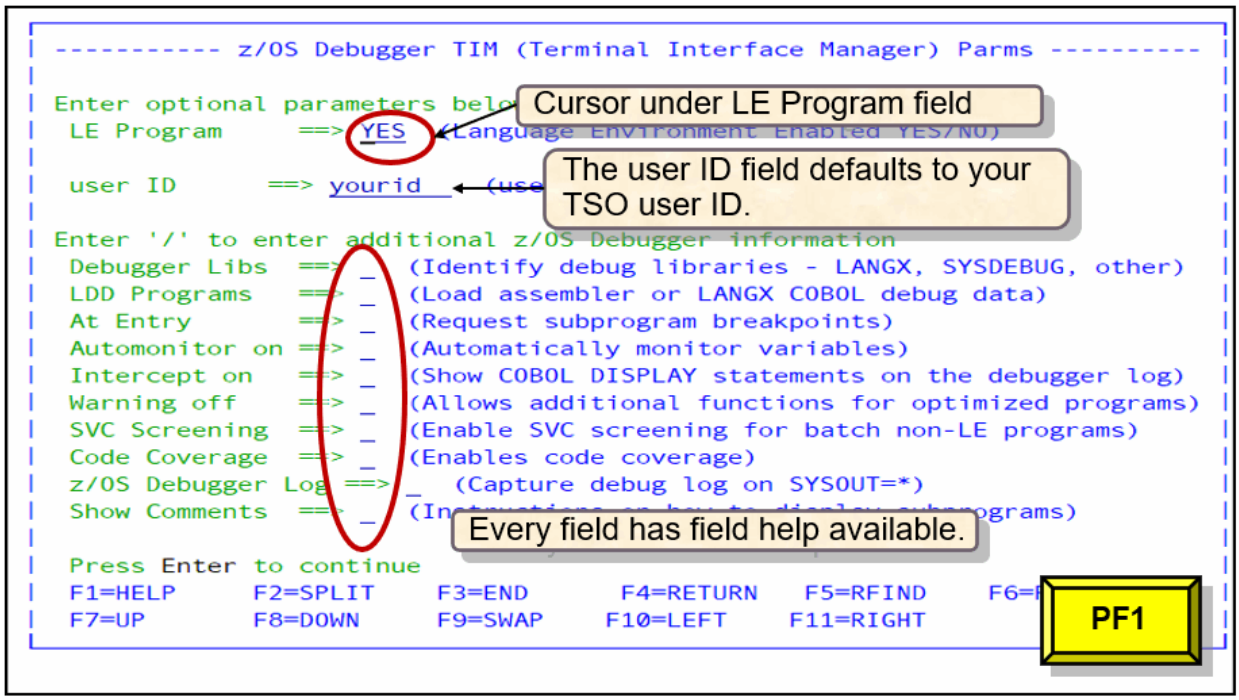


The **z/OS Debugger Wizard Getting Started** help panel is displayed. You can press **Enter** to continue reviewing.

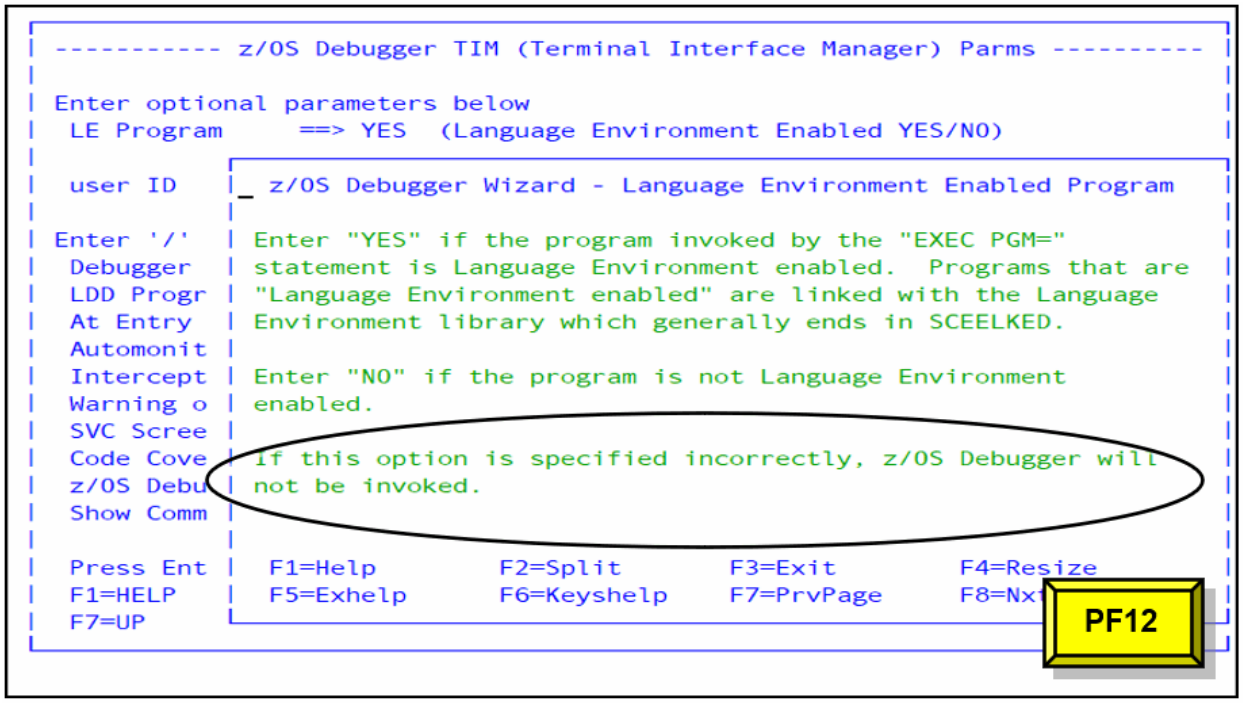




- To view the field help, place the cursor in the field and press **PF1**.



The field help is displayed.



Commands and parameters in IBM z/OS Debugger JCL Wizard

Commands

In ISPF Edit or Browse, after you use **EQAJCL** invoke IBM z/OS Debugger JCL Wizard. The following options are available in **z/OS Debugger JCL Wizard Option Selection** panel:

G1
Creates JCL to invoke z/OS Debugger by using the remote debugger in an Eclipse IDE with the workstation TCP/IP address.

G2
Creates JCL to invoke z/OS Debugger by using the remote debugger in an Eclipse IDE with Debug Manager.

T
Creates JCL to invoke z/OS Debugger by using the Terminal Interface Manager. The Terminal Interface Manager is a 3270 interface.

R
Removes the JCL created by the IBM z/OS Debugger JCL Wizard.

D
Creates JCL to invoke z/OS Debugger by using delayed debug mode. This mode is generally used with a special STEPLIB library that is required in your JCL. Delayed debug mode allows the first invocation of z/OS Debugger to be at a subprogram.

C
Creates code coverage information for programs that are compiled with Enterprise COBOL or Enterprise PL/I. This option requests z/OS Debugger Code Coverage invocation without an interactive debug session. To gather code coverage in a debug session, use **T** and then select **Code Coverage** from the parameters panel.

Parameters

In the parameters panels, the following parameters are available:

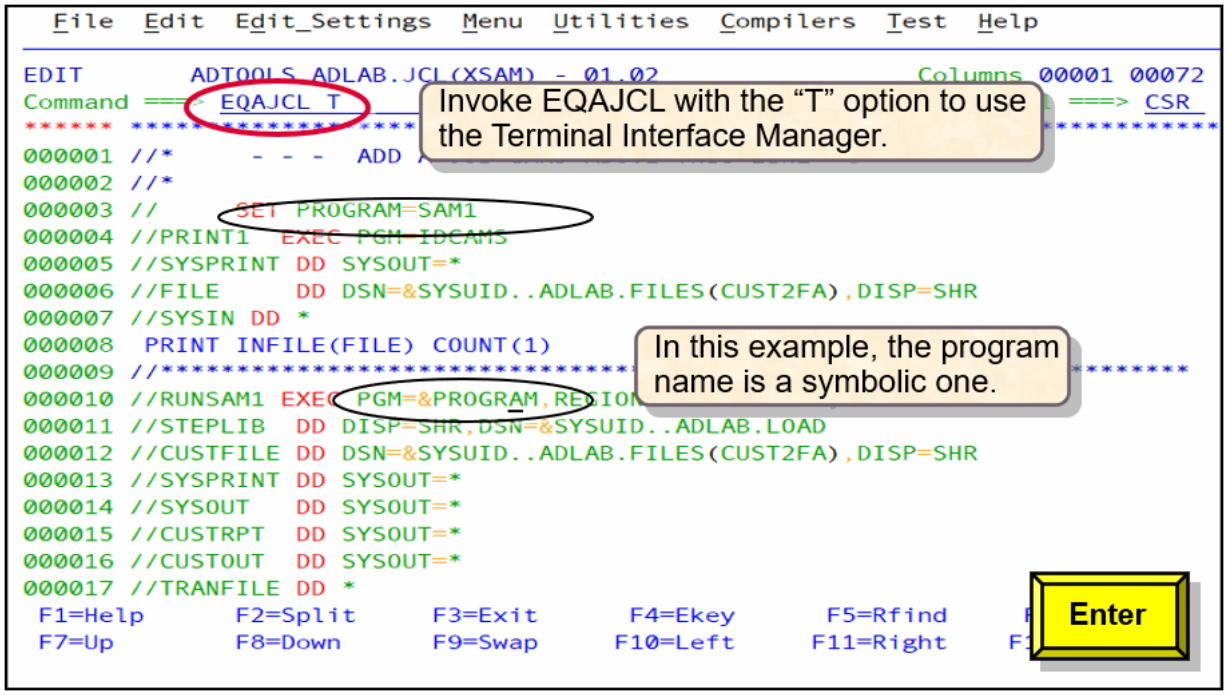
- **Debugger Libs**: Required by some programs to identify where the side file information is located.

- **LDD Programs**: Required by non-Language Environment programs to generate a Load Debug Data (LDD) command for the program.
- **At Entry**: Set breakpoints at subprograms by using the AT ENTRY command.
- **Automonitor on**: Automatically monitor variables.
- **Intercept on**: Show COBOL DISPLAY statements in the Terminal Interface Manager log or Debug Console in the Eclipse IDE.
- **Warning off**: Allow variable changes for optimized programs.
- **SVC Screening**: Required when a Language Environment program calls non-Language Environment assembler programs in non-CICS environments.
- **Code Coverage**: Enable code coverage.
- **z/OS Debugger Log**: Add a DD statement to capture the IBM z/OS Debugger log. This parameter is available only for Terminal Interface Manager (**T**).
- **Show Comments**: Show instructions to view subprogram source information. You can select to show comments that depict how to set breakpoints in a subprogram before invocation.

Debugging a Language Environment program using Terminal Interface Manager

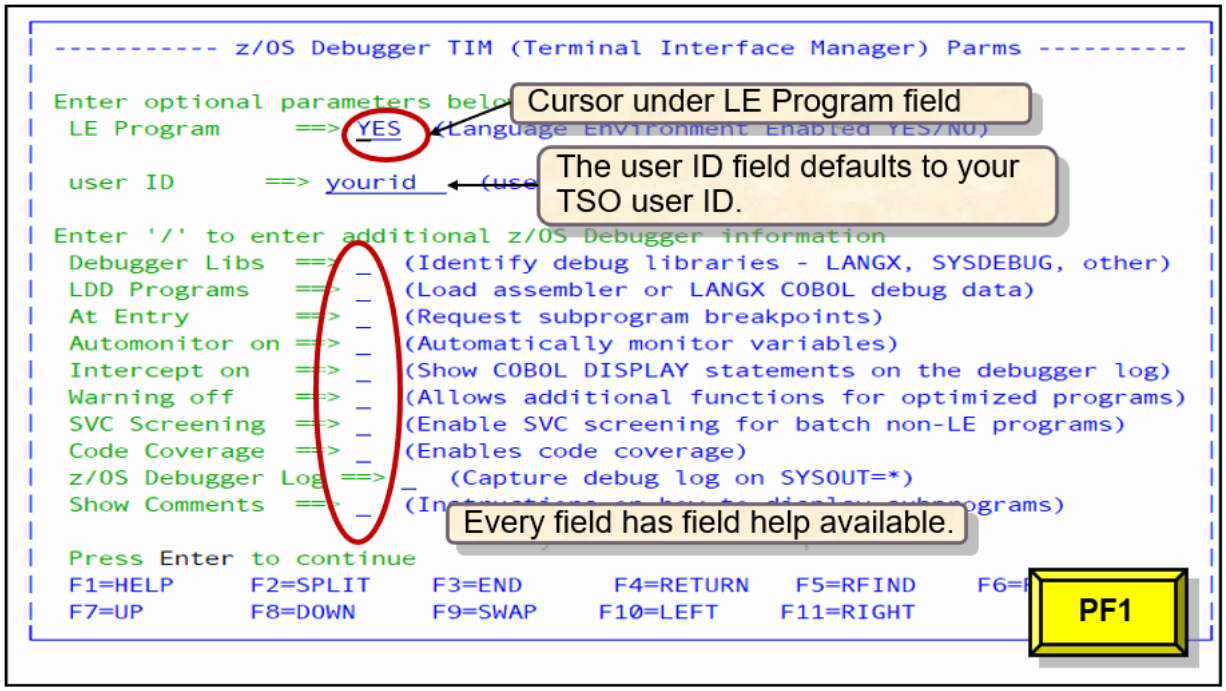
You can use IBM z/OS Debugger JCL Wizard to create JCL statements to debug a Language Environment program by using the Terminal Interface Manager.

1. In ISPF Edit or Browse, enter **EQAJCL T** and press **Enter** to invoke the Terminal Interface Manager. Alternatively, you can enter **EQAJCL** to display the **z/OS Debugger JCL Wizard Option Selection** panel and then enter **T**.

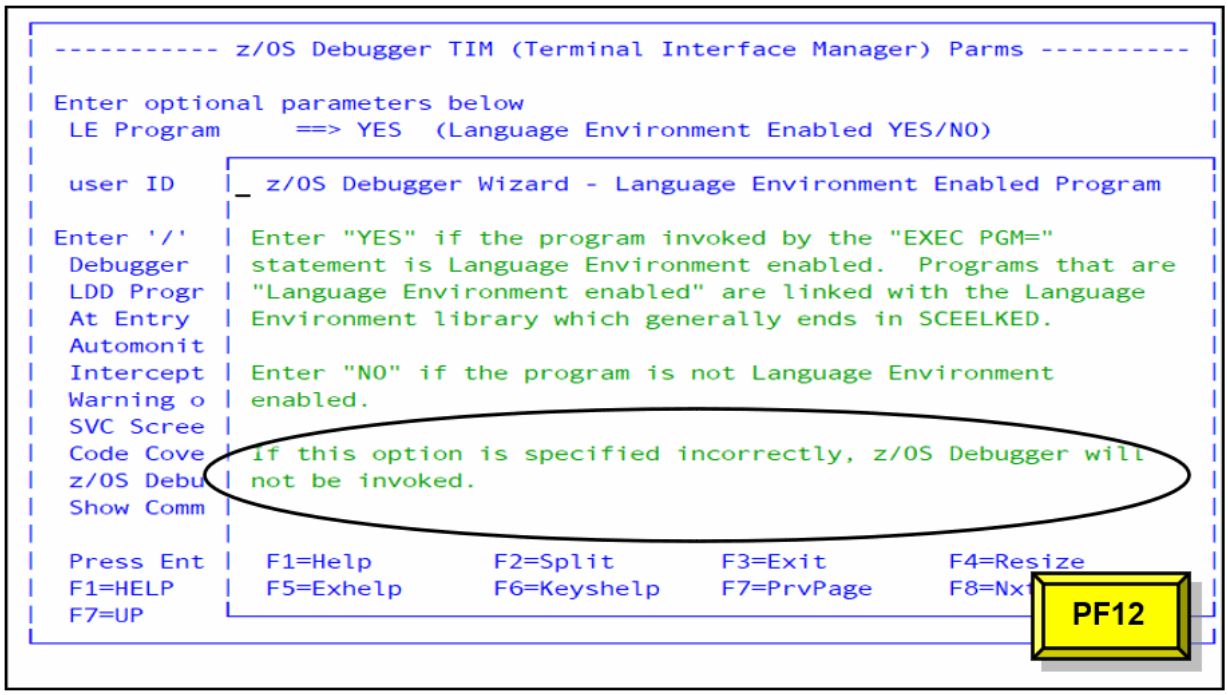


2. In the parameters panel, specify **YES** in the **LE Program** field.

   To view the field help that is associated with the **LE Program** field, place the cursor in the field and press **PF1**.



The field help for the **LE Program** field is displayed.



The field help indicates that you enter **YES** if the program invoked by the EXEC PGM= statement is Language Environment enabled and enter **NO** if the program is not Language Environment enabled.

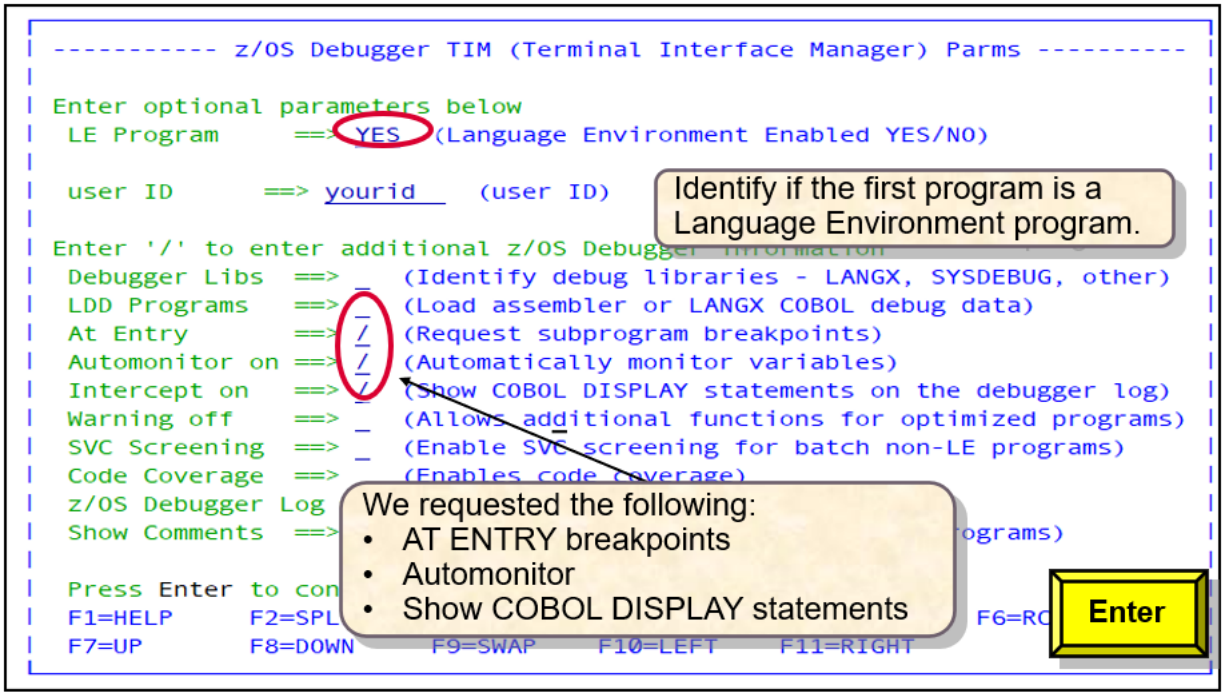
Programs that are Language Environment enabled are linked with the Language Environment library that generally ends in SCEELKED.

- Enterprise COBOL and PL/I programs are Language Environment enabled.
- Assembler programs and VS COBOL II programs might or might not be Language Environment enabled.
- OS/VS COBOL programs are not Language Environment enabled.

If the **LE Program** field is set incorrectly, z/OS Debugger will not be invoked.

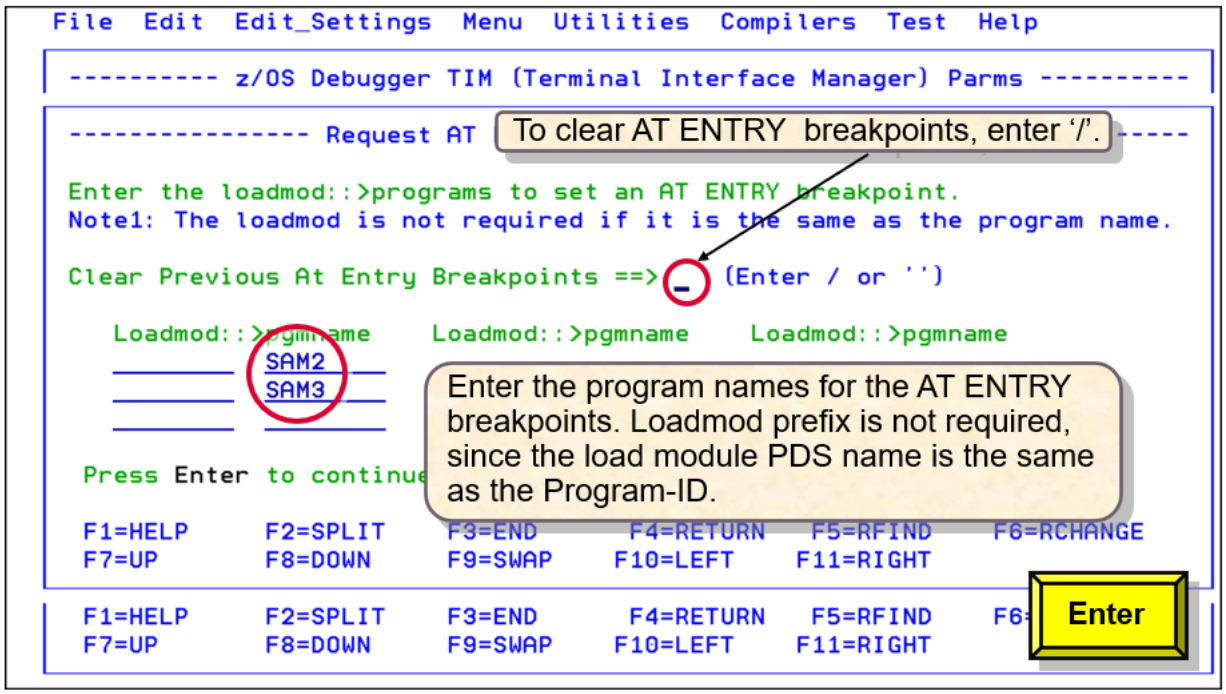
3. To select an optional parameter, specify a forward slash (/) in the field.

In this use case, **At Entry**, **Automonitor on**, and **Intercept on** are selected.

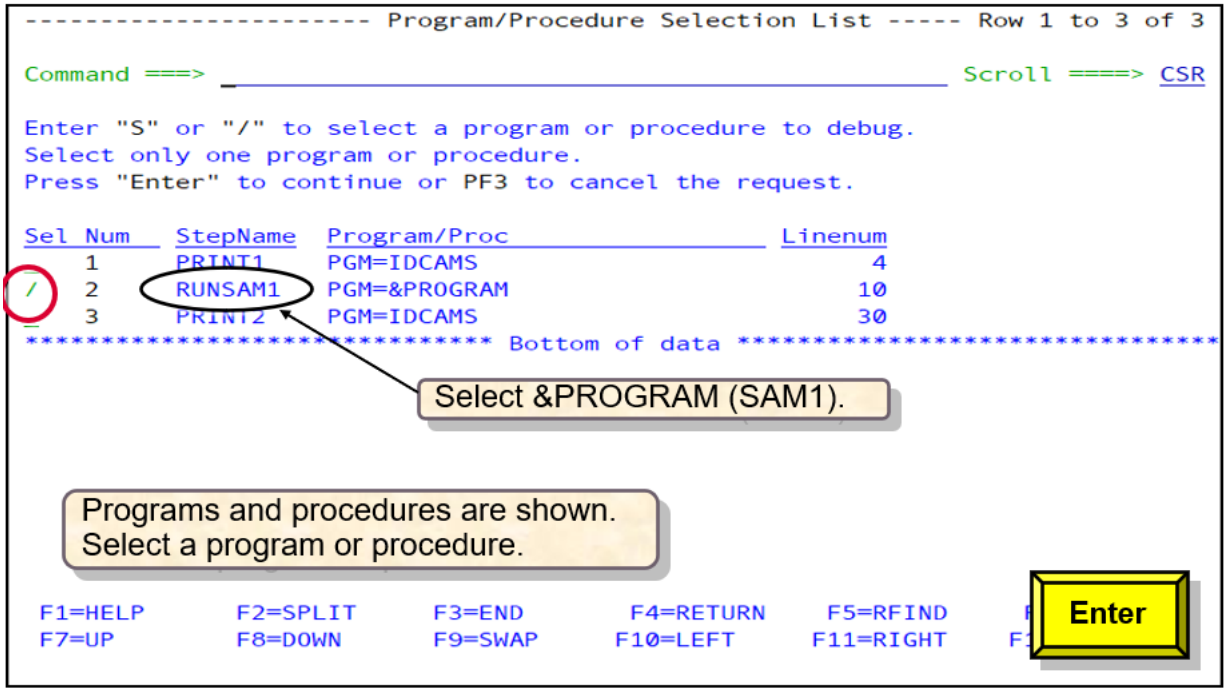


4. Since you chose to set AT ENTRY breakpoints for subprograms, the **Request AT ENTRY Sub-Program Breakpoints** panel is displayed. If you are saving your settings or using the remote debugger, AT ENTRY breakpoints are remembered between debug sessions. To clear previous AT ENTRY breakpoints before setting new ones, enter the forward slash (/) in the field.

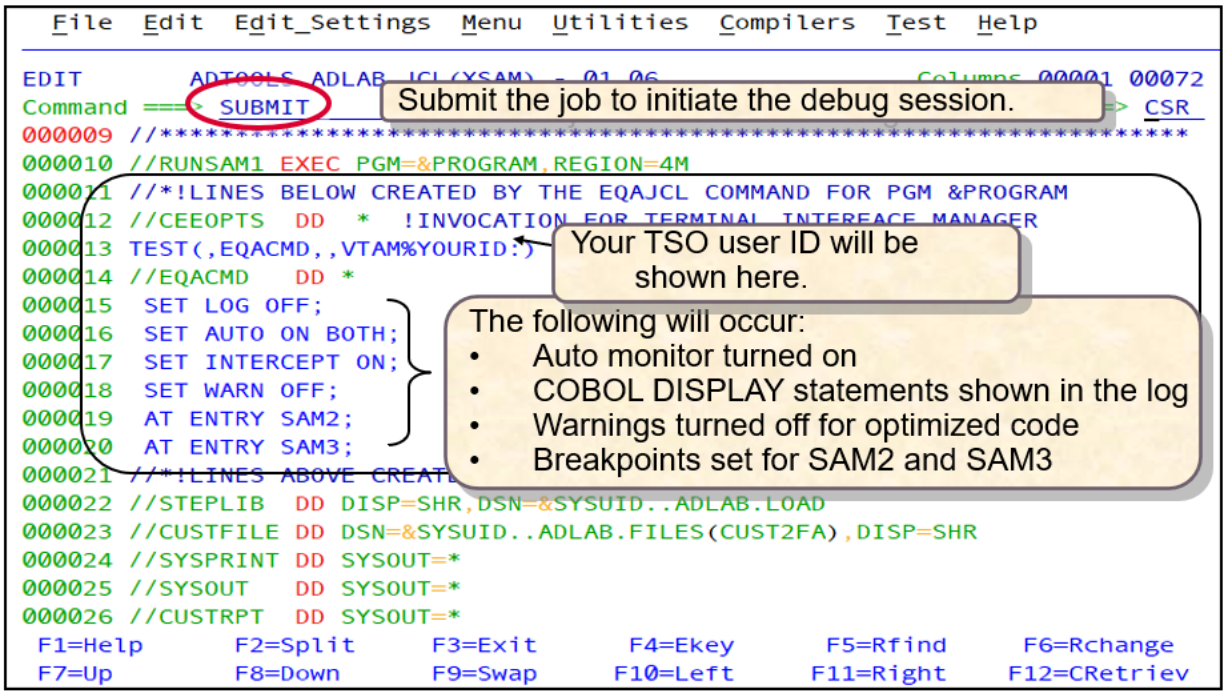
   Note: z/OS subprograms can be linked as static or dynamic. A static subprogram is included in the load module of the main program. A dynamic program is not included in the load module of the main program, but dynamically loaded into storage when the first `CALL` or a `LOAD` statement is issued. If you are using the remote debugger, and the load module name is different than the program name. Be sure to enter the load module name to identify the correct AT ENTRY breakpoint. A program name is the `PROGRAM-ID` for COBOL, the first `CSECT` for assembler, or the label defined on the `MAIN` procedure of a PL/I program. For statically linked modules, you need the load module name of the main program, and the program name of the subprogram name where you want to stop.



5. In the **Program/Procedure Selection List** panel, a list of job steps is displayed if more than one EXEC PGM statement is found in the JCL or procedure. Select only one program or procedure to debug.



6. JCL statements are generated to invoke z/OS Debugger.



- The first and last generated lines are comment lines. Do not modify these comment lines. The comment lines explain how to locate a program, and set breakpoints before invocation of the subprogram for the Terminal Interface Manager.
- The //CEEOPTS statement defines the EQACMD command file DD name, and the VTAM% user ID information that is used to invoke the Terminal Interface Manager.
- The SET LOG OFF command indicates you do not want to retain log information. Previous AT ENTRY breakpoints are cleared, and breakpoints for subprograms SAM2 and SAM3 are set.

You can also use the IBM z/OS Debugger JCL Wizard to create IBM z/OS Debugger invocation of Db2, or IMS batch JCL or procedures.

What to do next

You can now invoke a debug session with the Terminal Interface Manager:

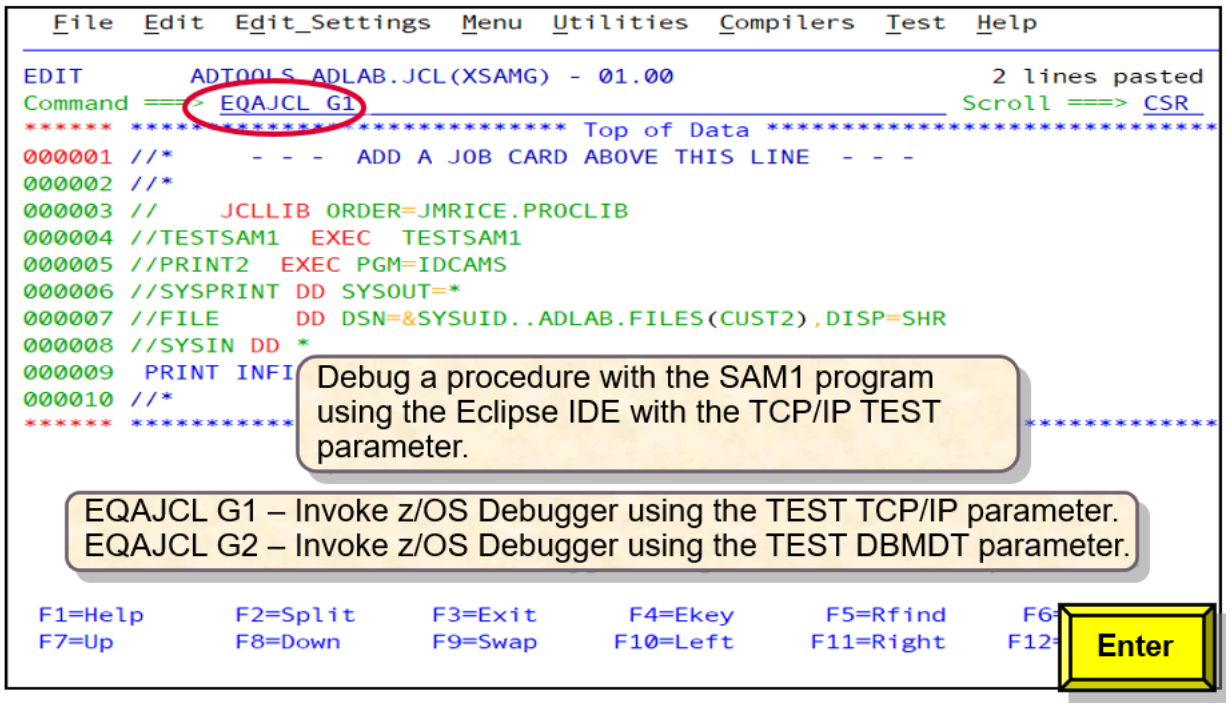
1. Start the Terminal Interface Manager.
2. Sign on to the Terminal Interface Manager with your user ID and password.
3. Submit the job.

If the Terminal Interface Manager does not start a debug session, verify that the job is not waiting for an initiator, or did not fail with a JCL error.

Debugging a Language Environment program using a remote debugger without Debug Manager

With the remote debugger in the Eclipse IDE, you can debug Enterprise COBOL, COBOL for MVS and VM, Enterprise PL/I, later versions of C/C++ and assembler. You can use IBM z/OS Debugger JCL Wizard to create JCL statements to debug a Language Environment program with the TCP/IP TEST parameter.

1. In ISPF Edit or Browse, enter **EQAJCL G1** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel, and request a debug session with the remote debugger. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **G1**.



2. Obtain your IP address from your workstation:

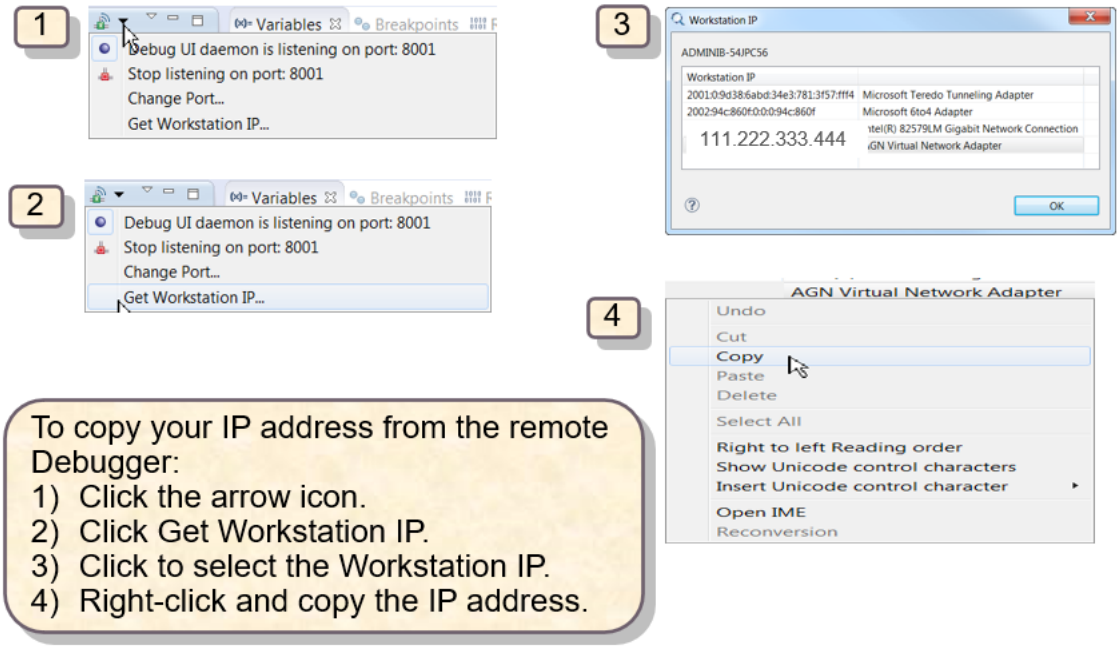
   a. Start the Eclipse IDE.

   b. Open the **Debug** perspective.

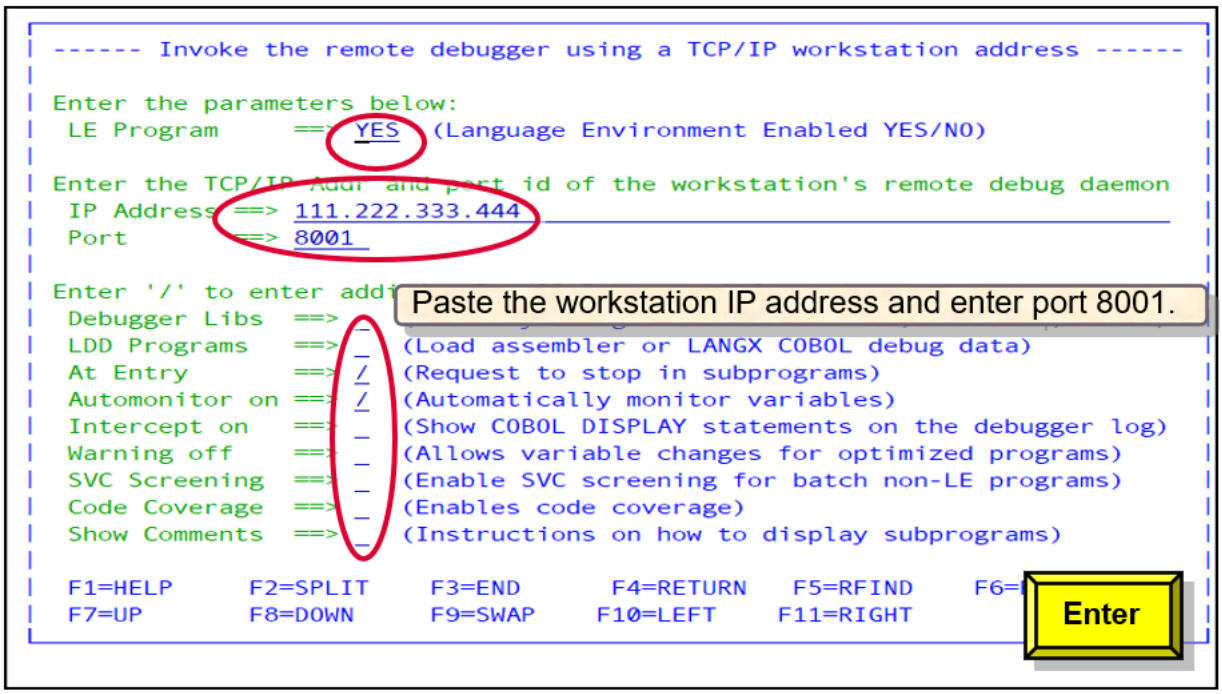
   c. Click the arrow icon.

   d. Select **Get Workstation IP** from the list. More than one IP address might be shown.

   e. Select the IP address associated with your workstation. Right-click the workstation IP address, and copy this address for use by the IBM z/OS Debugger JCL Wizard.



3. In the parameters panel, paste your IP address into the **IP address** field and enter the port number. The port number is generally 8001.

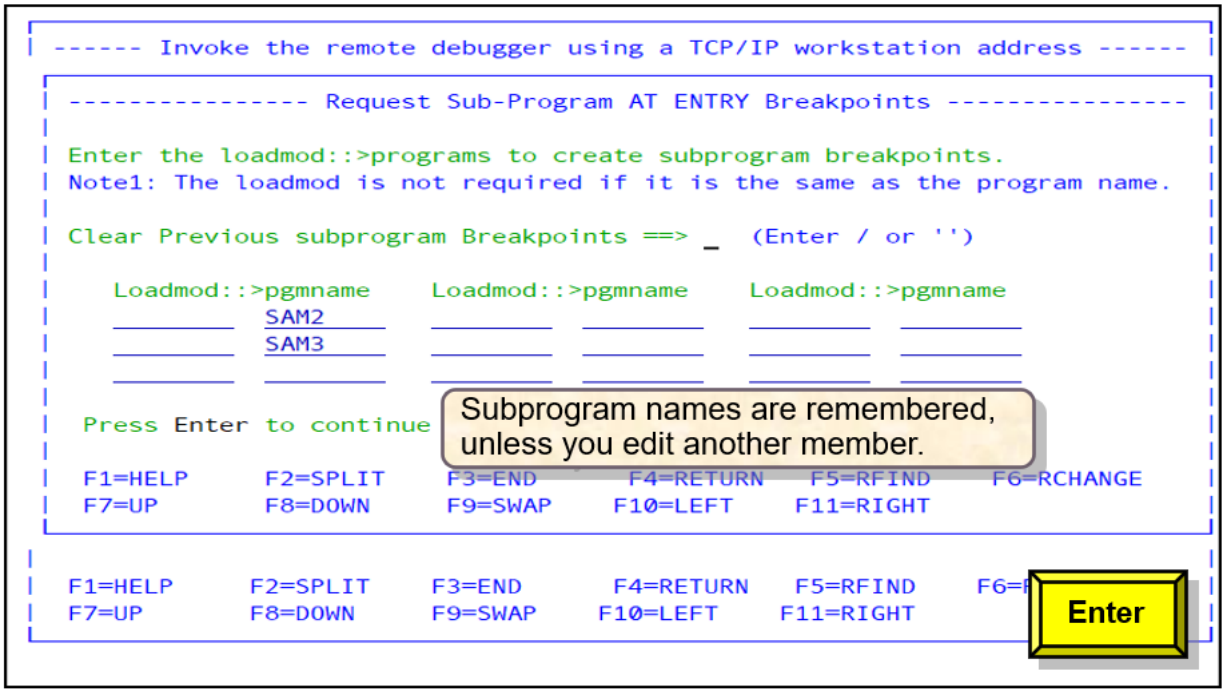


Type **YES** in the **LE Program** field because this program is Language Environment enabled.

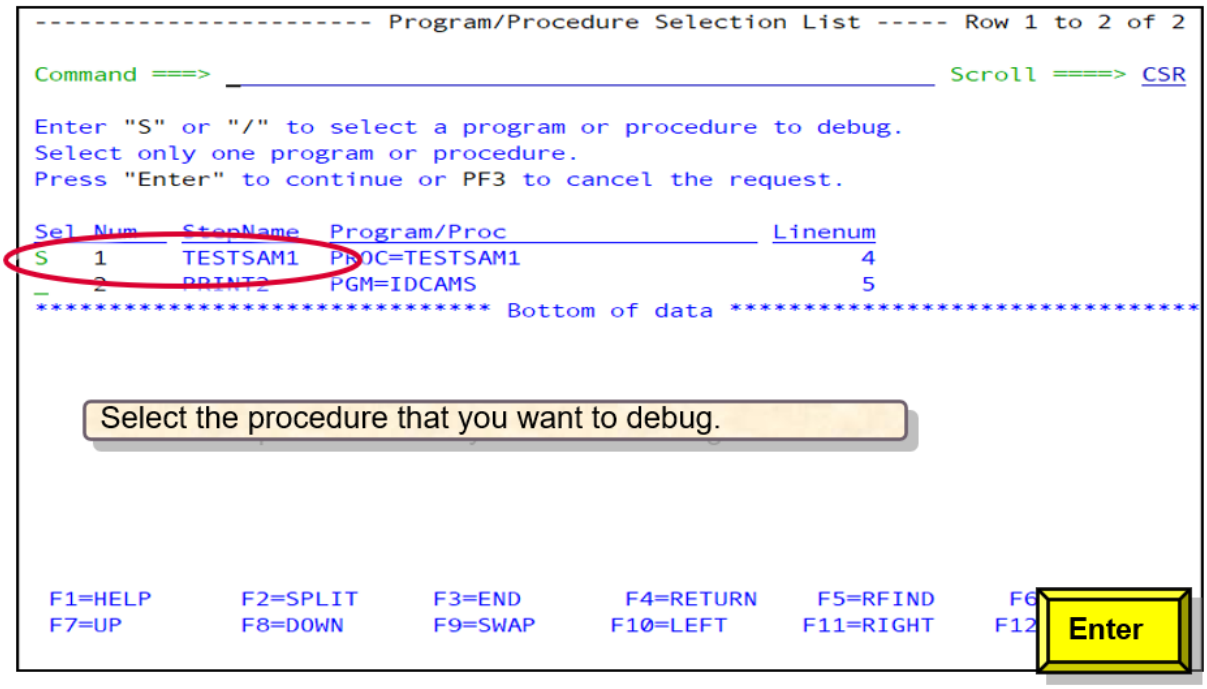
To select an optional parameter, specify a forward slash (/) in the field.

In this use case, **At Entry** and **Automonitor on** are selected.

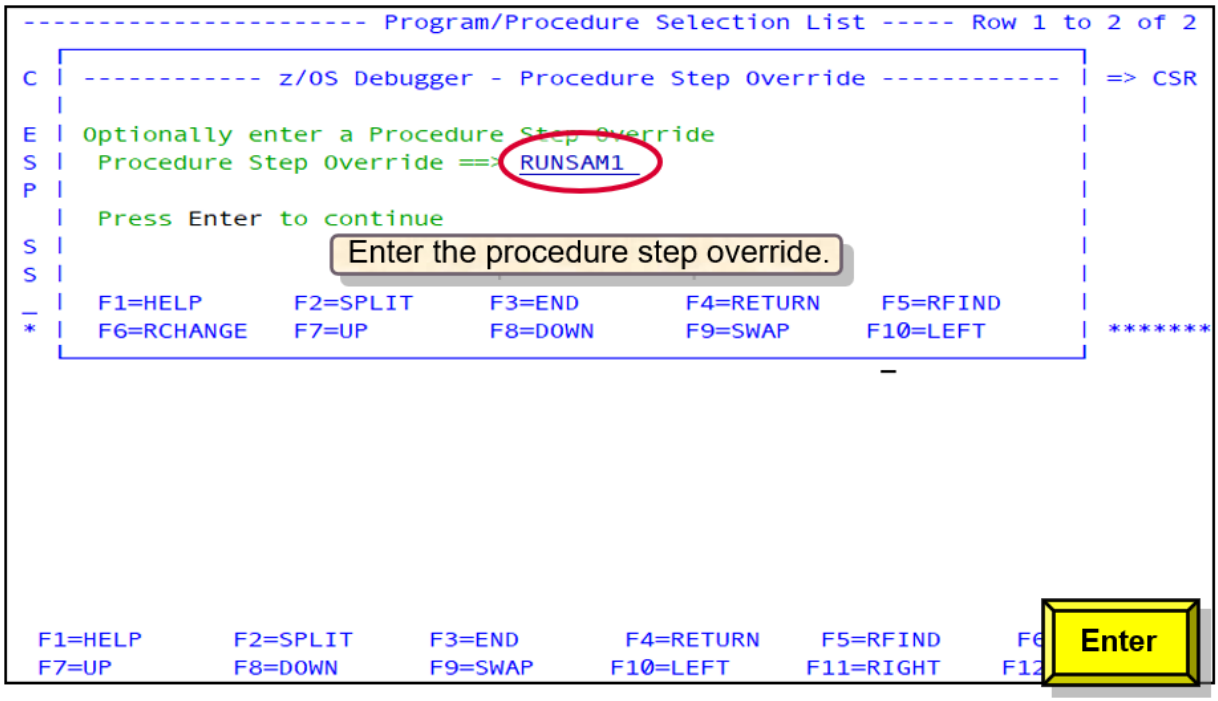
4. Since you chose to set AT ENTRY breakpoints for subprograms, the **Request AT ENTRY Sub-Program Breakpoints** panel is displayed. The subprograms SAM2 and SAM3 are dynamically called. The load module name of SAM2 and SAM3 is named SAM2 and SAM3 respectively. Therefore, only the program name is required.



5. In the **Program/Procedure Selection List** panel, the procedures and programs are listed. In this use case, the JCL points to a procedure and a program. Select the procedure that you want to debug.



6. In this use case, the JCL points to a procedure. If the After line command was chosen, you can enter the procedure step override. RUNSAM1 is entered in this use case.

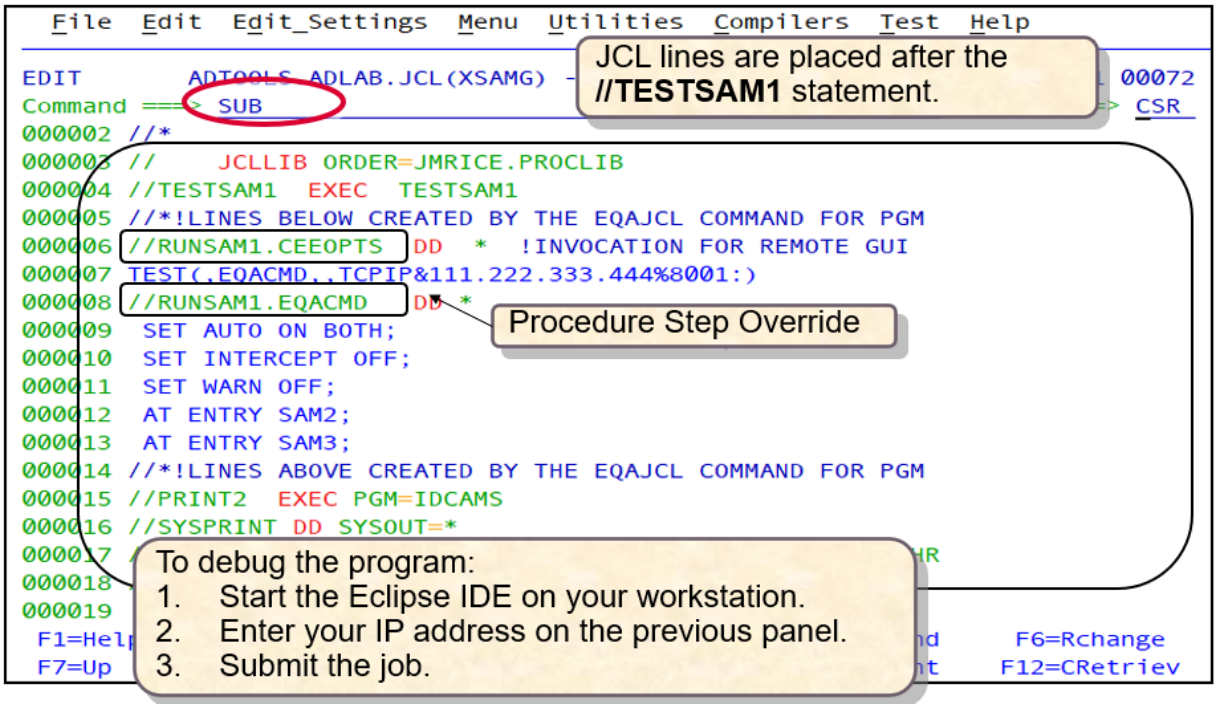


7. JCL statements are generated to invoke z/OS Debugger.

   The procedure TESTSAM1 contains a step RUNSAM1, which invokes the SAM1 program. Use the procedure step override to define EQACMD DD (and its contents) for the RUNSAM1 STEP.

   The CEEOPTS DD statement is generated with the parameter TCP/IP, indicating you want to debug using the remote debugger with the appropriate IP address and port number.

   The automonitor is turned on, AT ENTRY breakpoints are set, and instructions are provided on how to view subroutines before invocation.



What to do next

You can now start a debug session with the remote debugger:

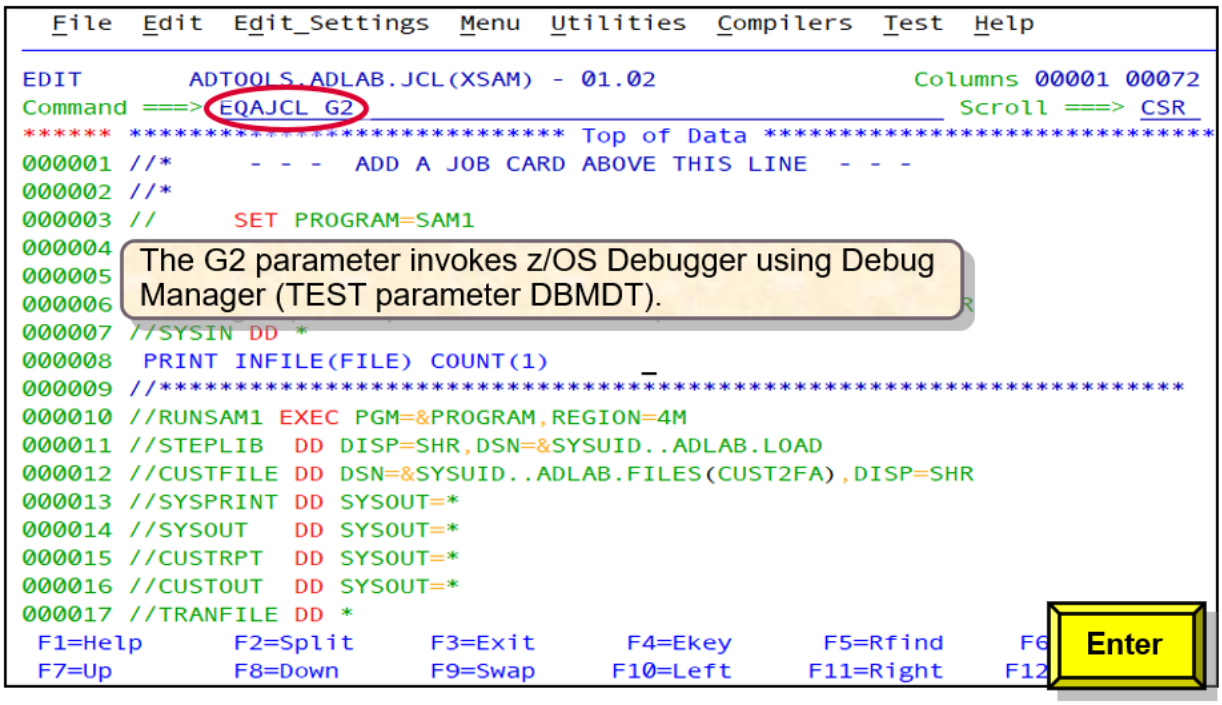
1. Start the remote debugger in the Eclipse IDE.
2. Enter your IP address.
3. Submit the job.

If the remote debugger does not depict the initiation of a debug session, verify that the job is not waiting for an initiator, or failed with a JCL error.

Debugging a Language Environment program using a remote debugger with Debug Manager

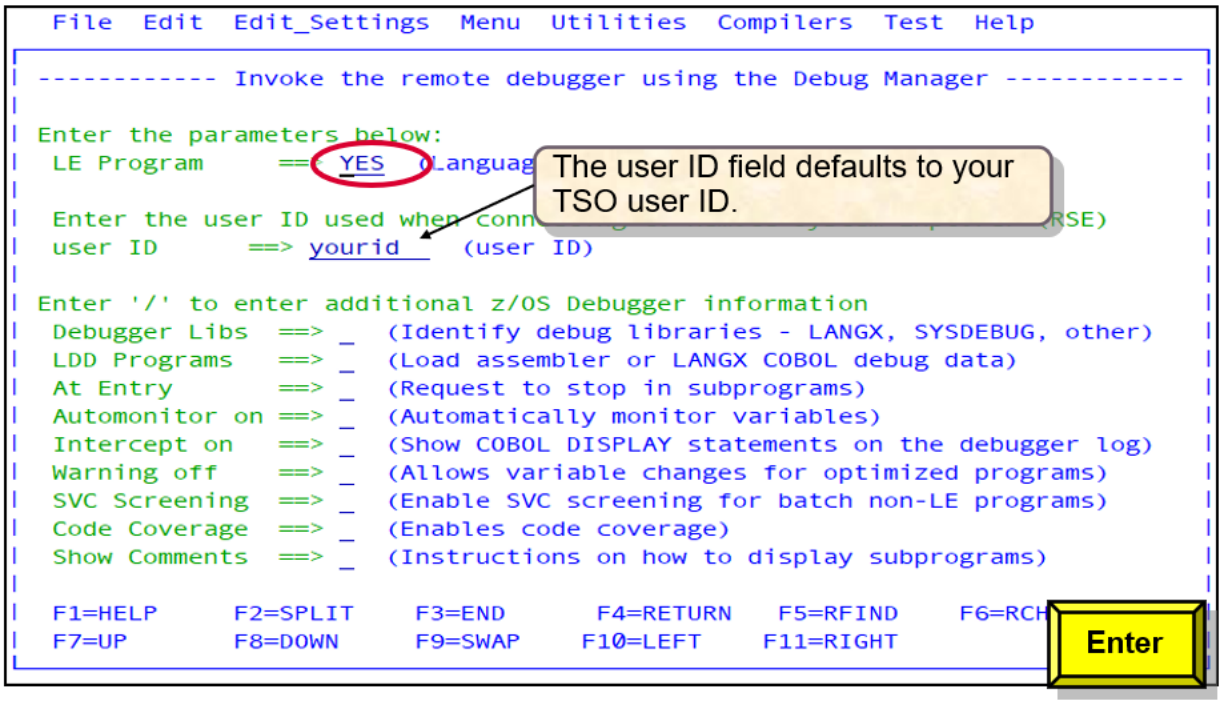
With the remote debugger in the Eclipse IDE, you can debug Enterprise COBOL, COBOL for MVS and VM, Enterprise PL/I, later versions of C/C++ and assembler. You can use IBM z/OS Debugger JCL Wizard to create JCL statements to debug a Language Environment program with Debug Manager.

1. In ISPF Edit or Browse, enter **EQAJCL G2** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel, and request a debug session with the remote debugger. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **G2**.

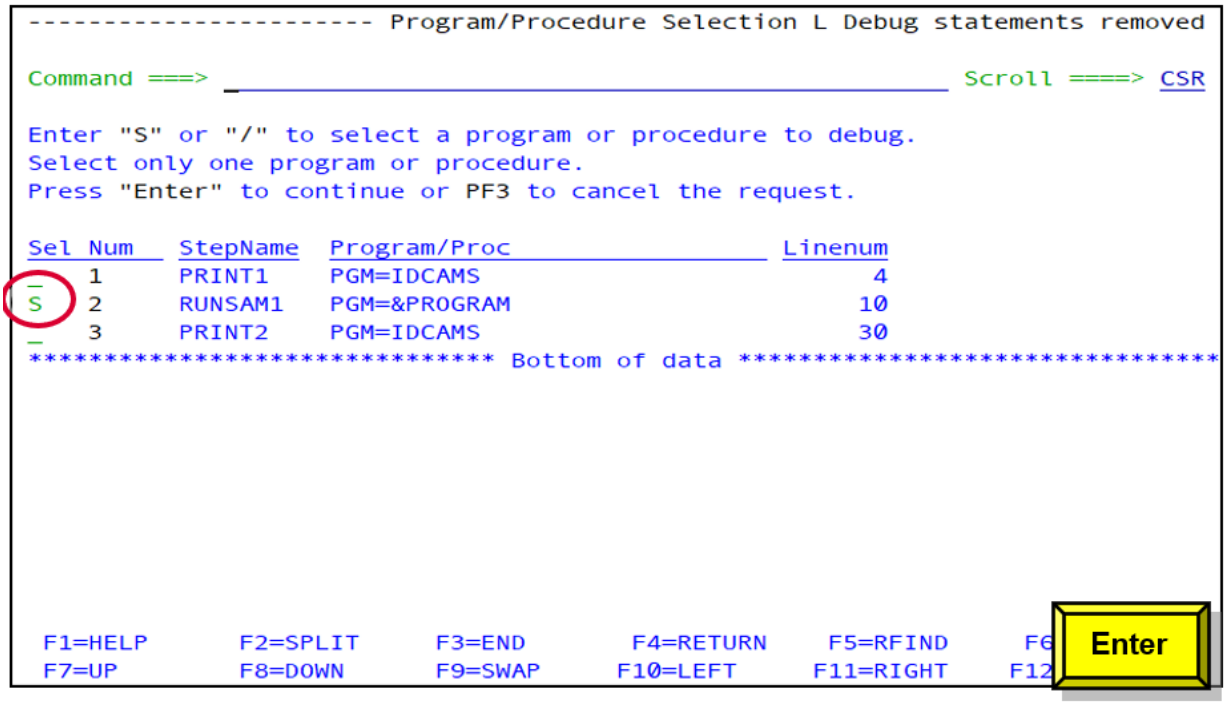


2. In the parameters panel, type **YES** in the **LE Program** field.

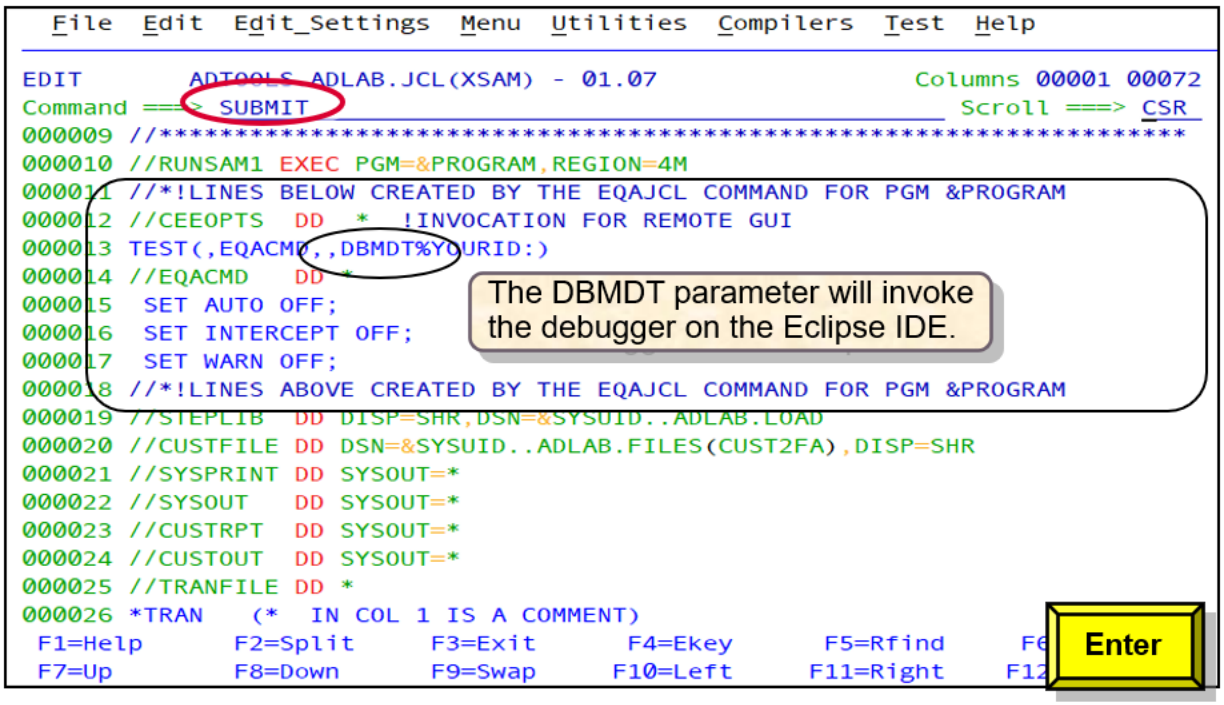
   To select an optional parameter, specify a forward slash (/) in the field.



3. In the **Program/Procedure Selection List** panel, choose the program that you want to debug. You can choose only one step.



4. JCL statements are generated to invoke z/OS Debugger.



What to do next

You can now start a debug session with the remote debugger:

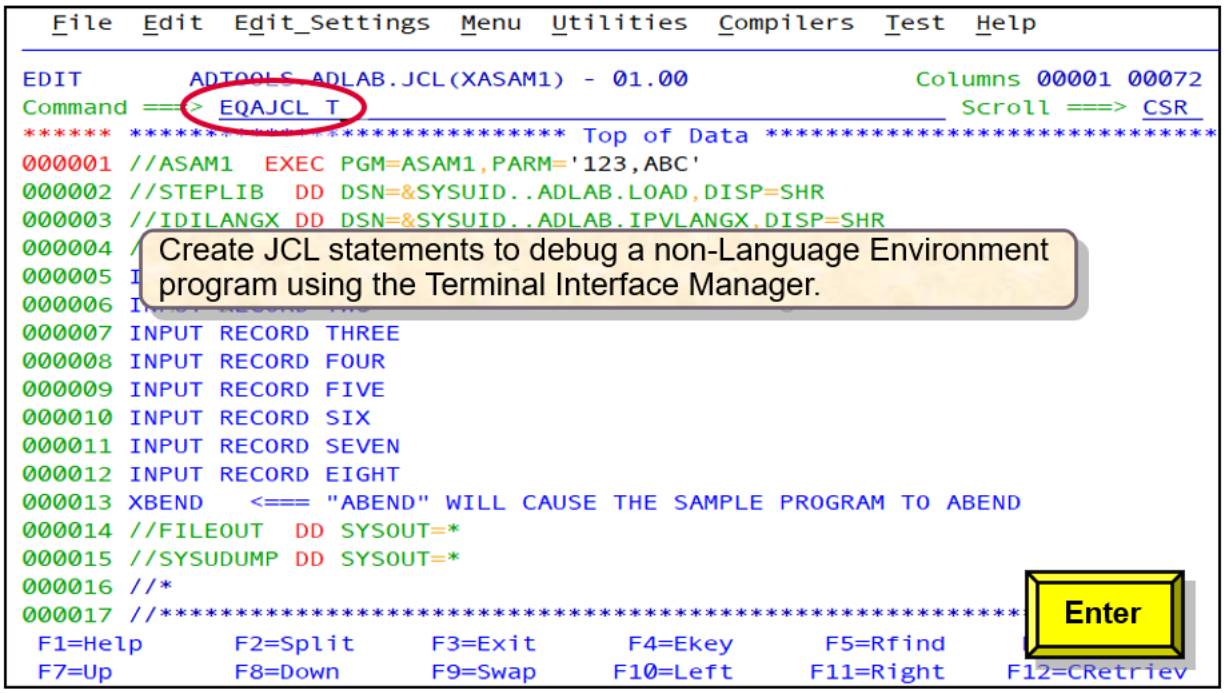
1. Start the remote debugger in the Eclipse IDE.
2. Connect to the Remote System Explorer (RSE).
3. Submit the job.

If the remote debugger does not depict the initiation of a debug session, verify that the job is not waiting for an initiator, or failed with a JCL error.

Debugging a non-Language Environment program using Terminal Interface Manager

You can use IBM z/OS Debugger JCL Wizard to create JCL statements to debug a non-Language Environment (non-LE) program by using the Terminal Interface Manager.

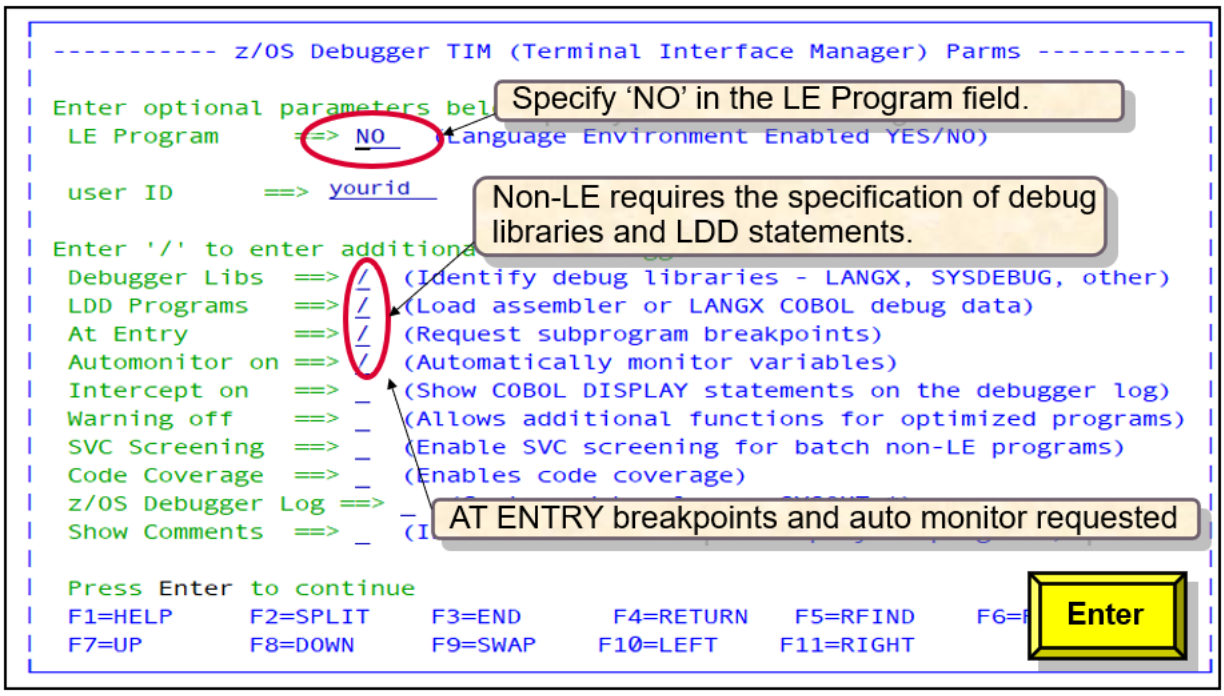
1. In ISPF Edit or Browse, enter **EQAJCL T** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel and invoke the IBM z/OS Debugger JCL Wizard for the Terminal Interface Manager. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **T**.



2. Enter **NO** in the **LE Program** field.

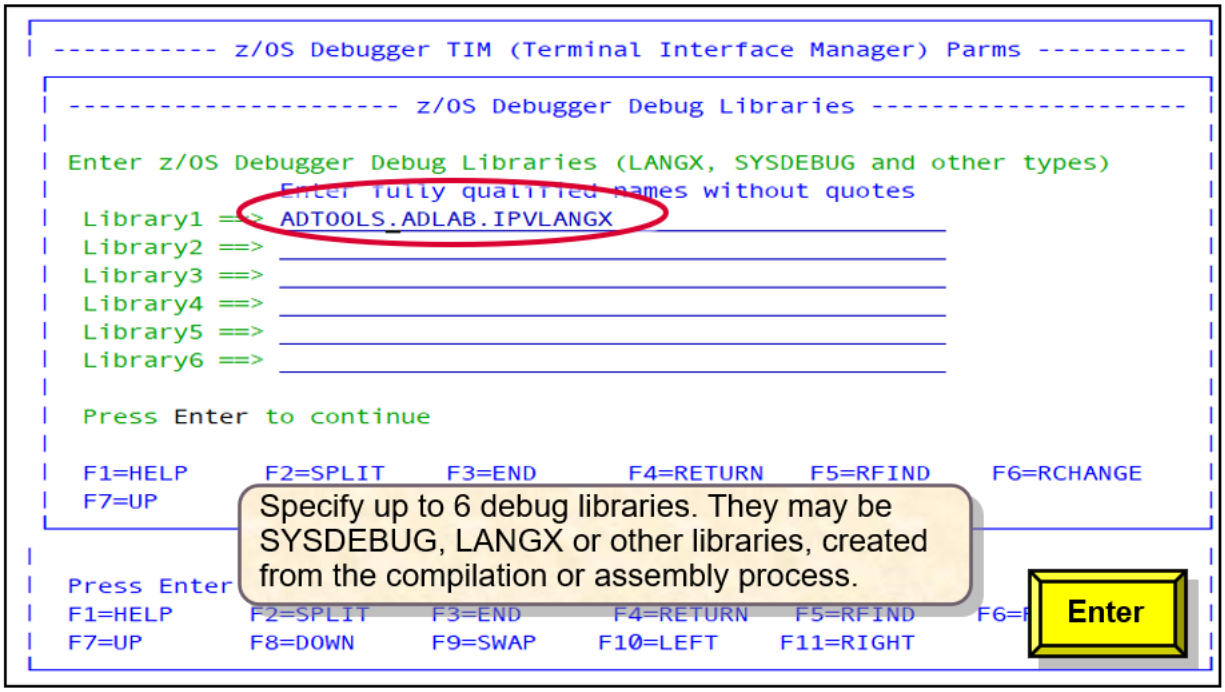
   To select an optional parameter, specify a forward slash (/) in the field.

   Non-Language Environment programs require debug libraries to identify where the side file information is located and also a Load Debug Data (LDD) command to load the program source. Specify a forward slash (/) in the **Debugger Libs** and **LDD Programs** fields. In this use case, **At Entry** and **Automonitor on** are also selected.

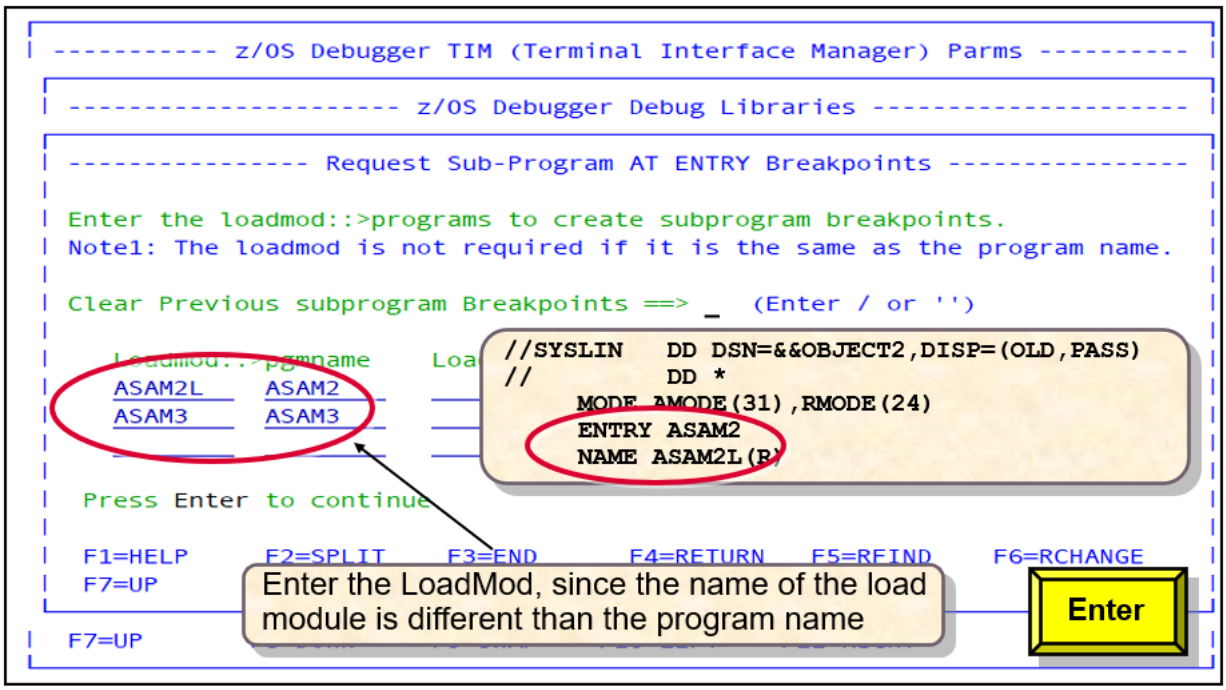


3. Since you chose **Debugger Libs**, the **z/OS Debugger Debug Libraries** panel is displayed. In this panel, you can identify up to six z/OS Debugger side file libraries. These side files are created during the compilation or assembly process.

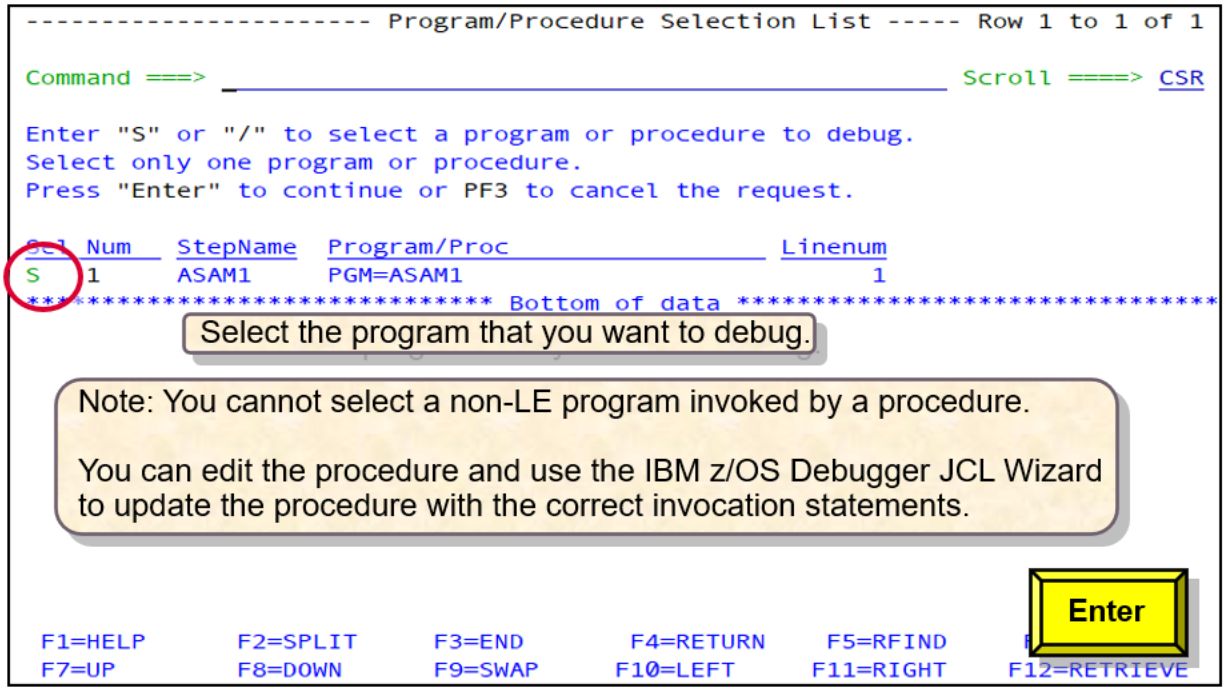
   You can add libraries of various types in this panel. For example, some languages use a LANGX file, others use SYSDEBUG or listing data sets. If you require more than six libraries, modify the JCL after the IBM z/OS Debugger JCL Wizard creates the appropriate statements.



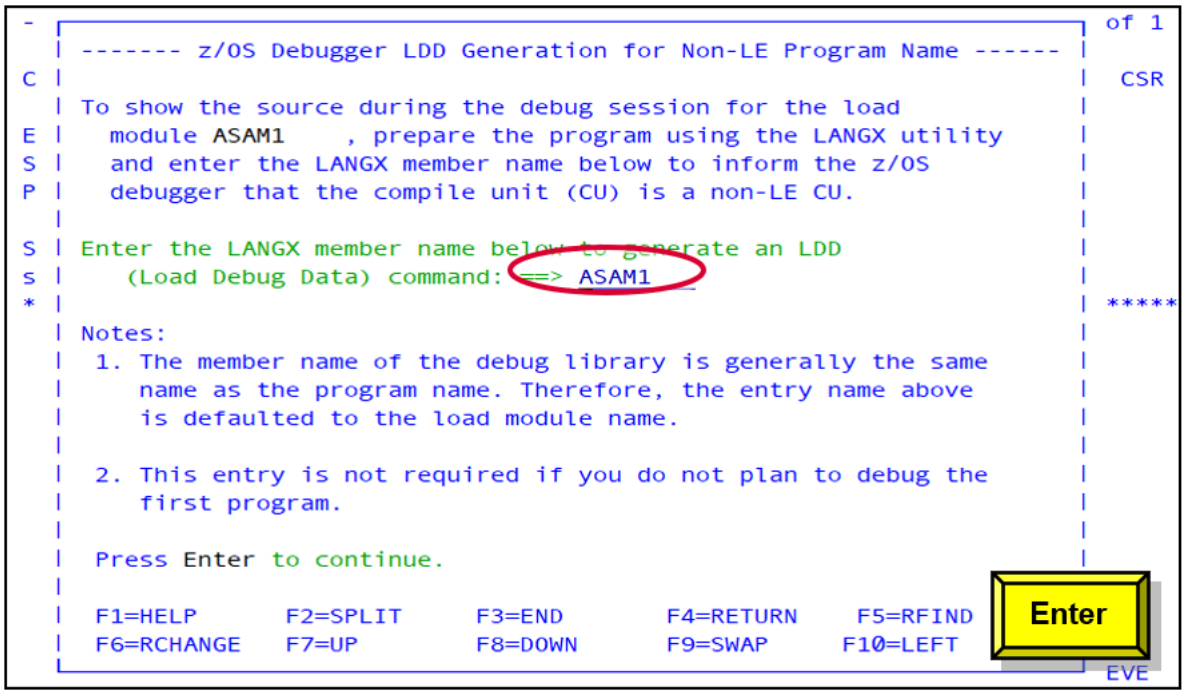
4. Since you chose to set AT ENTRY breakpoints for subprograms, the **Request AT ENTRY Sub-Program Breakpoints** panel is displayed. In this panel, enter one or more names of the z/OS Debugger side file data sets that you want to use. When you set breakpoints, generally only the program name is required. However, if the load module name is different than the program name, enter the load module name next to the program name. In this use case, the load module name of the ASAM2 program is ASAM2L.



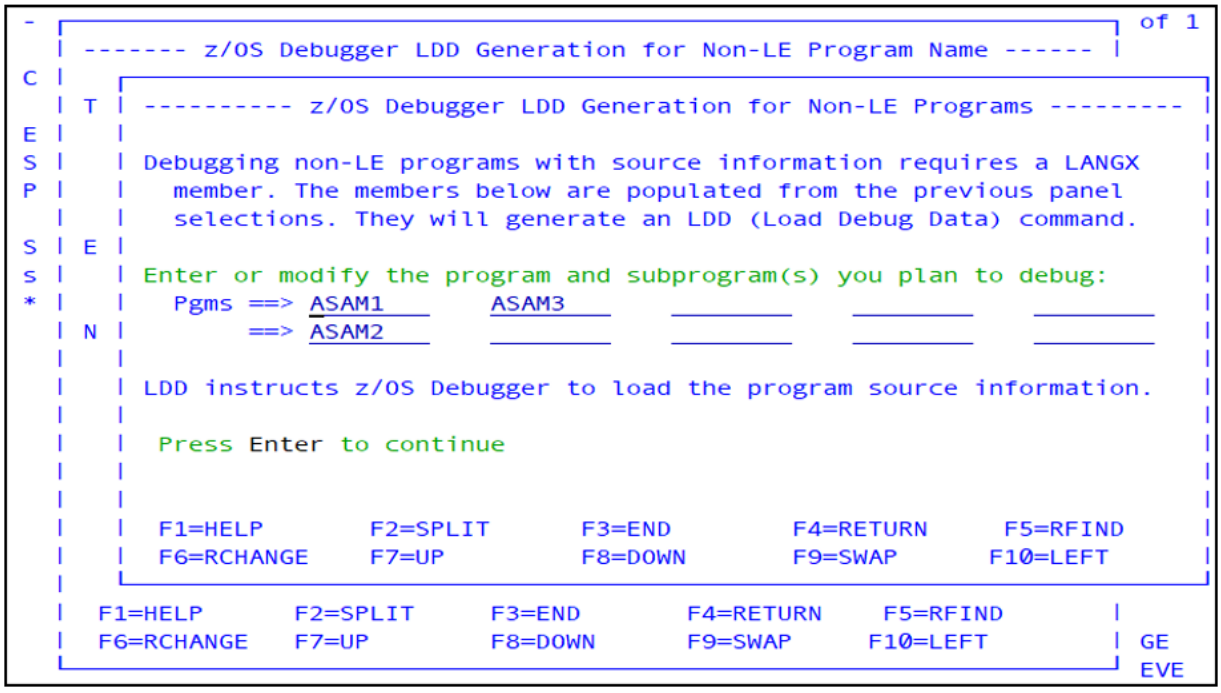
5. In the **Program/Procedure Selection List** panel, provide the program that you want to debug. The IBM z/OS Debugger JCL Wizard requires the program name to be explicitly identified for non-Language Environment programs. Enter the name of the non-Language Environment program shown on the EXEC PGM statement.



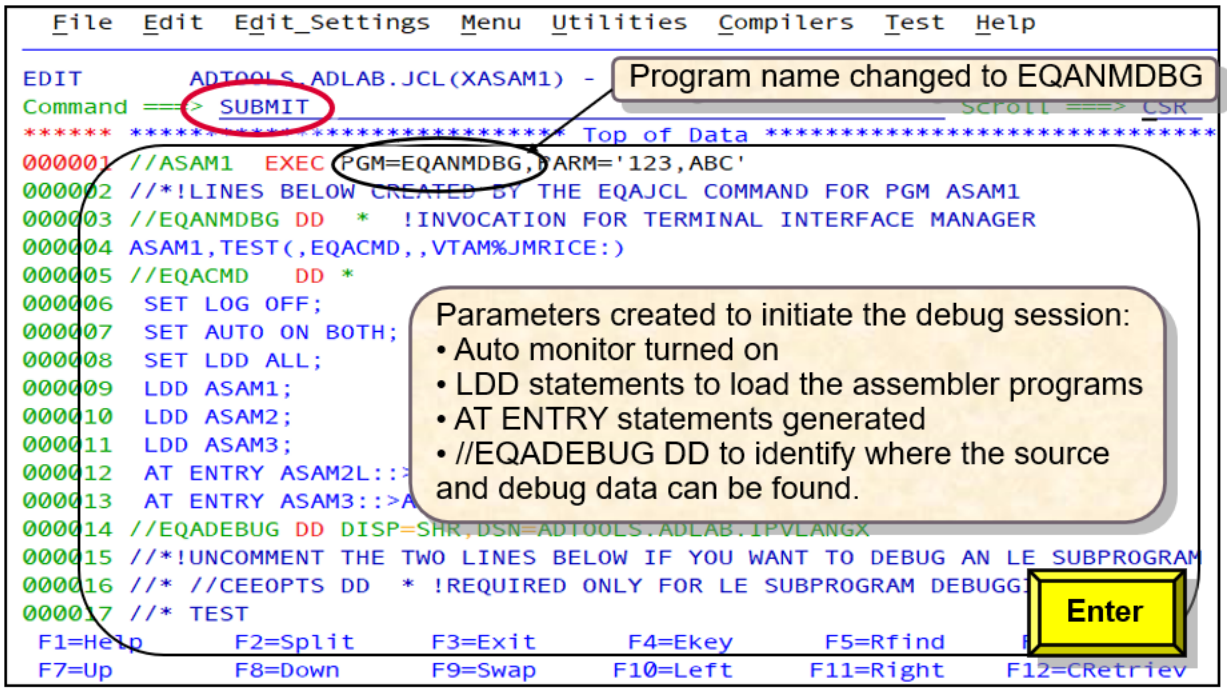
6. Since you chose **LDD Programs**, the **z/OS Debugger LDD generation for Non-LE Program Name** panel is displayed. The IBM z/OS Debugger JCL Wizard requires the member name of the LANGX library if you want to debug the first program with source.



7. The **z/OS Debugger LDD generation for Non-LE Programs** panel is defaulted to the LDD name provided in the **z/OS Debugger LDD generation for Non-LE Program Name** panel and the subprograms provided in the **Request AT ENTRY Sub-Program Breakpoints** panel. You can override or add more entries to this panel to depict the LANGX members that you want to use in the debug session.

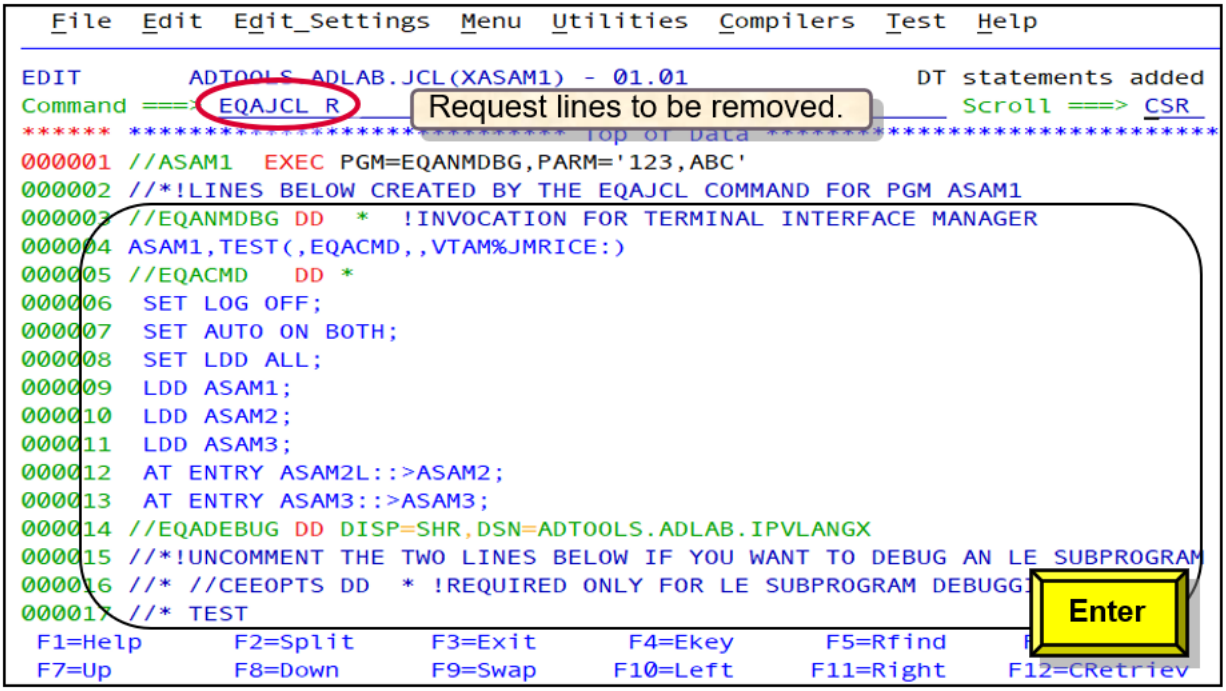


8. JCL statements are generated to invoke z/OS Debugger.

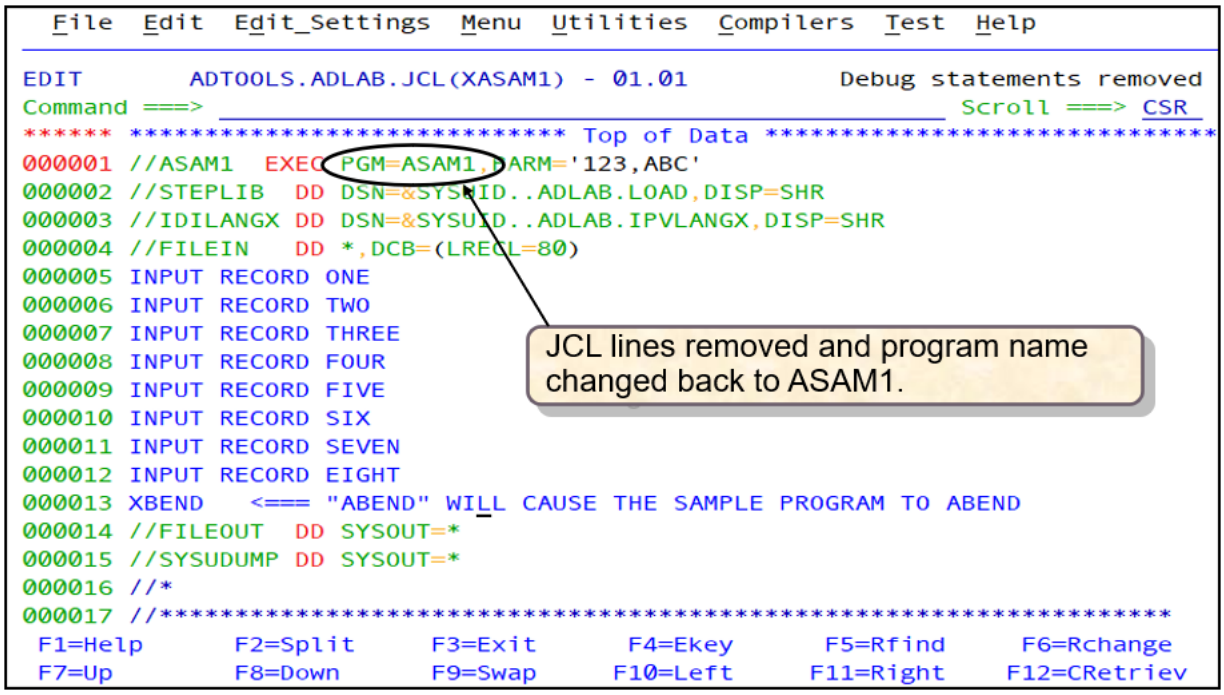


- The program name on line 3 was changed from ASAM1M to EQANMDBG. This program initiates the debug session and debug ASAM1M, which is the first program to be invoked.
- The VTAM%JMRICE requests z/OS Debugger to invoke the Terminal Interface Manager. This information is passed to the EQANMDBG DD statement.
- LDD statements are generated for programs ASAM1M, ASAM2, and ASAM3.
- Breakpoints are set for ASAM2 and ASAM3, using the load modules ASAM2L and ASAM3L respectively.
- The EQADEBUG DD statement defines the side files, where the program source and debug data can be found.

You can remove the JCL statements by using **EQAJCL R** as described in "Removing IBM z/OS Debugger JCL Wizard statements" on page 459.



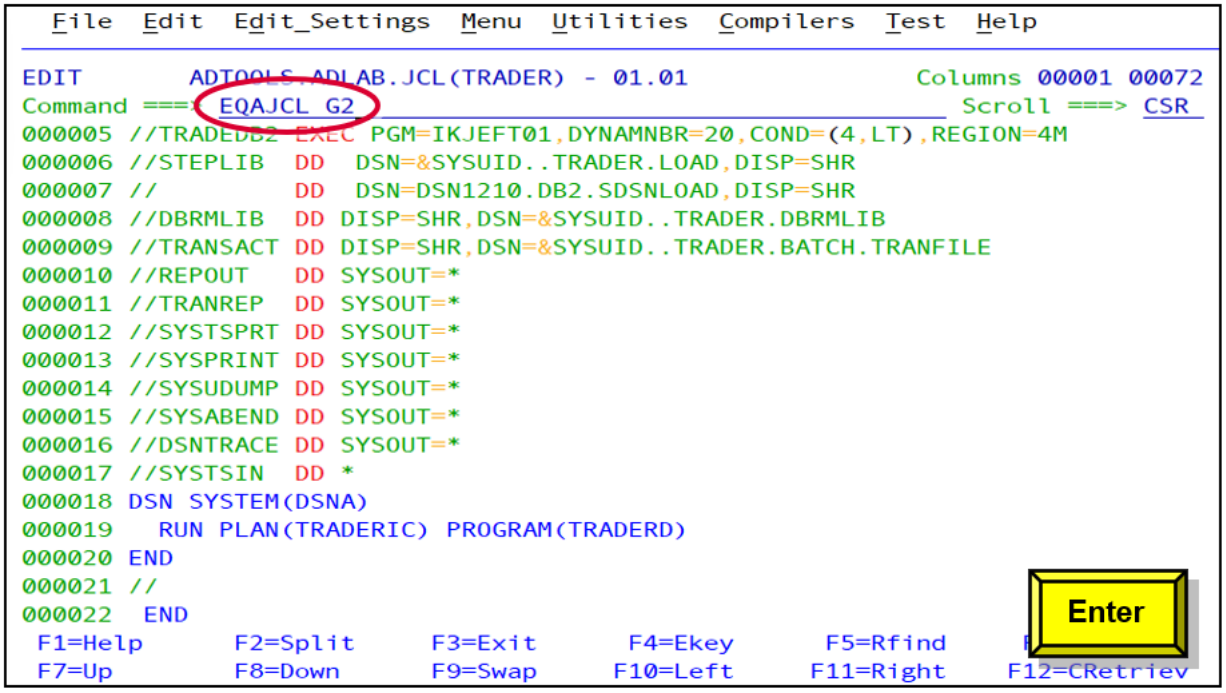
The PGM=EQANMDBG statement is modified back to the original program name, ASAM1M.



Debugging a Language Environment Db2 program using a remote debugger with Debug Manager

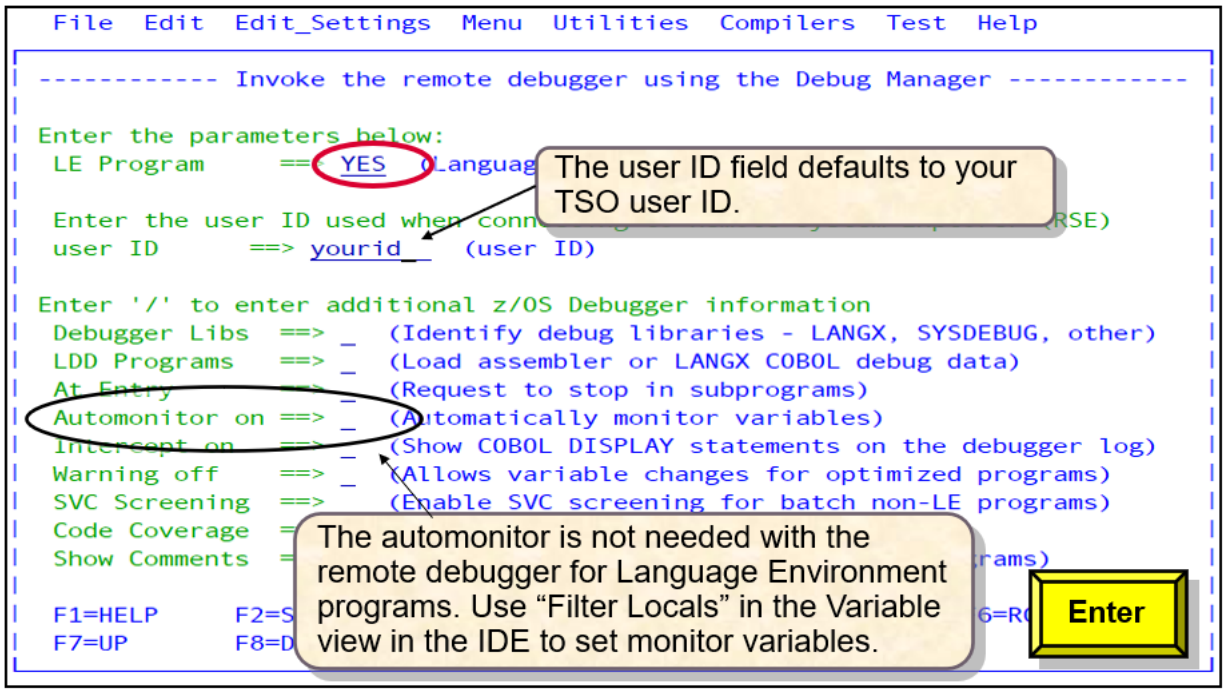
You can generate z/OS Debugger JCL statements to debug a Db2 batch program by using the remote debugger in the Eclipse IDE with Debug Manager.

1. In ISPF Edit or Browse, enter **EQAJCL G2** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel, and request a debug session with the remote debugger. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **G2**.



2. In the parameters panel, type **YES** in the **LE Program** field.

To select an optional parameter, specify a forward slash (/) in the field.



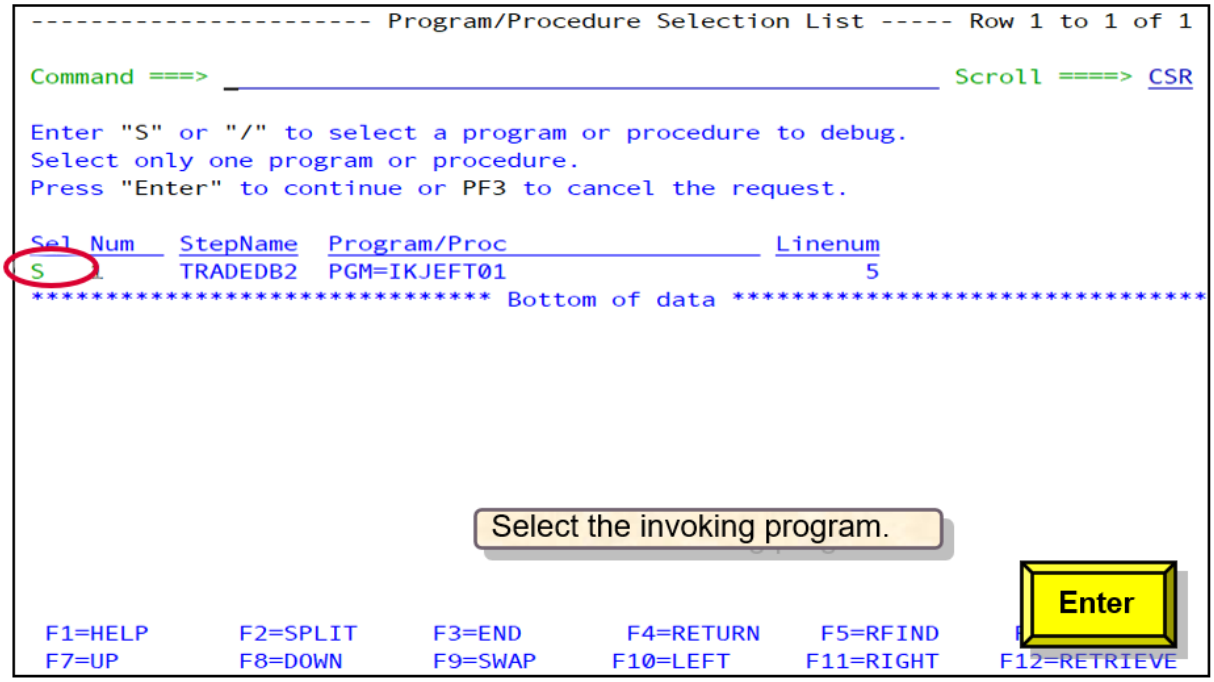
The **Automonitor on** is not needed when you debug Enterprise COBOL or PL/I programs. In the Eclipse IDE, you can monitor variables in the **Variables** view:

a. Right-click in the **Variables** view.

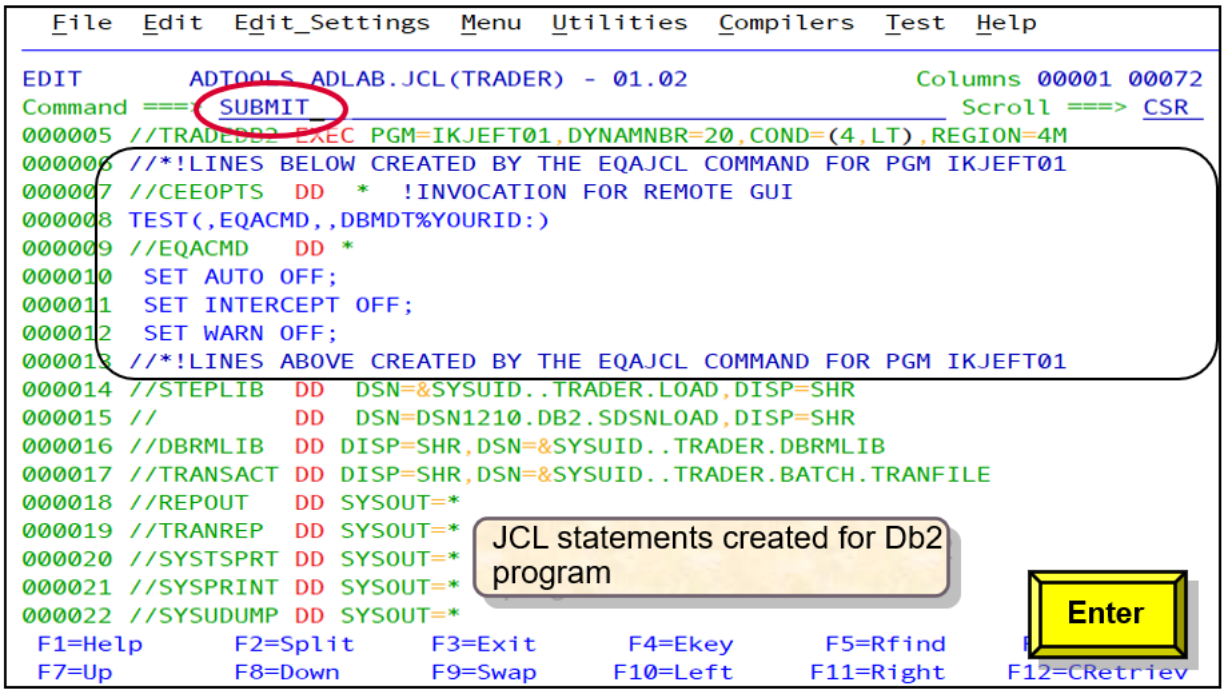
b. Select **Filter Locals**.

c. Select **Automonitor Current** or **Automonitor Previous**.

3. In the **Program/Procedure Selection List** panel, choose the program that you want to debug. Select the correct step name or procedure name.



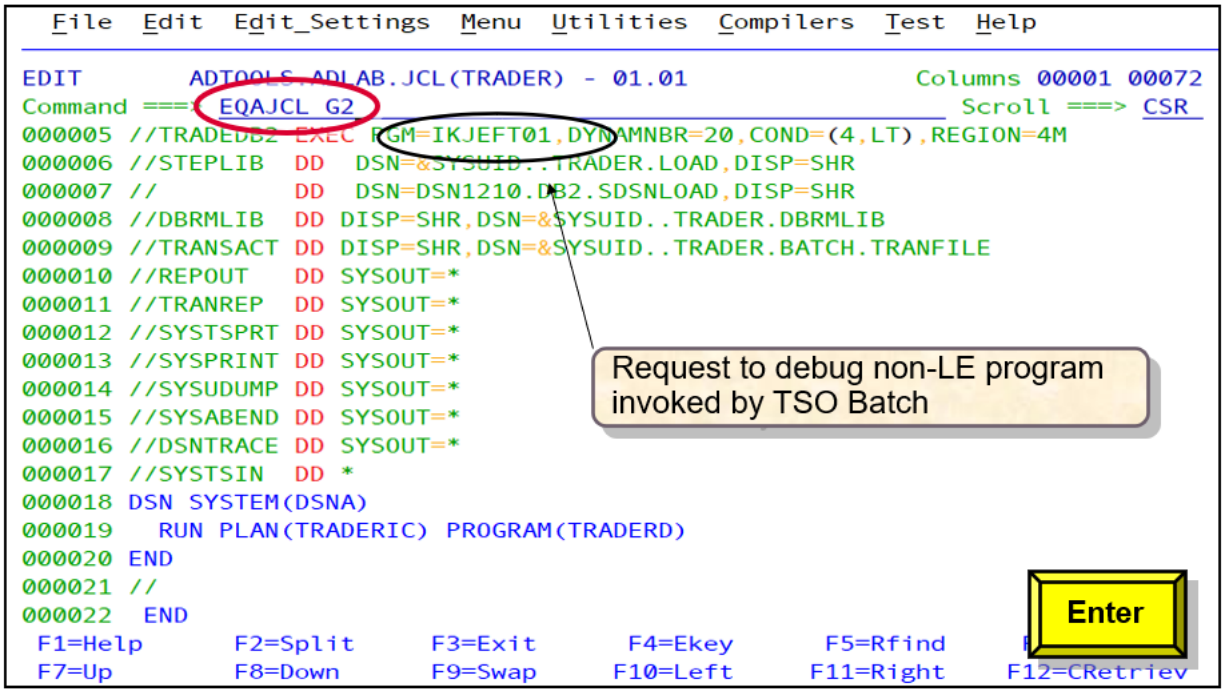
4. JCL statements are generated to invoke z/OS Debugger for a batch Db2 program. This process is identical to a non-Db2 program invocation.



Debugging a non-Language Environment Db2 program using a remote debugger with Debug Manager

You can generate z/OS Debugger JCL statements to debug a non-Language Environment Db2 batch program by using the remote debugger in the Eclipse IDE with Debug Manager.

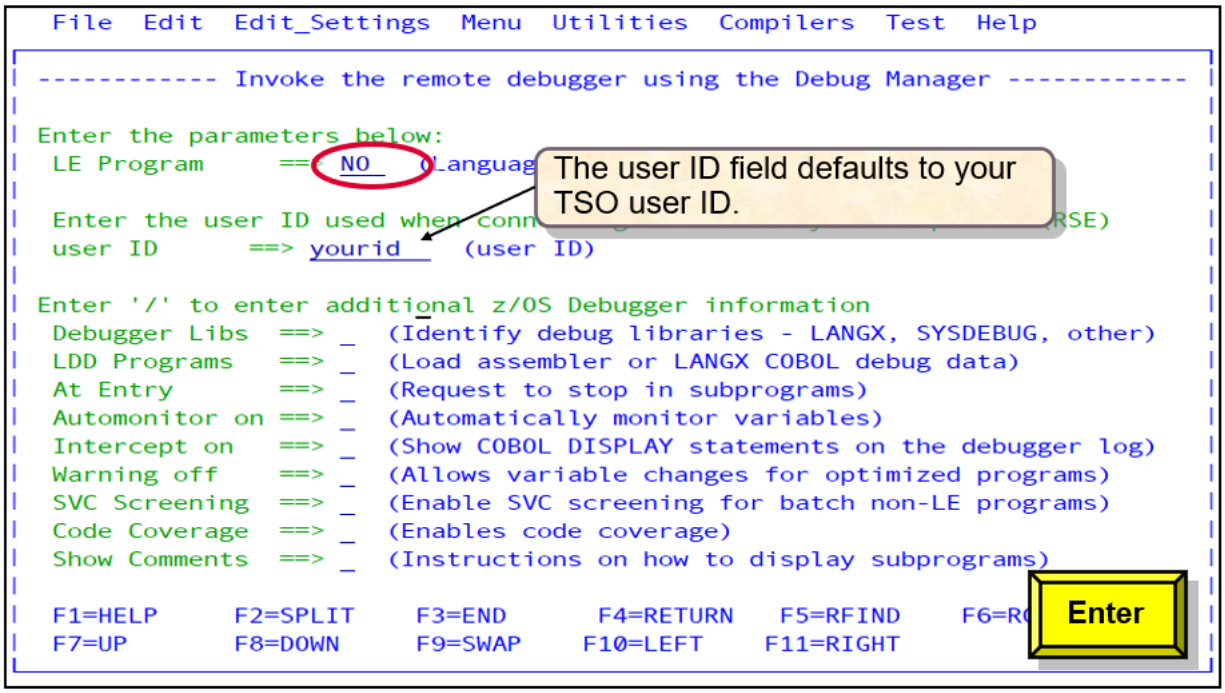
1. In ISPF Edit or Browse, enter **EQAJCL G2** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel, and request a debug session with the remote debugger. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **G2**.



2. Type **NO** in the **LE Program** field.

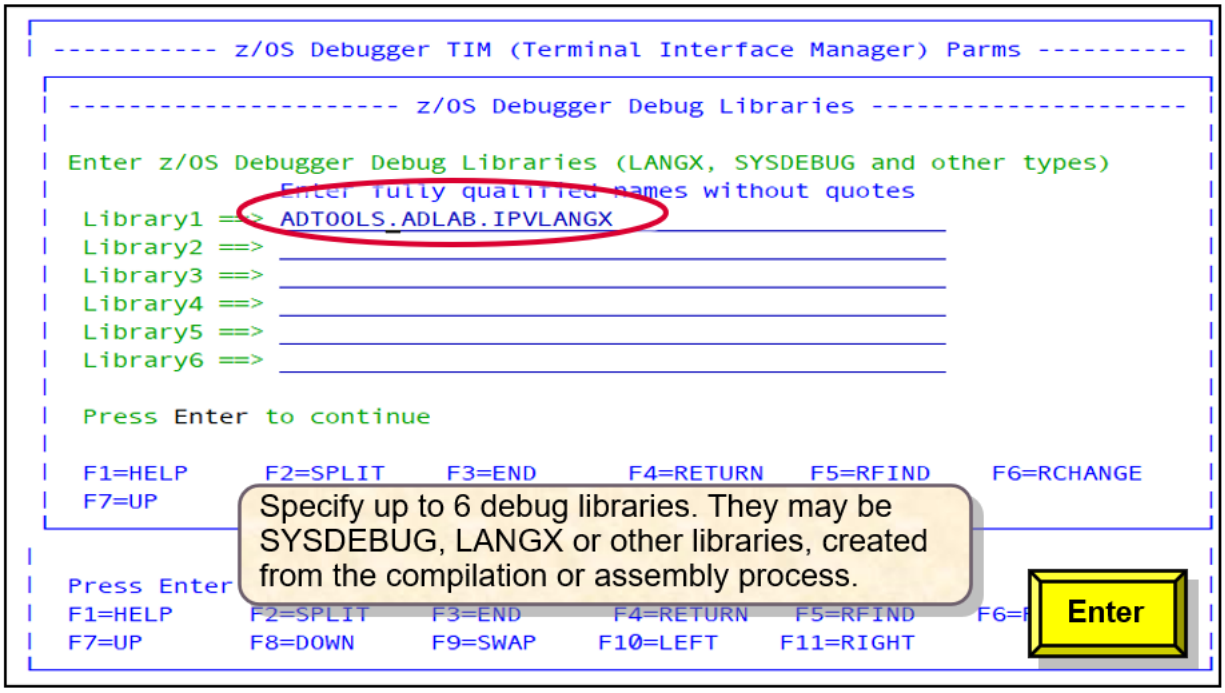
   To select an optional parameter, specify a forward slash (/) in the field.

Non-Language Environment programs requires debug libraries to identify where the side file information is located and also a Load Debug Data (LDD) command to load the program source. Specify a forward slash (/) in the **Debugger Libs** and **LDD Programs** fields. You can also select **Automonitor on** if you want to shows variables as you step through lines of code.

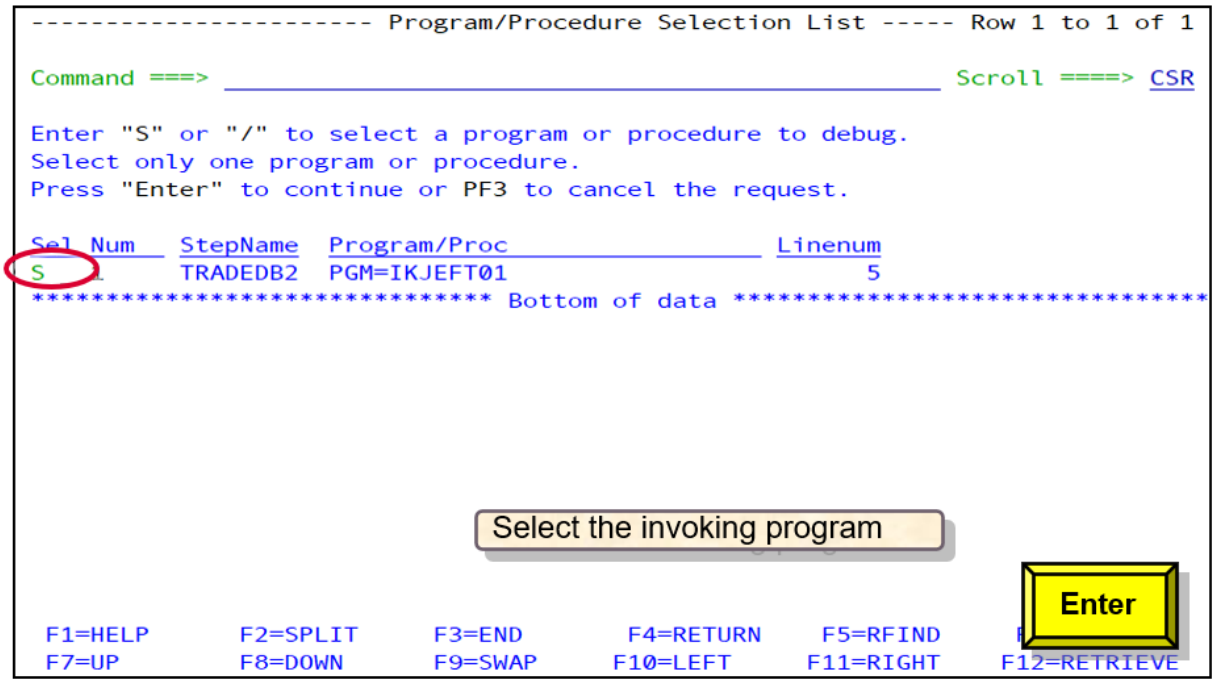


3. Since you chose **Debugger Libs**, the **z/OS Debugger Debug Libraries** panel is displayed. In this panel, you can identify up to six z/OS Debugger side file libraries. These side files are created during the compilation or assembly process.

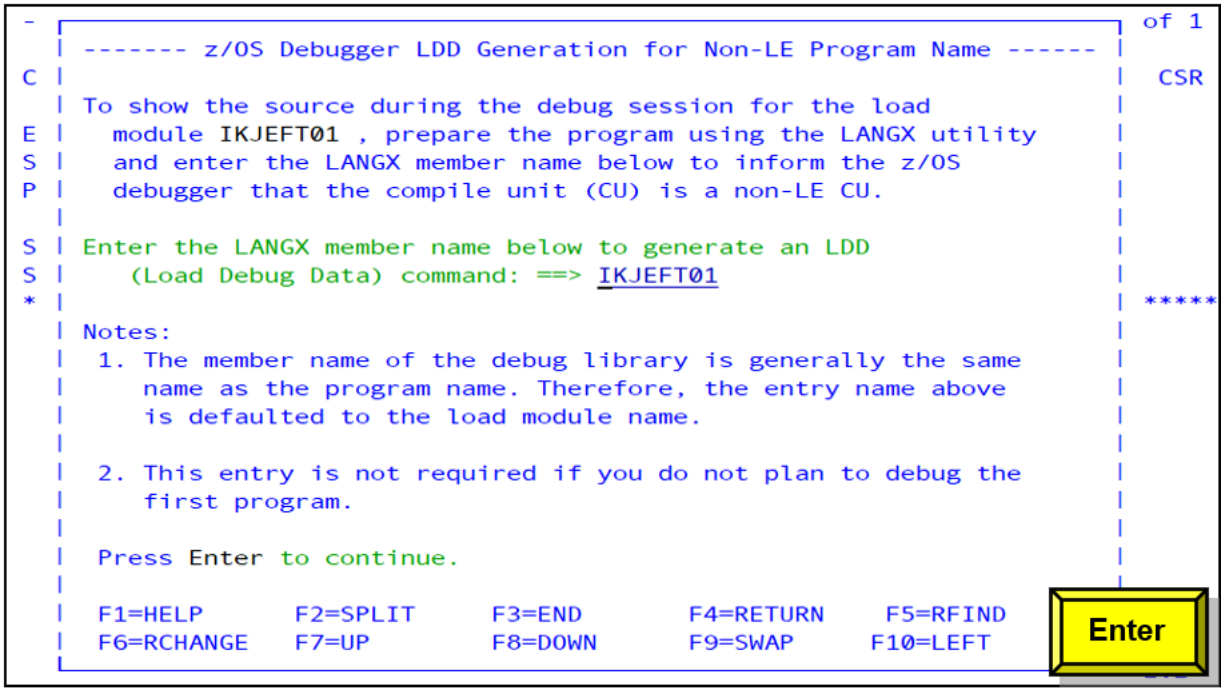
   You can add libraries of various types in this panel. For example, some languages use a LANGX file, others use SYSDEBUG or listing data sets. If you require more than six libraries, modify the JCL after the IBM z/OS Debugger JCL Wizard creates the appropriate statements.



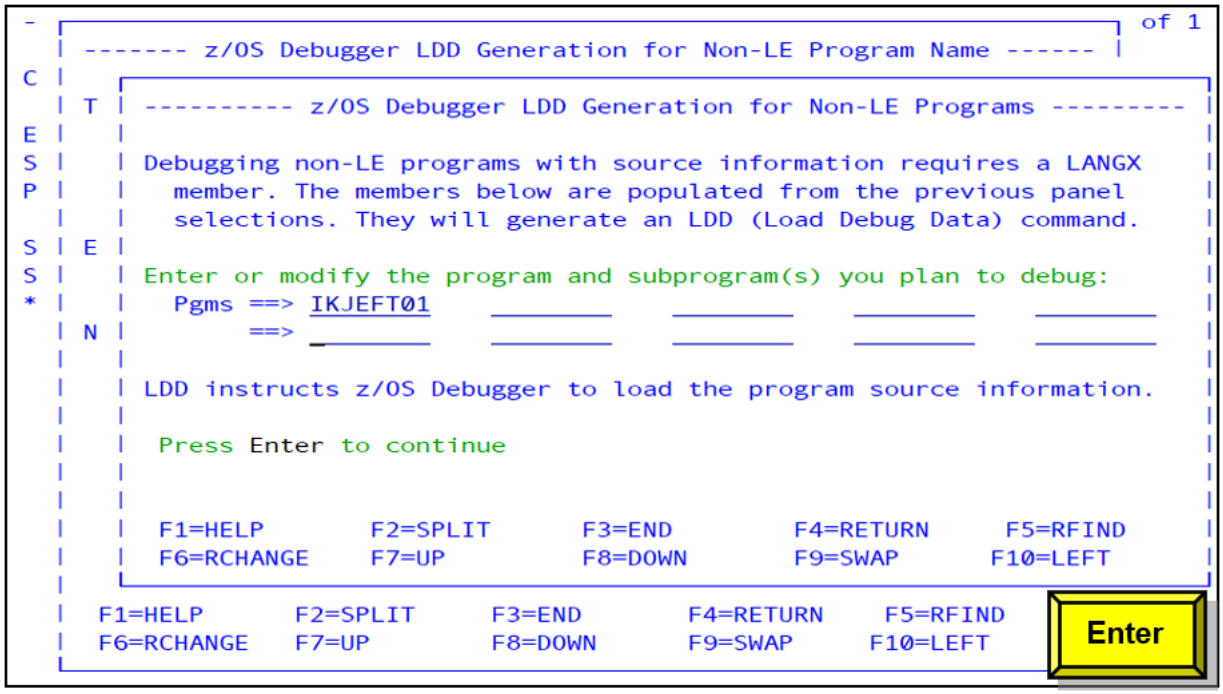
4. In the **Program/Procedure Selection List** panel, select the correct step name or procedure name.



5. Since you chose **LDD Programs**, the **z/OS Debugger LDD Generation for Non-LE Program Name** panel is displayed. The IBM z/OS Debugger JCL Wizard requires the member name of the LANGX library if you want to debug the first program with source.

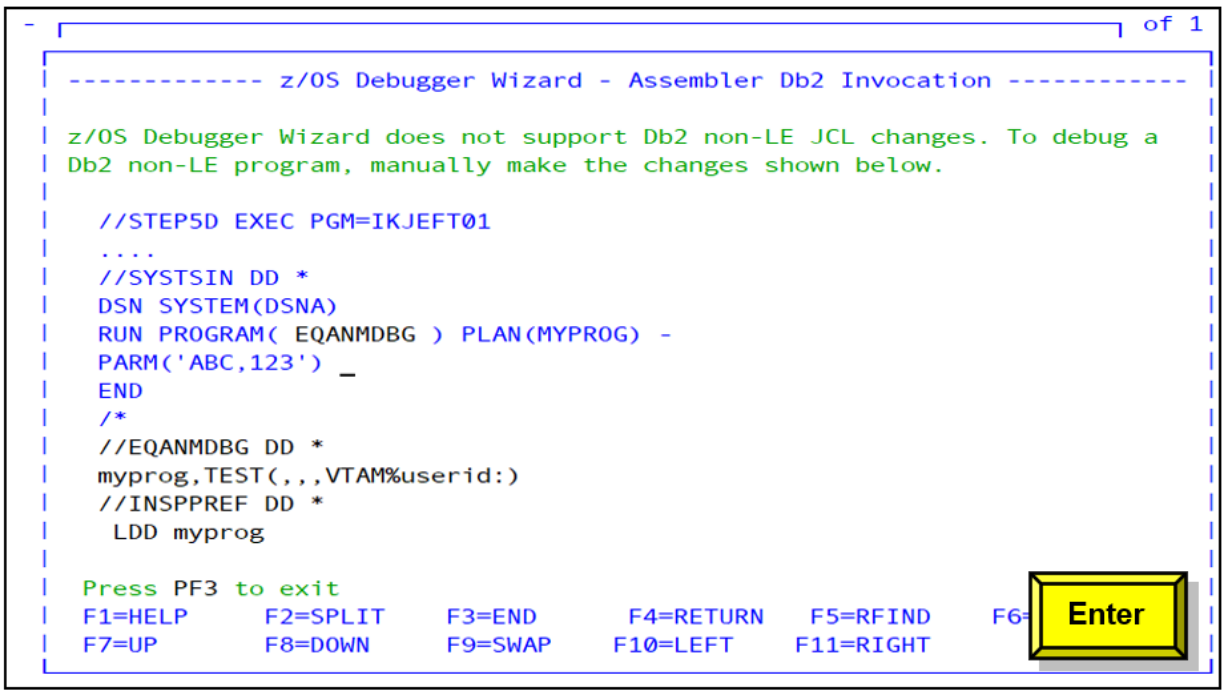


6. The **z/OS Debugger LDD generation for Non-LE Programs** panel is defaulted to the LDD name provided in the **z/OS Debugger LDD generation for Non-LE Program Name** panel. You can override or add more entries to this panel to depict the LANGX members that you want to use in the debug session.



7. Non-Language Environment Db2 program invocation requires the program name in the SYSTSIN statements to be changed from TRADERD to EQANMDBG. In addition, the DD name EQANMDBG is required, followed by the program name TRADERD, and the appropriate TEST parameters.

   Due to the complexity of locating and updating the SYSTSIN statements, this scenario must be done manually. Follow the example in the screen capture to create appropriate statements.



8. JCL statements are generated to invoke z/OS Debugger for a batch Db2 program. This process is identical to a Db2 program invocation.

Starting z/OS Debugger Code Coverage

You can use IBM z/OS Debugger JCL Wizard to create JCL statements to generate code coverage data.

Code coverage aggregates statement execution information from multiple executions of a program. This information can be used to depict any statements that were not tested. This function is limited to Enterprise COBOL programs compiled with TEST(SEPARATE), Enterprise PL/I programs compiled with TEST(SEPARATE), and z/OS XL C programs compiled with DEBUG(FORMAT(DWARF)). Your programs must be compiled with Enterprise COBOL or PL/I with the appropriate options to create a separate SYSDEBUG file.

You can invoke z/OS Debugger Code Coverage in one of the following ways:

- Without a debug session.
- With a debug session on the 3270 interface using the code coverage indicator.

The code coverage files are identified in one of the following ways:

- A customized z/OS Debugger EQAOPTS module to identify the z/OS Debugger Code Coverage files is in the load module search path

  or

- The installer sets the variable CODE_COVERAGE_SETUP = YES in the EQAJCL exec, which generates the following statements:

```
//EQAOPTS DD *
 EQAXOPT  CCPROGSELECTDSN,'&&USERID.DBGTOOL.CCPRGSEL'
 EQAXOPT  CCOUTPUTDSN,'&&USERID.DBGTOOL.CCOUTPUT'
 EQAXOPT  CCOUTPUTDSNALLOC,'MGMTCLAS(STANDARD)                          +
               STORCLAS(DEFAULT) LRECL(255) BLKSIZE(0) RECFM(V,B)       +
               DSORG(PS) SPACE(2,2) CYL'
 EQAXOPT END
```

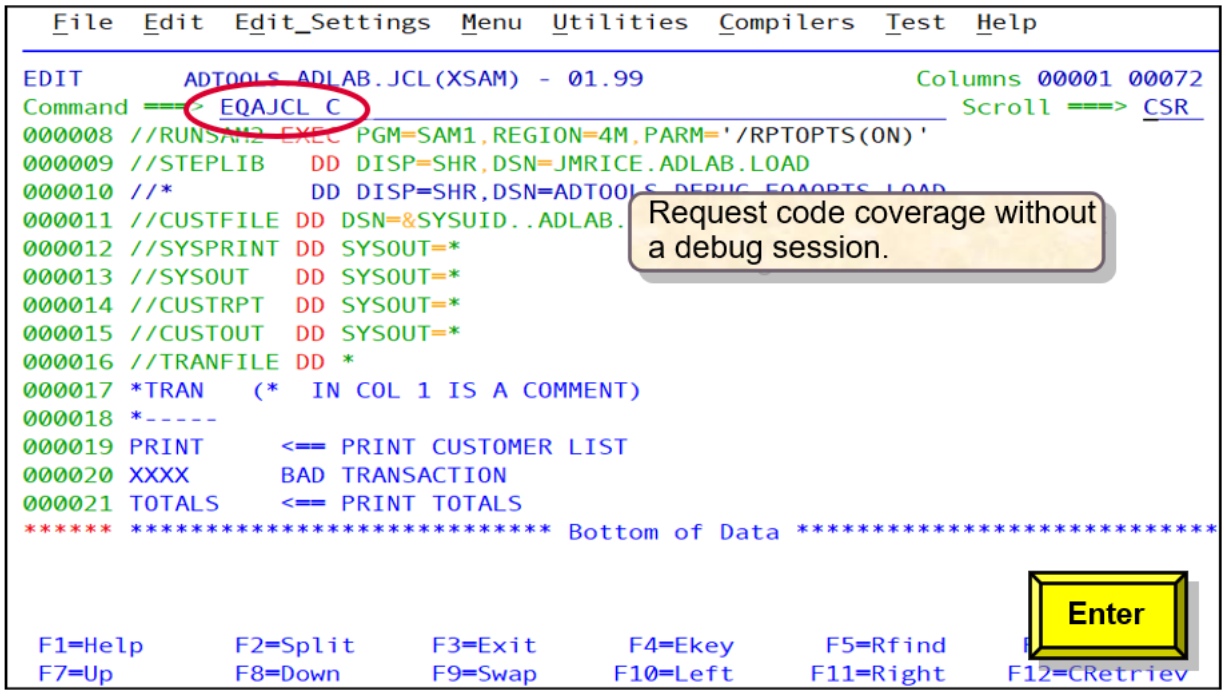
The identification of the code coverage files can be included by your installation team. However, the defaults are not to provide this information. The IBM z/OS Debugger JCL Wizard routine defaults are set to CODE_COVERAGE_SETUP = YES. When this value is specified, the EQAOPTS DD statements shown above are generated to create the appropriate file properties for z/OS Debugger Code Coverage.

For information about what compilers are supported and which compiler options are required, see Appendix E, “z/OS Debugger Code Coverage,” on page 461.

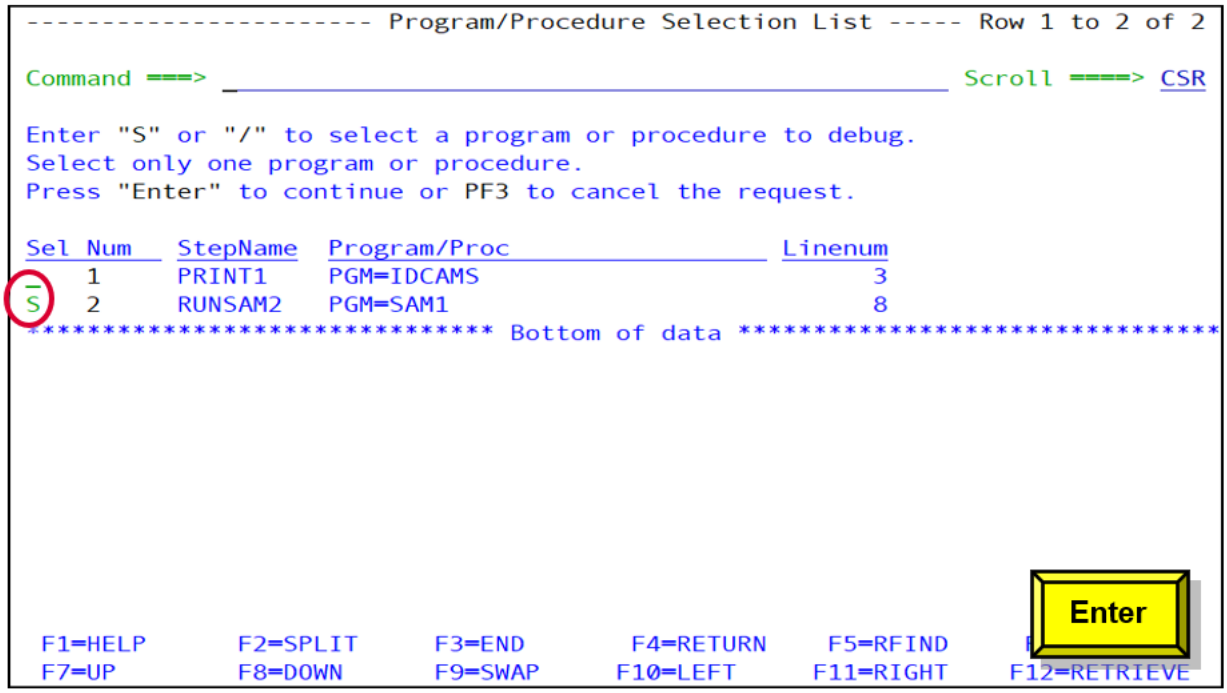
Starting z/OS Debugger Code Coverage without a debug session

You can start z/OS Debugger Code Coverage without an interactive z/OS Debugger session.

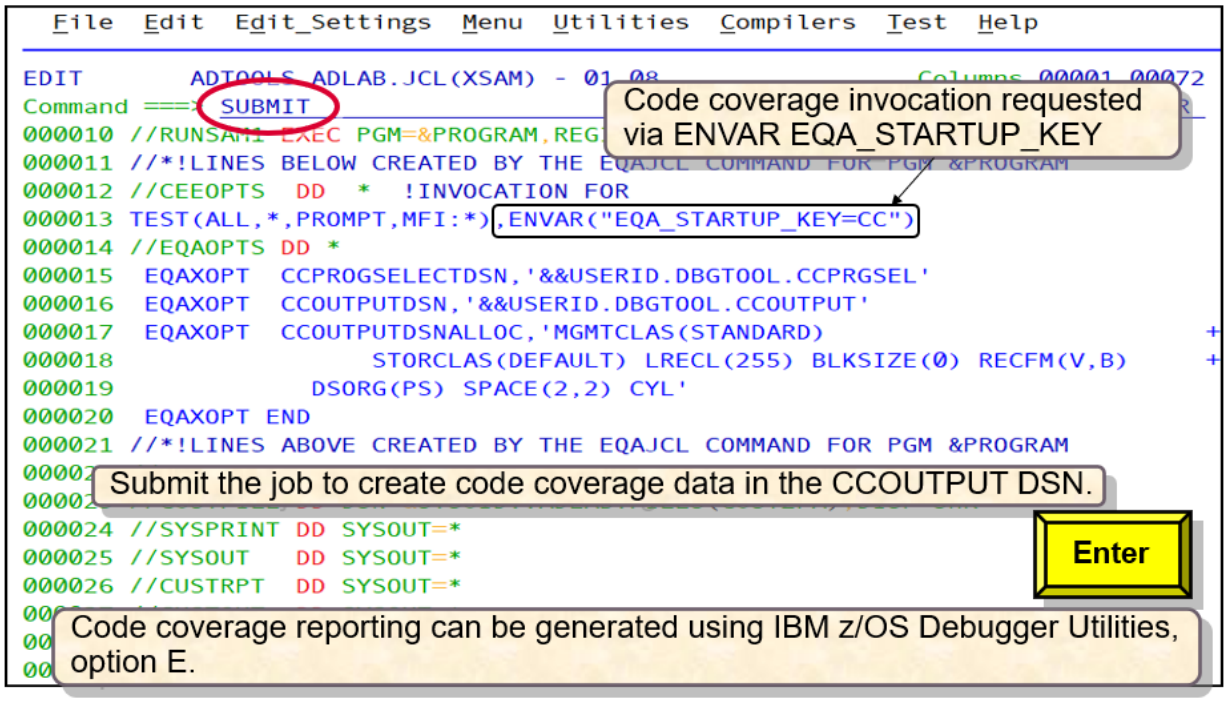
1. To generate code coverage data without a debug session, in ISPF Edit or Browse, enter **EQAJCL C** to bypass the **z/OS Debugger JCL Wizard Option Selection** panel. Alternatively, enter **EQAJCL** to invoke the IBM z/OS Debugger JCL Wizard and then select **C**.



2. In the **Program/Procedure Selection List** panel, select the program step name or procedure that you want to gather code coverage for.



3. Lines are added to the JCL to invoke z/OS Debugger and initiate code coverage with a Language Environment variable EQA_STARTUP_KEY=CC added to CEEOPTS DD. The EQAOPTS DD statements are generated to provide the appropriate data sets for code coverage.

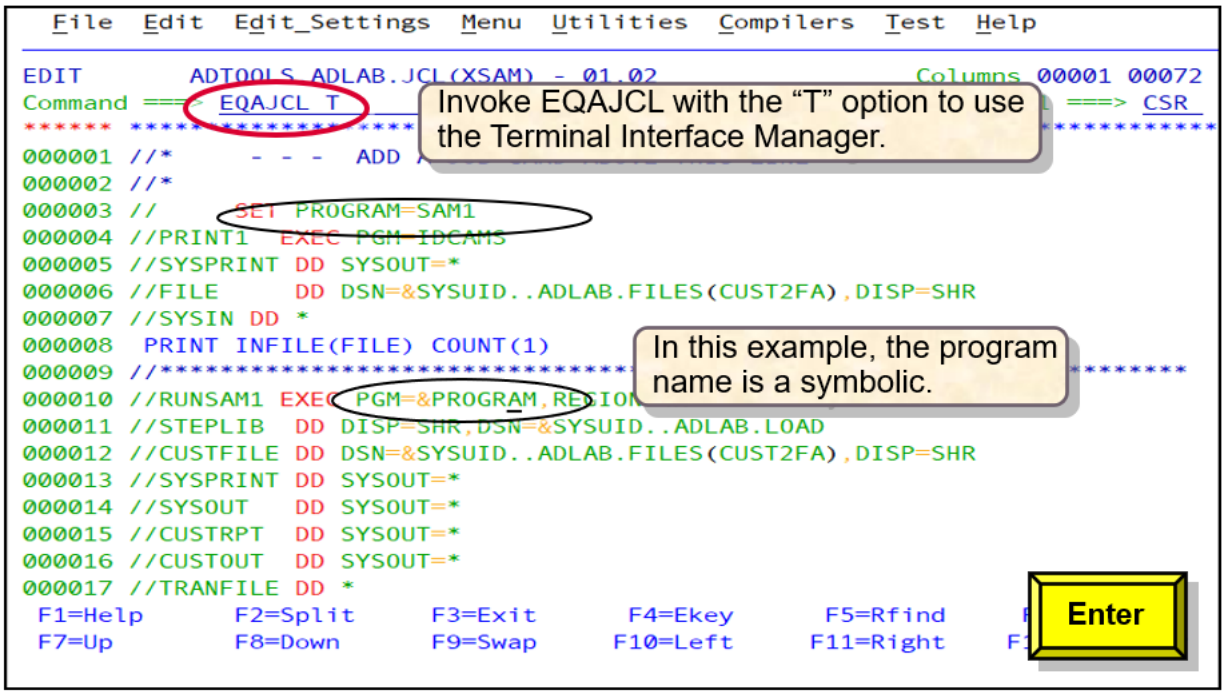


After the program completes, you can review the code coverage information by using the IBM z/OS Debugger Utilities option E z/OS Debugger Code Coverage.

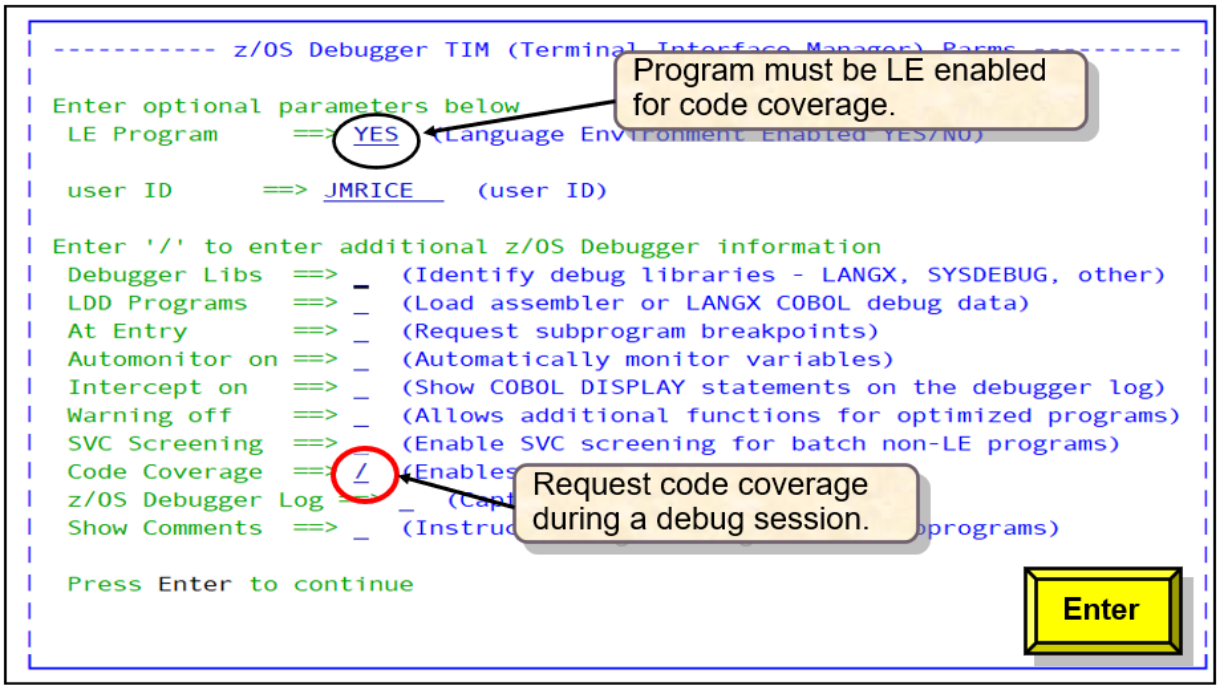
z/OS Debugger Code Coverage with a debug session using Terminal Interface Manager

You can start z/OS Debugger Code Coverage to generate code coverage information during an interactive z/OS Debugger session by using the Terminal Interface Manager.

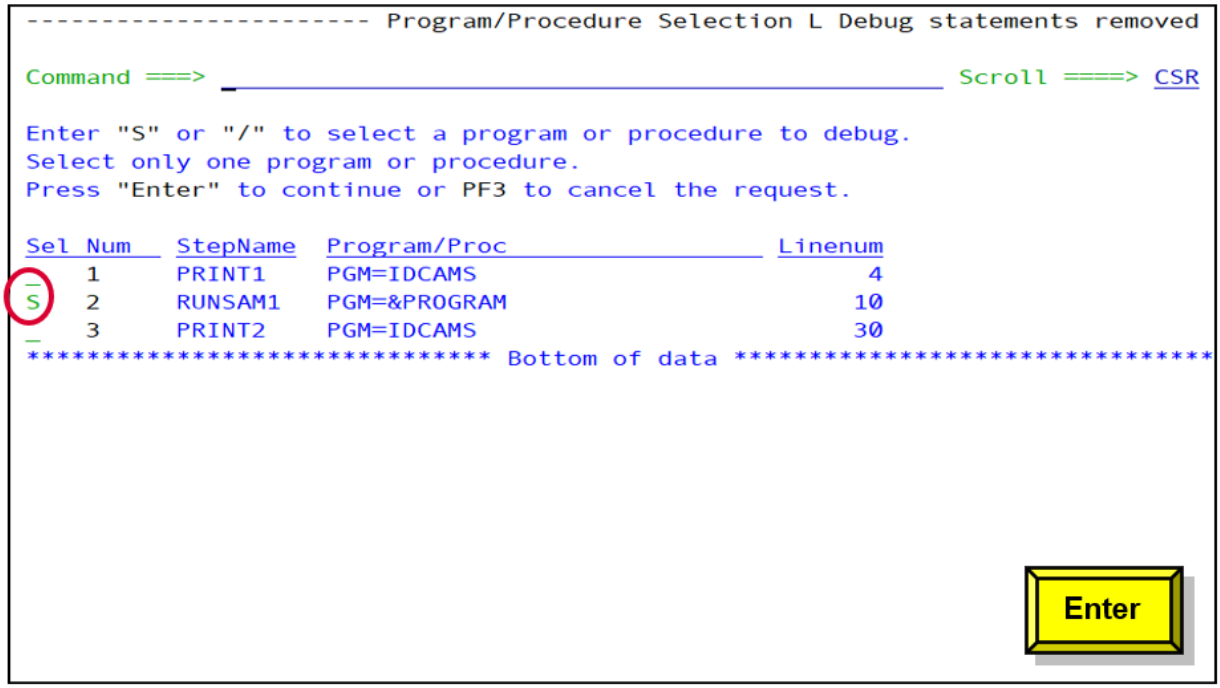
1. In ISPF Edit or Browse, enter **EQAJCL T** to navigate to the **z/OS Debugger TIM (Terminal Interface Manager) Parms** panel.



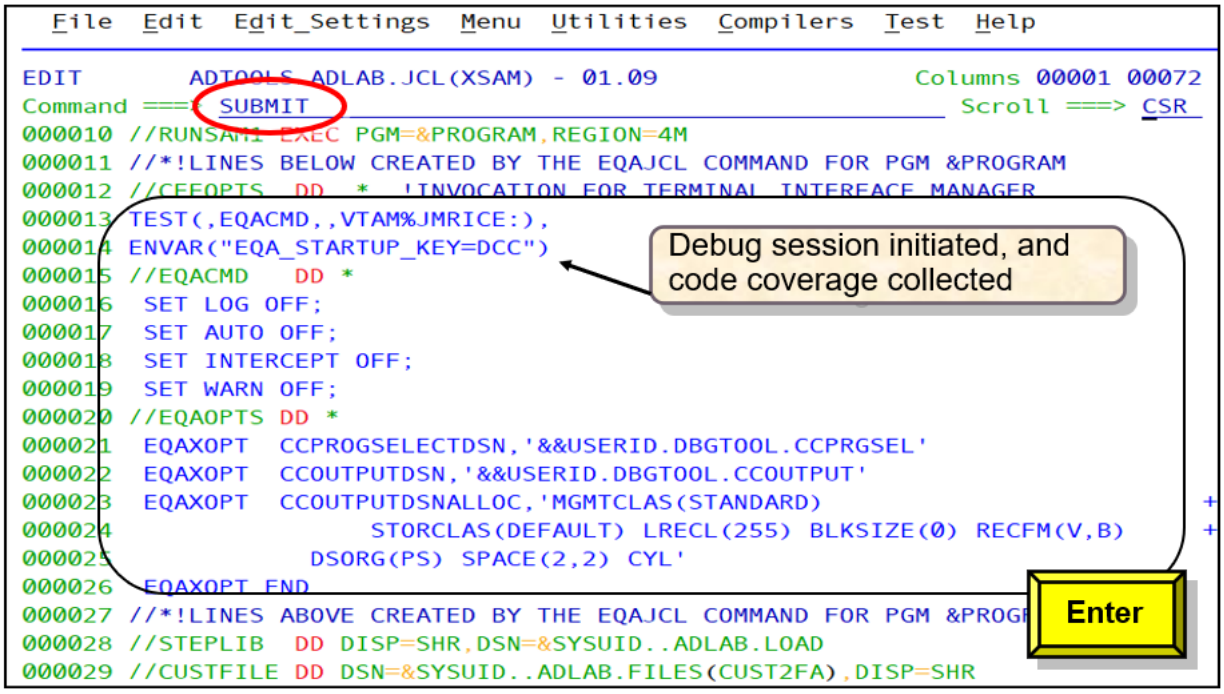
2. Code coverage is available for Enterprise COBOL, Enterprise PL/I or z/OS XL C programs that are compiled with certain compilers and with the appropriate options. To collect the code coverage information, enter a forward slash (/) in the **Code Coverage** field.



3. In the **Program/Procedure Selection List** panel, select the program and step name that you want to debug and gather code coverage for.



4. JCL statements are generated to invoke z/OS Debugger. You can use the JCL to start a debug session on the Terminal Interface Manager, and collect code coverage information.

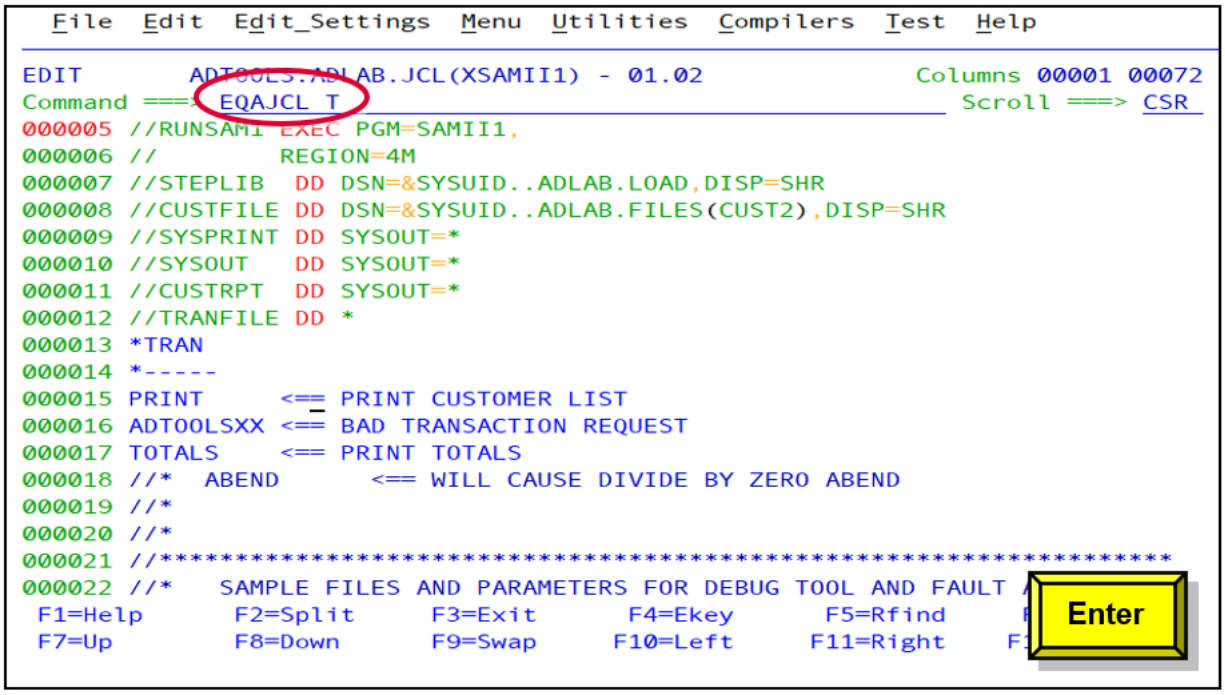


After the debug session completes, you can view this information by using the IBM z/OS Debugger Utilities menu, option E z/OS Debugger Code Coverage.

Debugging a Language Environment VS COBOL II program compiled with the NOTEST option using Terminal Interface Manager

You can use either the TEST or NOTEST compilation option to debug a VS COBOL II program. This use case is about debugging a Language Environment VS COBOL II program compiled with the NOTEST option. This method is called LANGX COBOL in the z/OS Debugger manuals.

1. In ISPF Edit or Browse, enter **EQAJCL T** for Terminal Interface Manager.

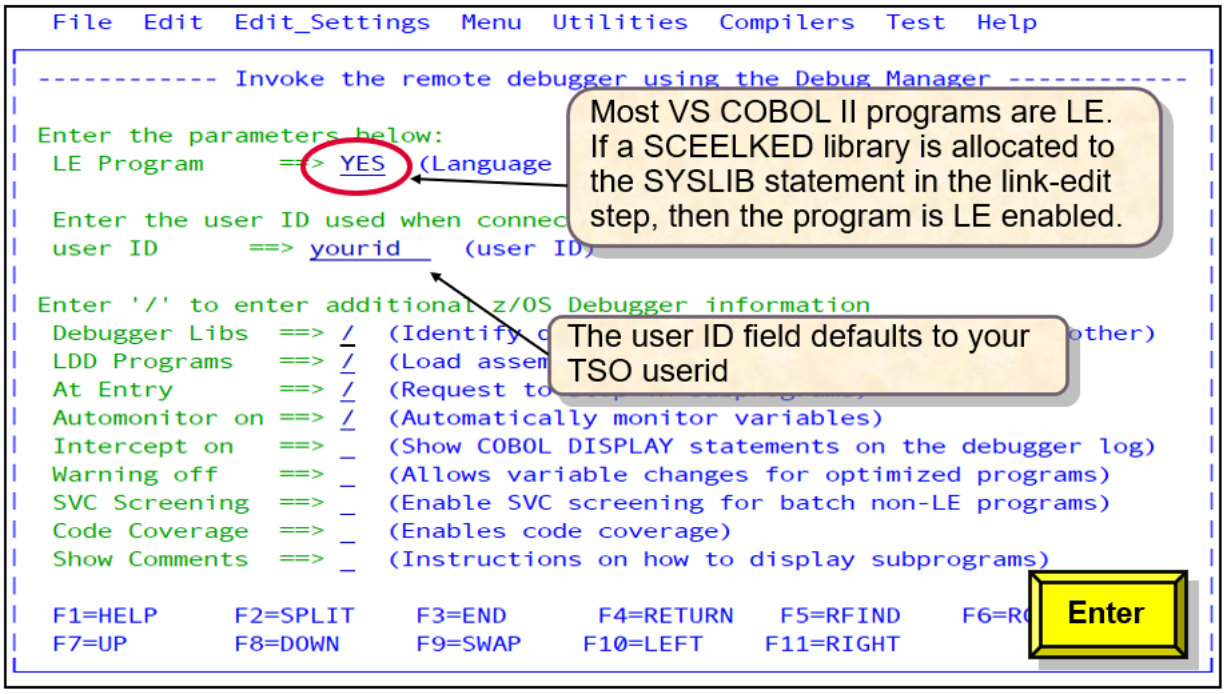


2. In the parameters section panel, specify the values.

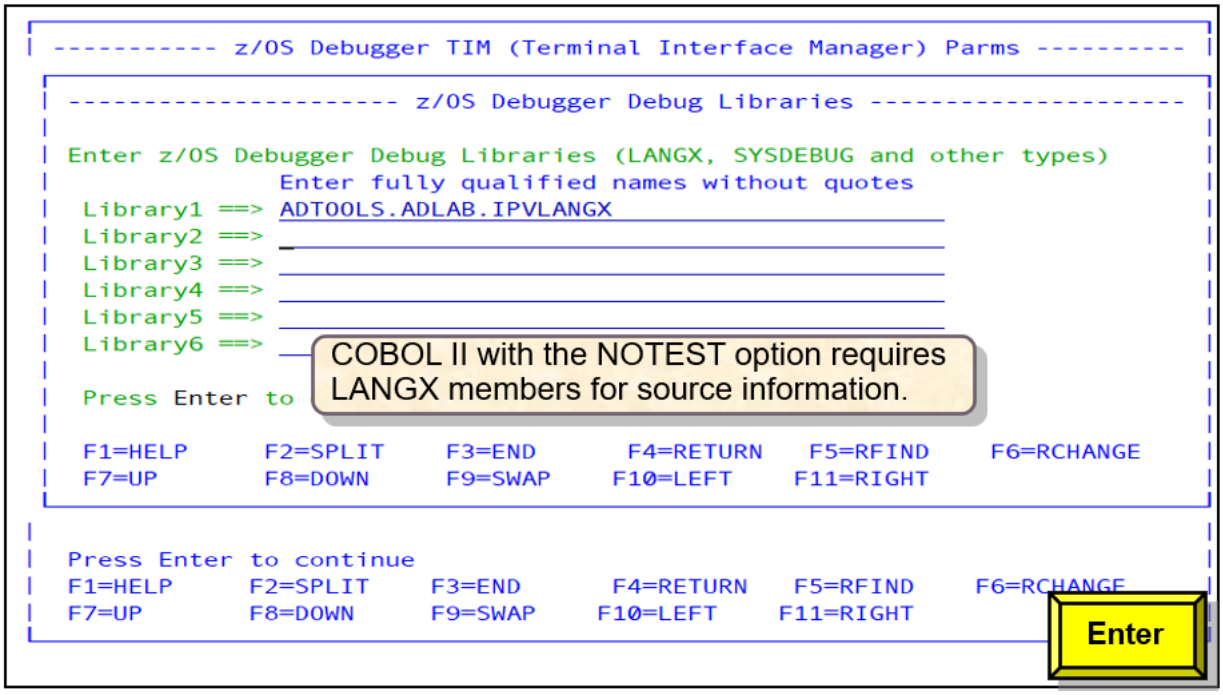
   VS COBOL II programs might be linked either as Language Environment (LE) programs or non-Language Environment (non-LE) programs. In this use case, the program is linked as Language Environment enabled. Type **YES** in the **LE Program** field.

   To select an optional parameter, specify a forward slash (/) in the field.

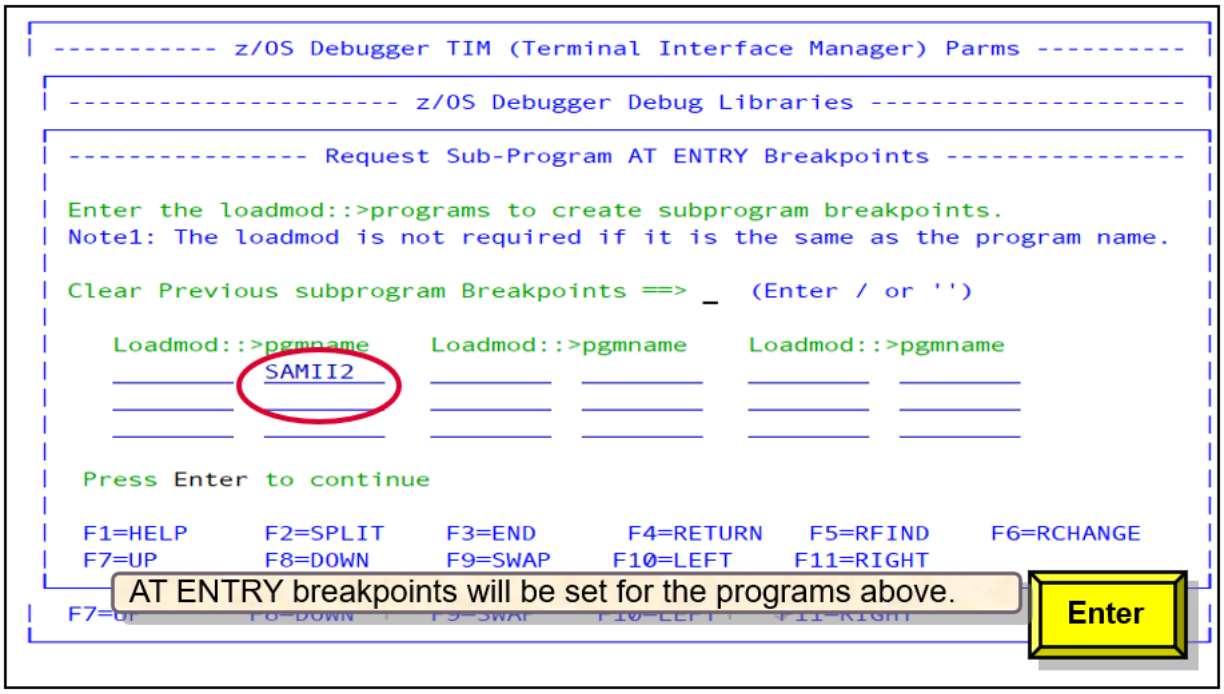
   Although this is a Language Environment program, you still need to identify the z/OS Debugger debug libraries, and issue LDD statements for the modules that you want to debug. Optionally, you can set breakpoints for subprograms, or set the automonitor on.



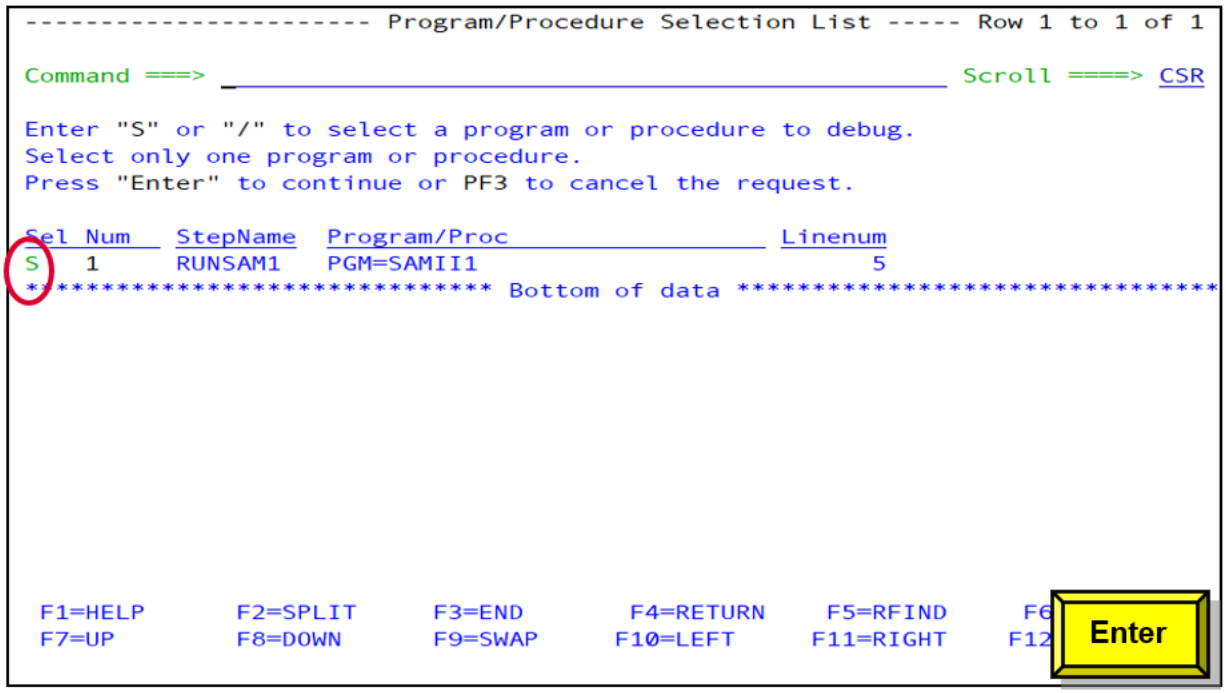
3. Since you chose **Debugger Libs**, the **z/OS Debugger Debug Libraries** panel is displayed. In this panel, enter one or more names of the z/OS Debugger side files that you want to use.



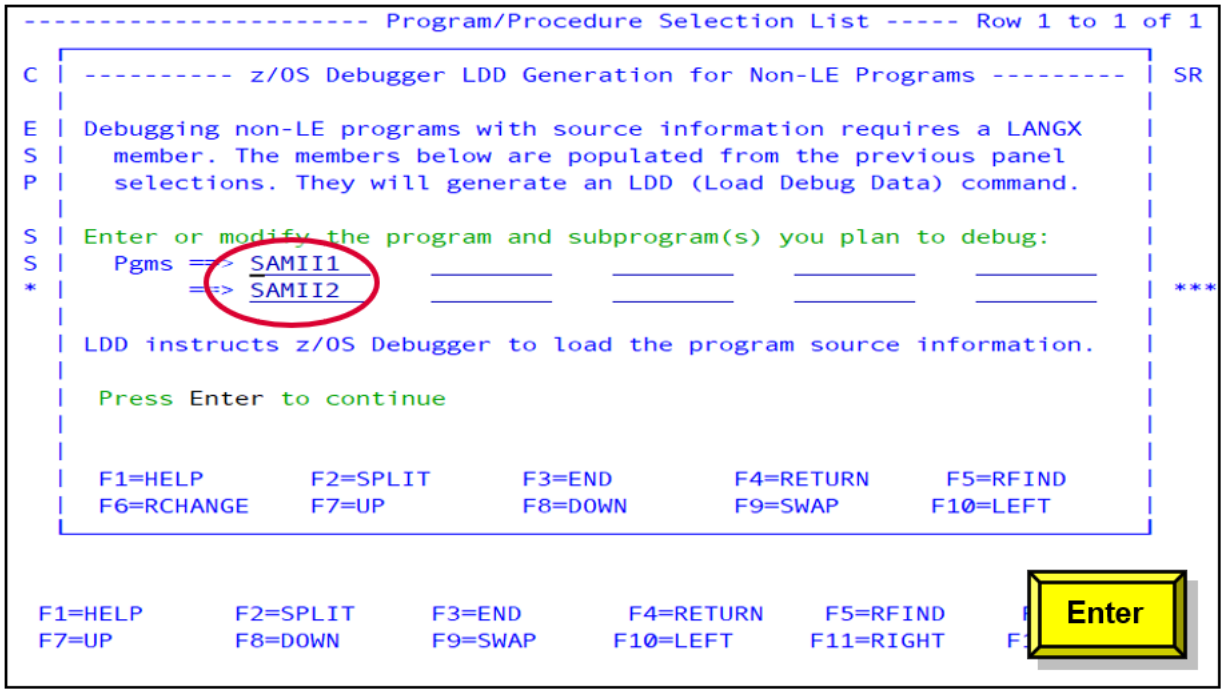
4. Since you chose to set AT ENTRY breakpoints for subprograms, the **Request Sub-Program AT ENTRY Breakpoints** panel is displayed. Enter one or more names of the z/OS Debugger side file data sets that you want to use.



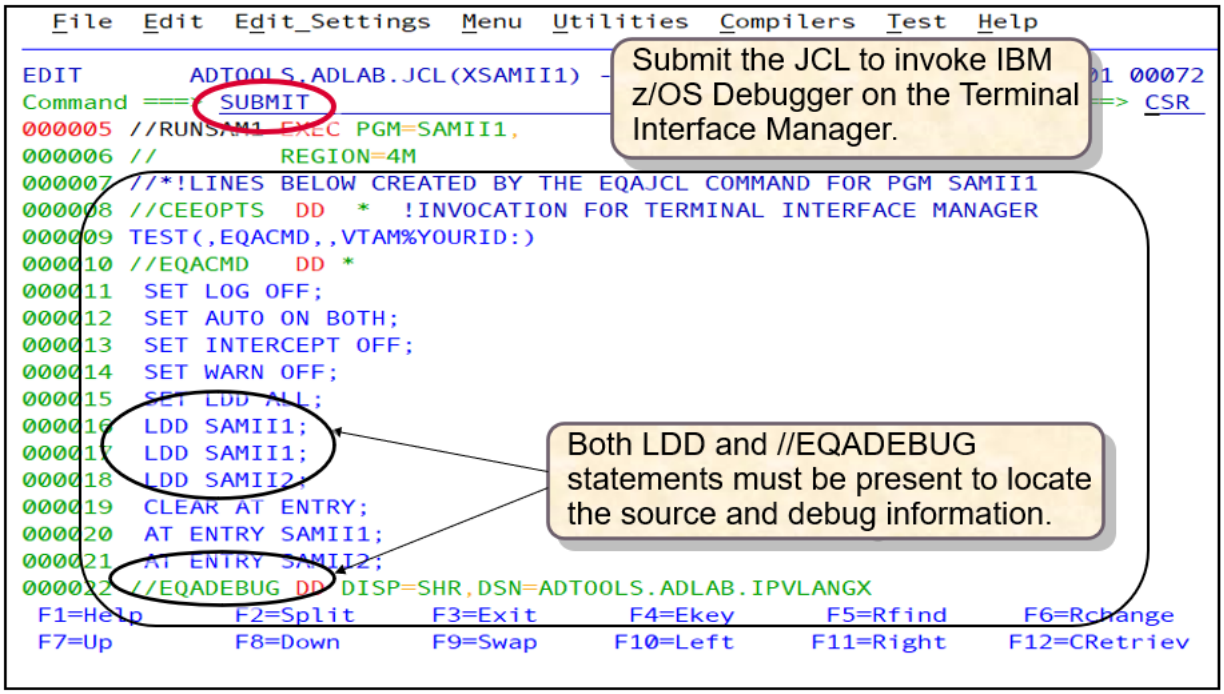
5. In the **Program/Procedure Selection List** panel, enter the step name or procedure that you want to debug.



6. Since you chose **LDD Programs**, the **z/OS Debugger LDD Generation for Non-LE Program Name** panel is displayed. The **z/OS Debugger LDD Generation for Non-LE Programs** panel is defaulted to the LDD name provided in the **z/OS Debugger LDD generation for Non-LE Program Name** panel and the subprograms provided in the **Request Sub-Program AT ENTRY Breakpoints** panel. You can override or add more entries to this panel to depict the LANGX members that you want to use in the debug session.



7. The JCL for VS COBOL II z/OS Debugger invocation is created. To define the source and debug information during the debug session, the LDD statements and EQADEBUG libraries are required.

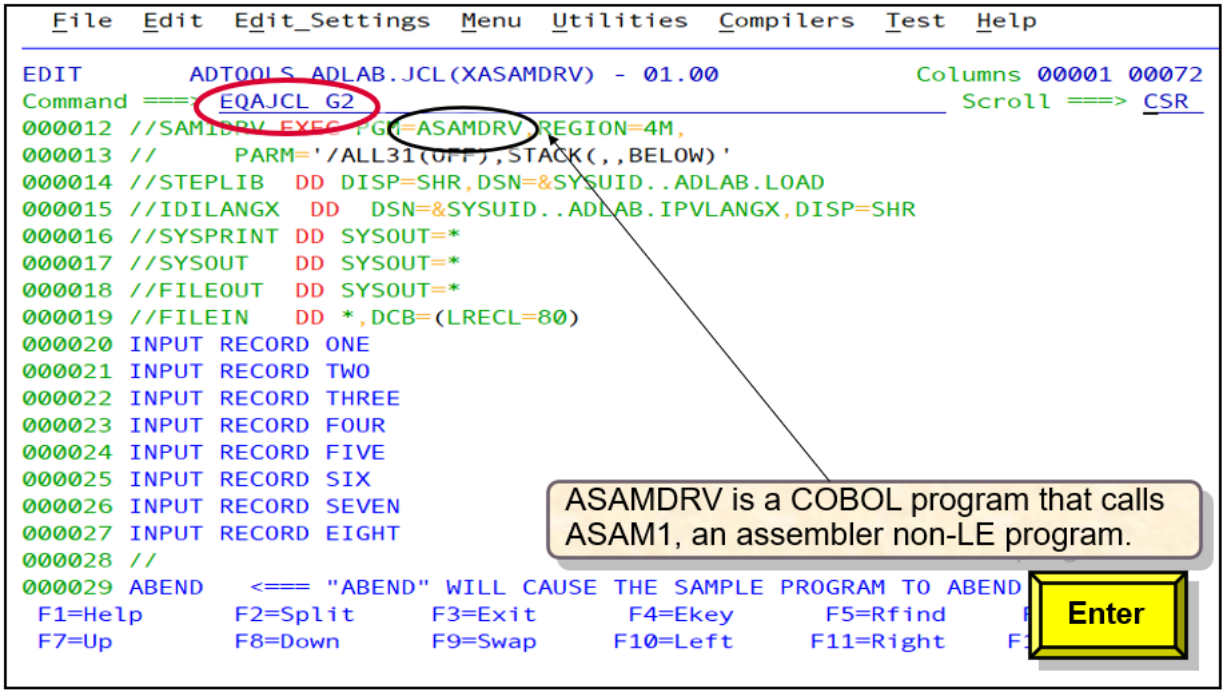


Debugging a Language Environment COBOL program that calls non-Language Environment subprograms

SVC screening is required when the first program is Language Environment enabled and non-Language Environment subprograms are called. SVC screening is used only for assembler subprograms in non-CICS environments.

1. In ISPF Edit or Browse, enter **EQAJCL G2** to invoke IBM z/OS Debugger JCL Wizard, and request a debug session with the remote debugger. Alternatively, you can enter **EQAJCL** and then select **G2**.

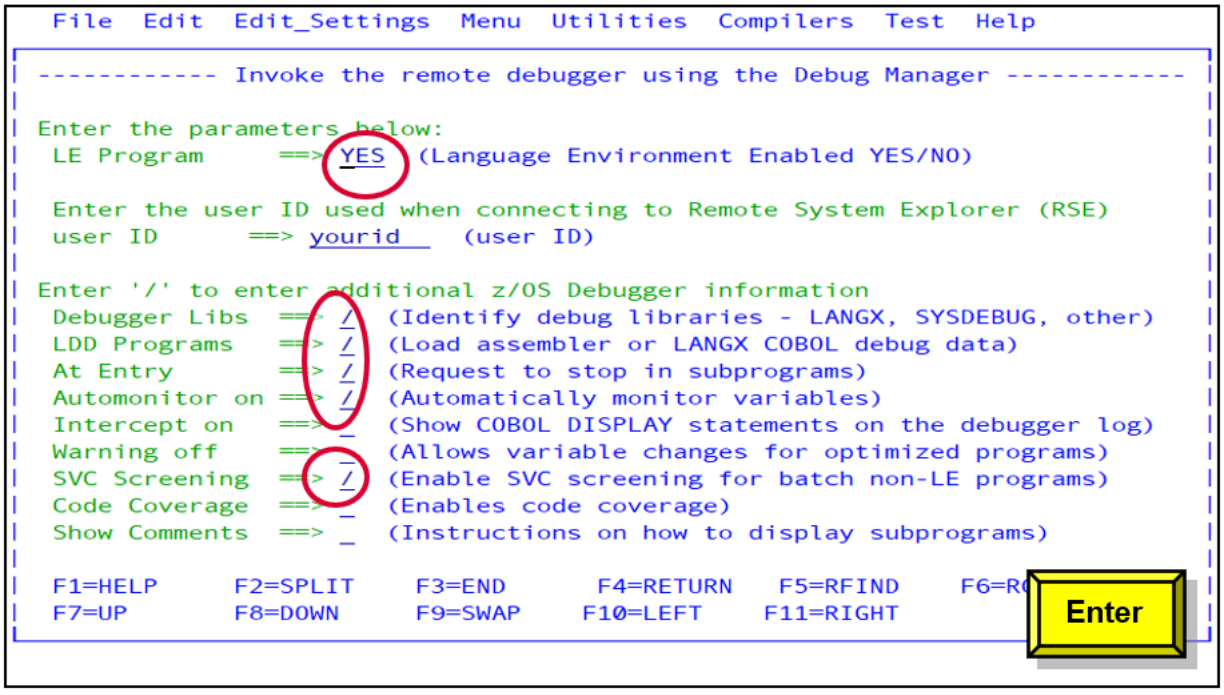
In this use case, the Language Environment COBOL program ASAMDRV will call the non-Language Environment assembler program ASAM1.



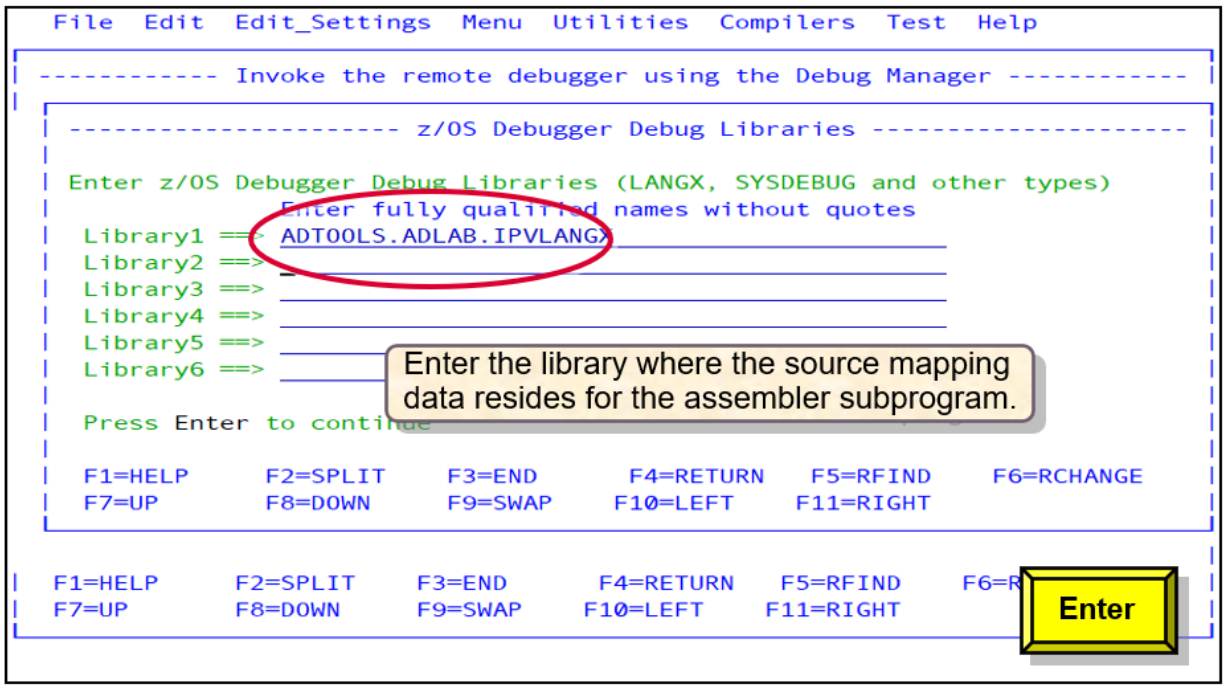
2. In the parameters panel, type **YES** in the **LE Program** field.

   To select an optional parameter, specify a forward slash (/) in the field.

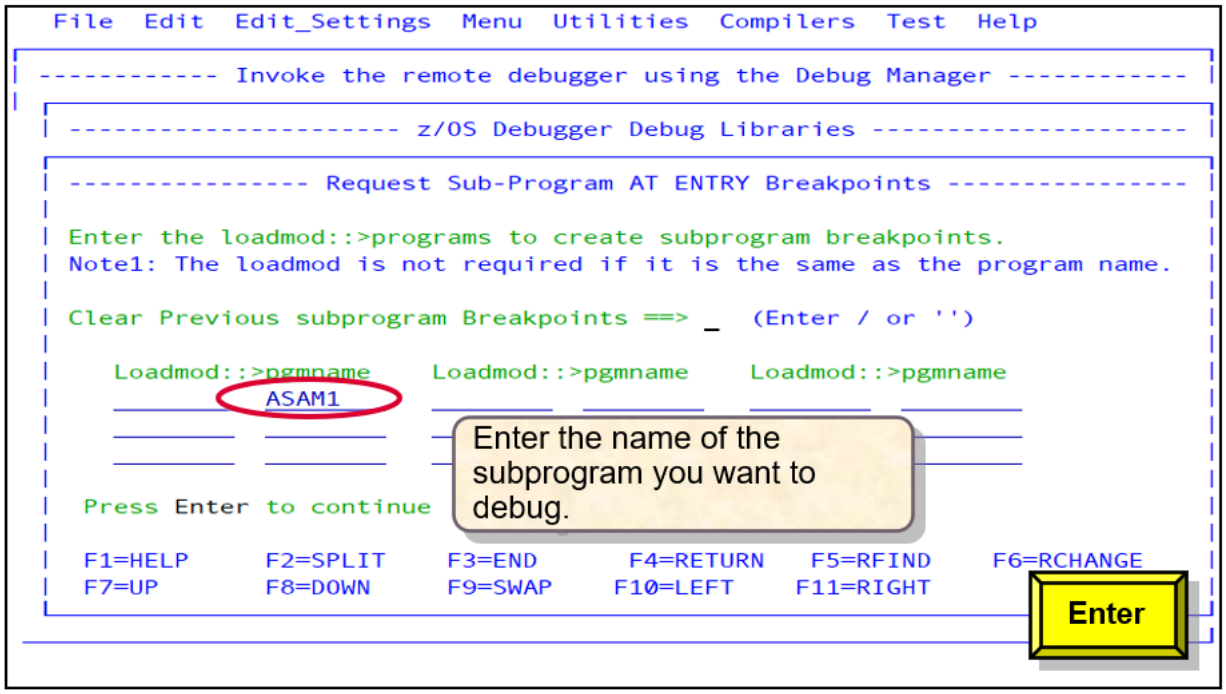
   The assembler subprogram requires debug libraries, LDD programs and AT ENTRY breakpoints. Because the subprogram is a non-Language Environment one invoked from a Language Environment program, SVC screening is also required. Specify a forward slash (/) in the corresponding fields.



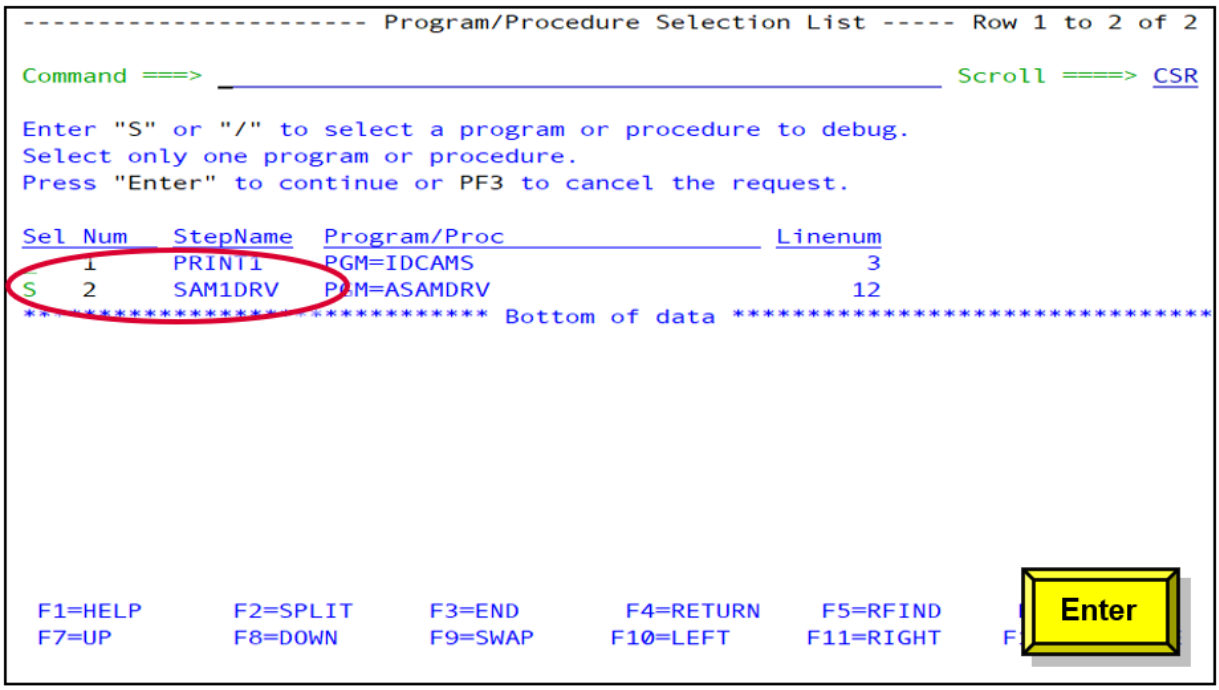
3. Since you chose **Debugger Libs**, the **z/OS Debugger Debug Libraries** panel is displayed. In this panel, enter the library where the debugger source information is found.



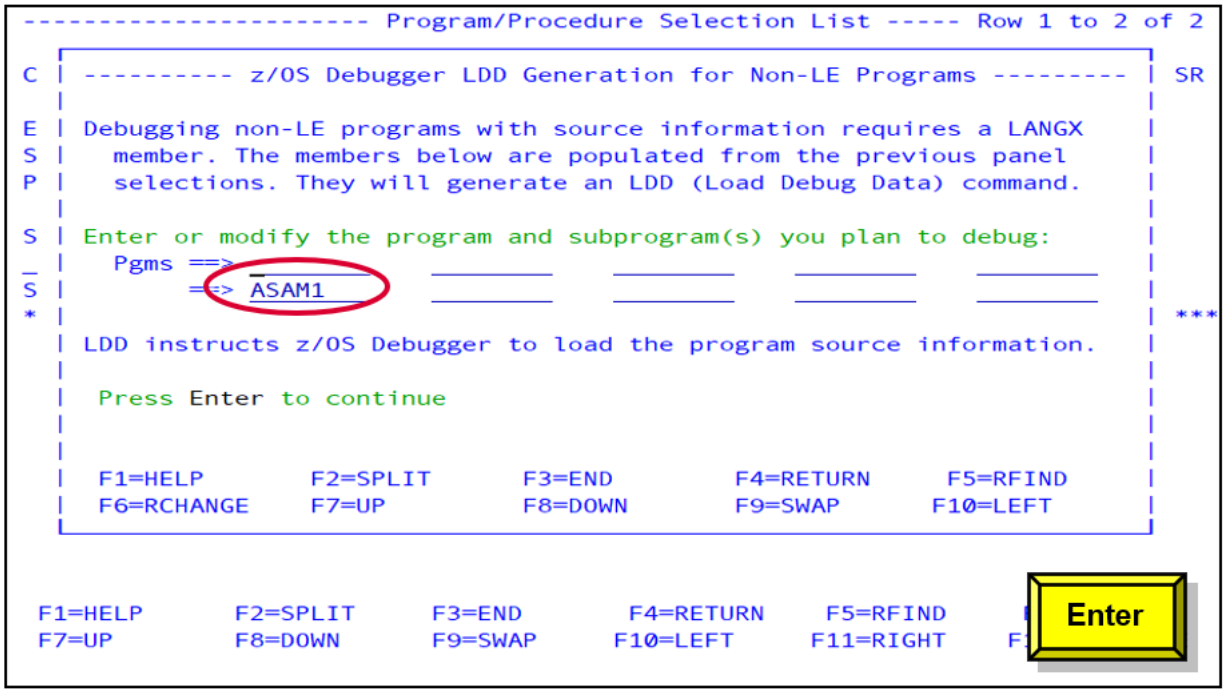
4. Since you chose to set AT ENTRY breakpoints for subprograms, the **Request AT ENTRY Sub-Program Breakpoints** panel is displayed. In this panel, enter the name of the subprogram that you want to debug.



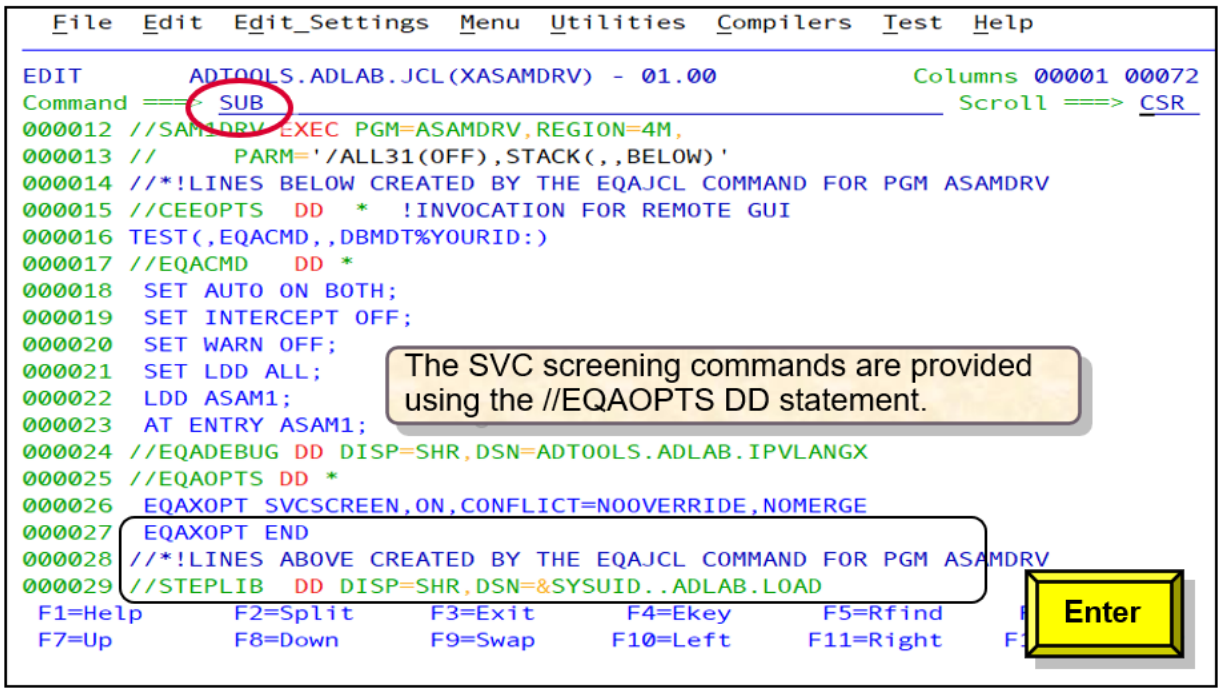
5. In the **Program/Procedure Selection List** panel, select the step to debug.



6. Since you chose **LDD Programs**, the **z/OS Debugger LDD Generation for Non-LE Program Name** panel is displayed. This panel is pre-populated with the initial program and the programs specified in the **Request AT ENTRY Sub-Program Breakpoints** panel. Load Data Descriptor (LDD) commands are generated from this panel.



7. JCL statements are generated to invoke z/OS Debugger.

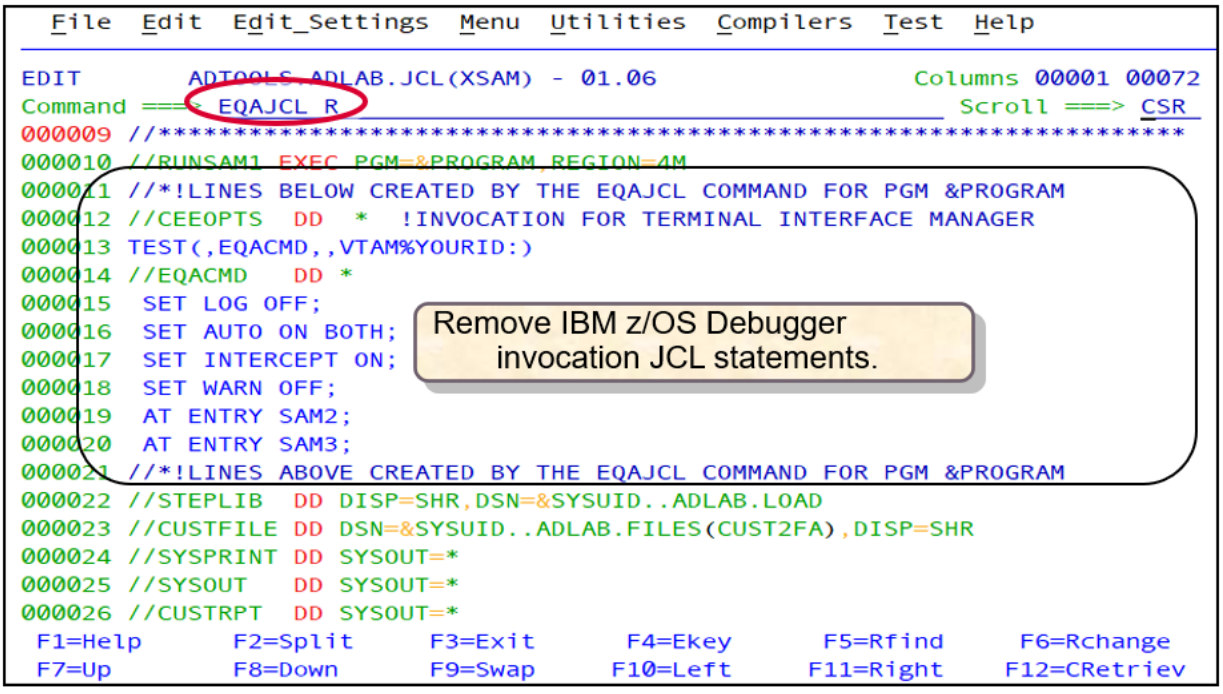


The EQAOPTS override statements in the example will initiate SVC screening to enable debugging non-Language Environment when the initial program is Language Environment.

Removing IBM z/OS Debugger JCL Wizard statements

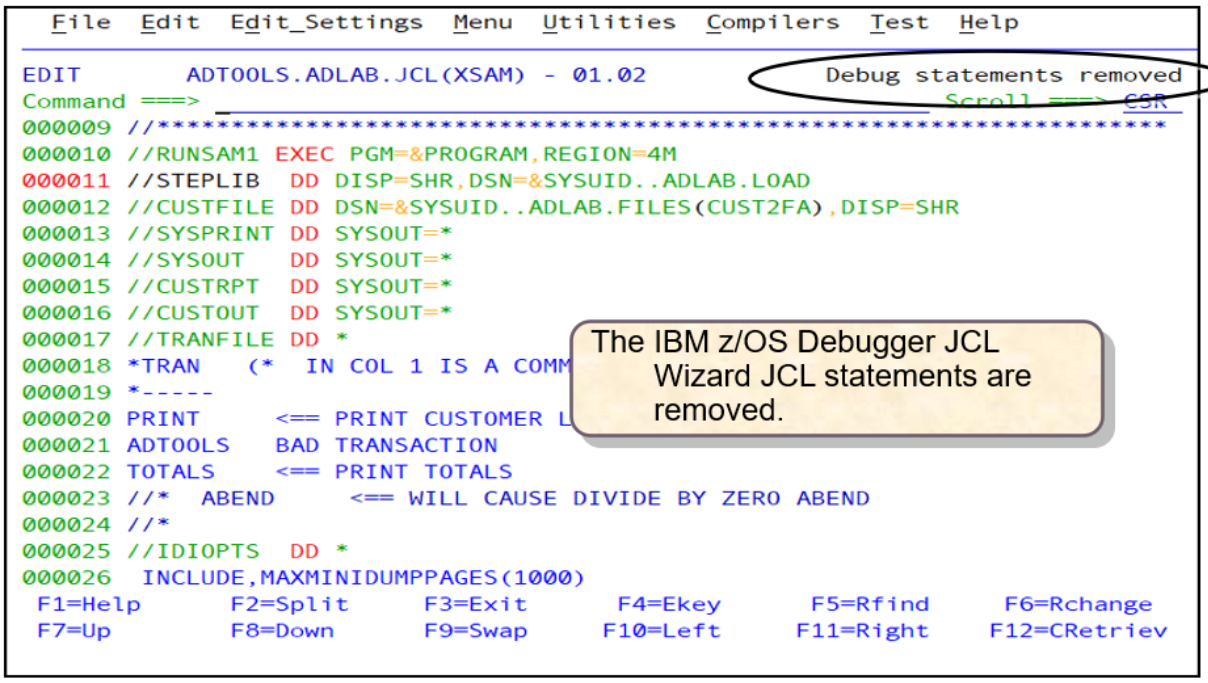
You can remove the JCL statements generated by the IBM z/OS Debugger JCL Wizard with the **EQAJCL R** command.

1. In ISPF Edit or Browse, enter **EQAJCL R**.



The first and last lines generated are comment lines. If you modify the comment lines, the statements that are generated by the IBM z/OS Debugger JCL Wizard might not be removed properly.

2. The JCL statements that are generated by IBM z/OS Debugger JCL Wizard are removed.


File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT ADTOOLS.ADLAB.JCL(XSAM) - 01.02
Debug statements removed
Command ===>
Scroll ===> CSR
000009 //*****
000010 //RUNSAM1 EXEC PGM=&PROGRAM,REGION=4M
000011 //STEPLIB DD DISP=SHR,DSN=&SYSUID..ADLAB.LOAD
000012 //CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
000013 //SYSPRINT DD SYSOUT=*
000014 //SYSOUT DD SYSOUT=*
000015 //CUSTRPT DD SYSOUT=*
000016 //CUSTOUT DD SYSOUT=*
000017 //TRANFILE DD *
000018 *TRAN (* IN COL 1 IS A COMM
000019 *-----
000020 PRINT <== PRINT CUSTOMER L
000021 ADTOOLS BAD TRANSACTION
000022 TOTALS <== PRINT TOTALS
000023 //* ABEND <== WILL CAUSE DIVIDE BY ZERO ABEND
000024 //*
000025 //IDIOPTS DD *
000026 INCLUDE,MAXMINIDUMPPAGES(1000)
F1=Help F2=Split F3=Exit F4=Ekey F5=Rfind F6=Rchange
F7=Up F8=Down F9=Swap F10=Left F11=Right F12=CRetriev
The IBM z/OS Debugger JCL Wizard JCL statements are removed.

Appendix E. z/OS Debugger Code Coverage

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can use IBM z/OS Debugger to generate, view, and report code coverage observations. The code coverage observations can be generated interactively or in batch mode.

There are five activities that are described here:

1. Setup: Start a code coverage session with z/OS Debugger.
2. Code coverage observations gathering: Using z/OS Debugger to generate the code coverage observations.
3. Selection and filtering: Creating a selection and filtering criteria to be used in the creation of a report.
4. Viewing: Viewing code coverage observations interactively.
5. Report creation: Creating a code coverage report that is based on the selection and filtering criteria that are provided.

Batch facilities are provided so that collection of the code coverage data, using selection criteria to create extracted observations, and report creation can be done in unattended mode (batch).

Overview of z/OS Debugger Code Coverage

You can use z/OS Debugger to measure code coverage in your application testing. In this part, you can learn the basics of running the code coverage function of z/OS Debugger from setup to generating reports. New users are encouraged to read this part to learn the basics of the tool, including how to create code coverage observations and the use of the ISPF dialogs.

Introduction to z/OS Debugger Code Coverage

z/OS Debugger Code Coverage measures test case code coverage in application programs that are written in COBOL, PL/I and C and compiled with certain compilers and compiler options. The code coverage function enables you to test your application and generate information to determine which code statements are executed.

The code coverage function in z/OS Debugger has the following advantages:

- You can use the same load modules that you use when you develop your application to generate the code coverage data.
- In some cases, the debugger can help reach sections of code that are difficult to simulate with a test case during development. When such needs arise, z/OS Debugger marks the observations with special indicator so it is known that interaction with the user created a deviation from the normal logic of the program.
- You can run code coverage unattended using batch facilities.
- XML is used to render information, which makes it easier for users to develop their own facilities to present and evaluate information.

This section contains the following topics:

- Graphical Overview of the process of starting a code coverage data gathering session with z/OS Debugger, creating code coverage reports, and displaying the reports.
- Startup
- EQAOPTS
- IBM z/OS Debugger Utilities Option E. z/OS Debugger Code Coverage
  - Observation Viewer. Option E.1: Browse code coverage observations.

© Copyright IBM Corp. 1992, 2023

- z/OS Debugger Options. Option E.2: Create or modify z/OS Debugger Code Coverage options.
- Observation Selection Criteria. Option E.3: Create or modify the observation selection criteria and source markers.
- Observation Extraction. Option E.4: Extract code coverage observations using selection criteria.
- Report Generation. Option E.5: Create reports.

Collecting code coverage observations with z/OS Debugger

The following figure shows the steps that are required for z/OS Debugger to collect code coverage information. The key elements are as follows:

- *EQA_STARTUP_KEY*. An environment variable that needs to be specified at the start of the z/OS Debugger Code Coverage session.
- An Options file that indicates what programs you want z/OS Debugger to monitor to get code coverage observations.
- EQAOPTS commands that indicate the location of the input and output data sets.

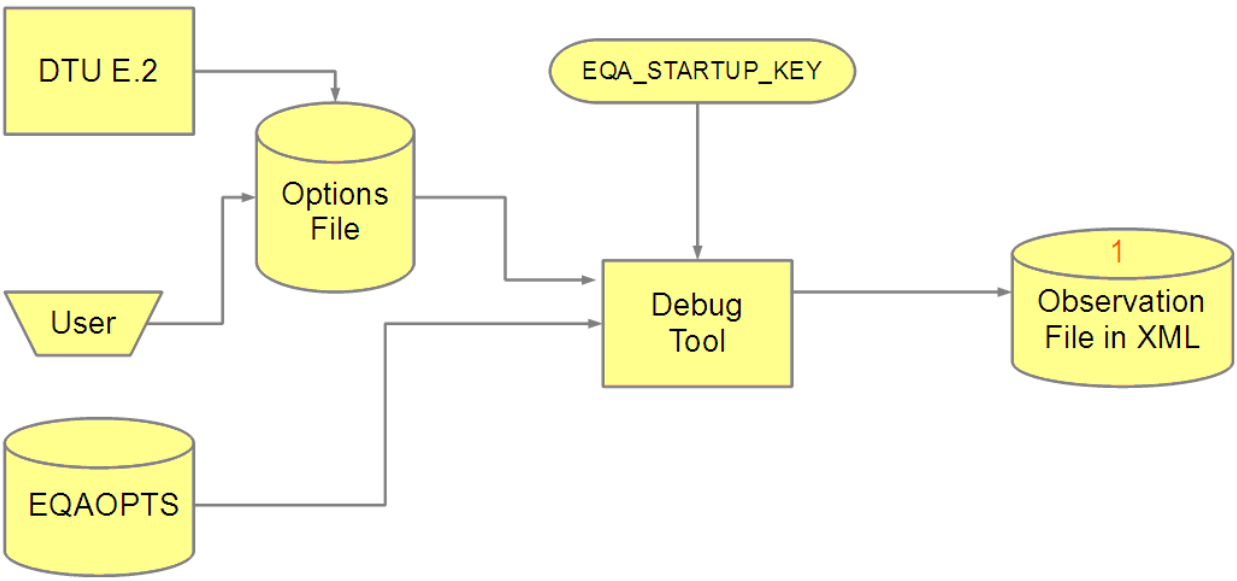


Figure 3. Step 1 Gathering code coverage observations with z/OS Debugger

The code coverage observations collection process is as follows:

1. When the environment variable *EQA_STARTUP_KEY* is specified during invocation of the debugger, z/OS Debugger collects code coverage observations.
2. z/OS Debugger gathers code coverage data based on input from the Options file.
3. The Options file can be created by using IBM z/OS Debugger Utilities Option E.2. Alternatively, you can code an Options file by following the Options file XML DTD syntax.
4. z/OS Debugger retrieves the Options file data set name from a value that is provided by an EQAOPTS command.
5. z/OS Debugger retrieves the Observation file data set name from a value that is provided by an EQAOPTS command.

Code coverage selection and extraction process

The following figure shows the selection and extraction process. In this process, a code coverage Observation file that is created during a z/OS Debugger Code Coverage session is evaluated using a Selection file. The Selection file is provided by the user, and it indicates the type and granularity of the code coverage extracted observations that must be extracted from the original Observation file. For

example, you want a report for only a program with a specific compile time and date. The Selection file can be created using IBM z/OS Debugger Utilities Option E.3. Alternatively, you can code a Selection file by following the Selection file XML DTD syntax.

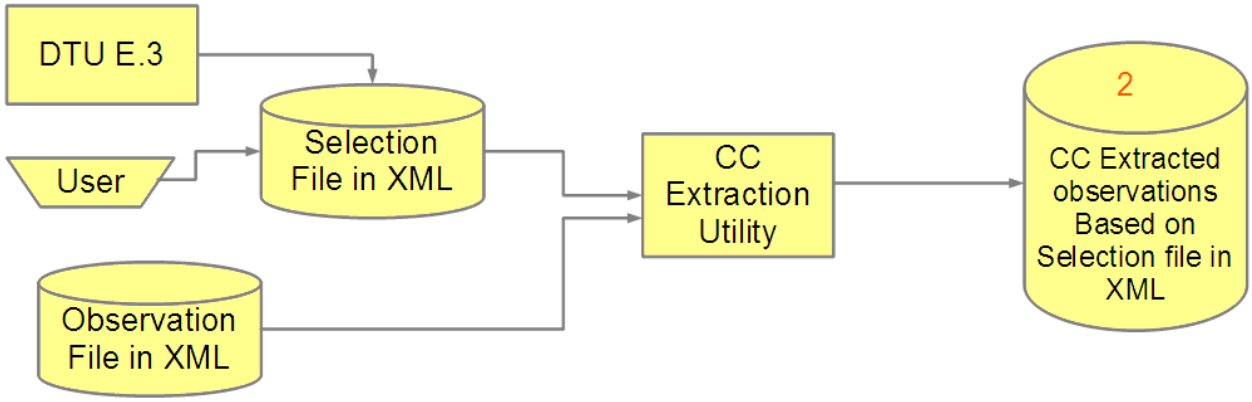


Figure 4. Step 2 Code coverage selection and extraction process

The code coverage selection and extraction process is as follows:

1. The code coverage Extraction Utility operates on the observations and applies the selection criteria to create a file with extracted observations based on the selections.
2. The Selection file can be created using IBM z/OS Debugger Utilities Option E.3. Alternatively, you can code a Selection file by following the Selection file XML DTD syntax.
3. You can run the code coverage Extraction Utility from DTU E.4. When you select this option, you are prompted to provide the name of the selection file, the Observation file, and the file where the code coverage extracted observations are stored.
4. You can run the code coverage Extraction Utility in batch as well by running the EQAXCCX2 REXX exec. You must specify the following DDNAMES:

   EQACSINP
   Location of Observation file.

   EQACSSEL
   Location of Selection file.

   EQACSOUT
   Location of output code coverage extracted observations file.

   An example of using EQAXCCX2 in batch can be found in *hlq*.SEQASAMP(EQACCEXT).

Code coverage reporting process

The following figure shows the process for creating a XML report of the code coverage results. The report can be created by using batch facilities or by using IBM z/OS Debugger Utilities Option E.5 suboption 1. The input to the report utility is the Selection file that is created by the user in the *Step 2. Code Coverage selection and extraction process*, the resulting data set from that process, and the Code Coverage extracted observations data set.

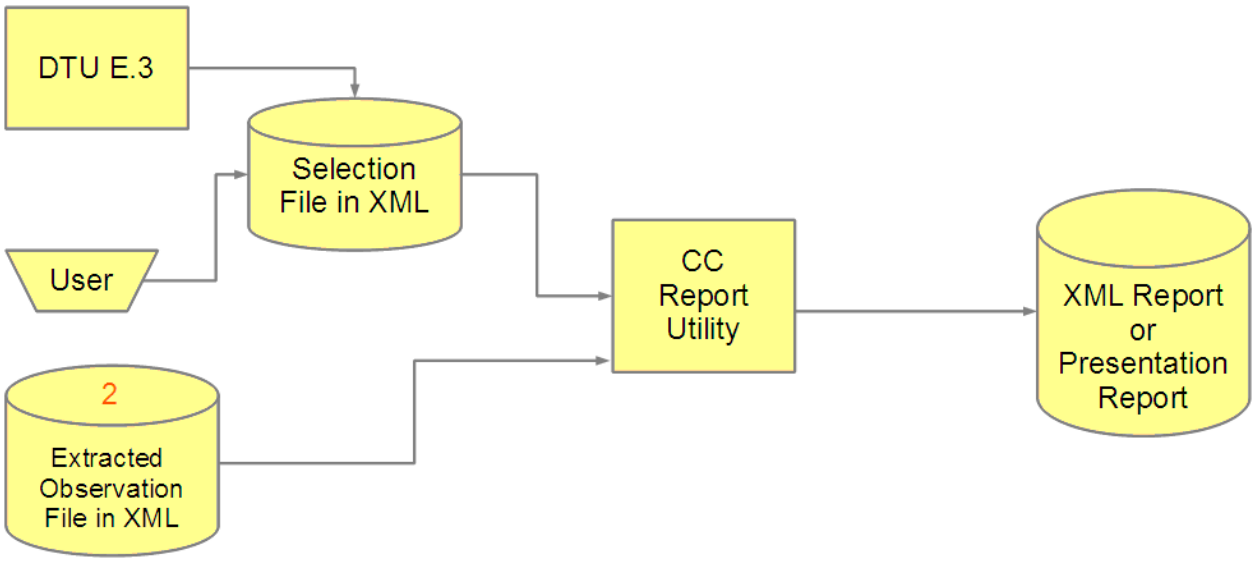


Figure 5. Step 3 Code coverage report process

The code coverage XML report process is as follows:

1. The code coverage Report Utility uses the code coverage extracted observations that are created after you apply the selection criteria to create the code coverage report.
2. The code coverage Report Utility also uses the Selection file that is created by using DTU Option E.3 or coded manually by following the Selection file XML DTD syntax to include only the selection criteria as part of the report.
3. You can start the code coverage Report Utility from DTU Option E.5 suboption 1. When you select this option, you can provide the name of the Selection file, the extracted Observation file, and the file where the XML report is stored.
4. The code coverage Report Utility can be run in batch as well by running the EQAXCCR2 REXX exec with the XML parameter. You must specify the following DDNAMES:

   EQACRINP
   Code coverage extracted observations that are based on selection criteria.

   EQACRSEL
   Code coverage Selection file.

   EQACROUT
   XML report output.

An example of using EQAXCCR2 in batch to generate a XML report can be found in *hlq*.SEQASAMP(EQACCXRP).

In addition to the XML report, you can also generate a Presentation report. This is generated by selecting DTU Option E.5 sub-option 2 or 3. In batch specify the PFMT parameter. An example of using EQAXCCR2 in batch to generate a Presentation report can be found in *hlq*.SEQASAMP(EQACCPRP).

Code coverage Viewer

The following figure shows the input to code coverage Viewer. The Viewer displays the results of a z/OS Debugger Code Coverage session. It takes as input either the code coverage Observation files first created by z/OS Debugger or the code coverage extracted Observation file, that is the one created after you apply the selection criteria in *Step 2. Code coverage selection and extraction process*. With the Viewer, you can display all the entries in either data set. You can sort the entries and view an annotated listing that is associated with an entry.

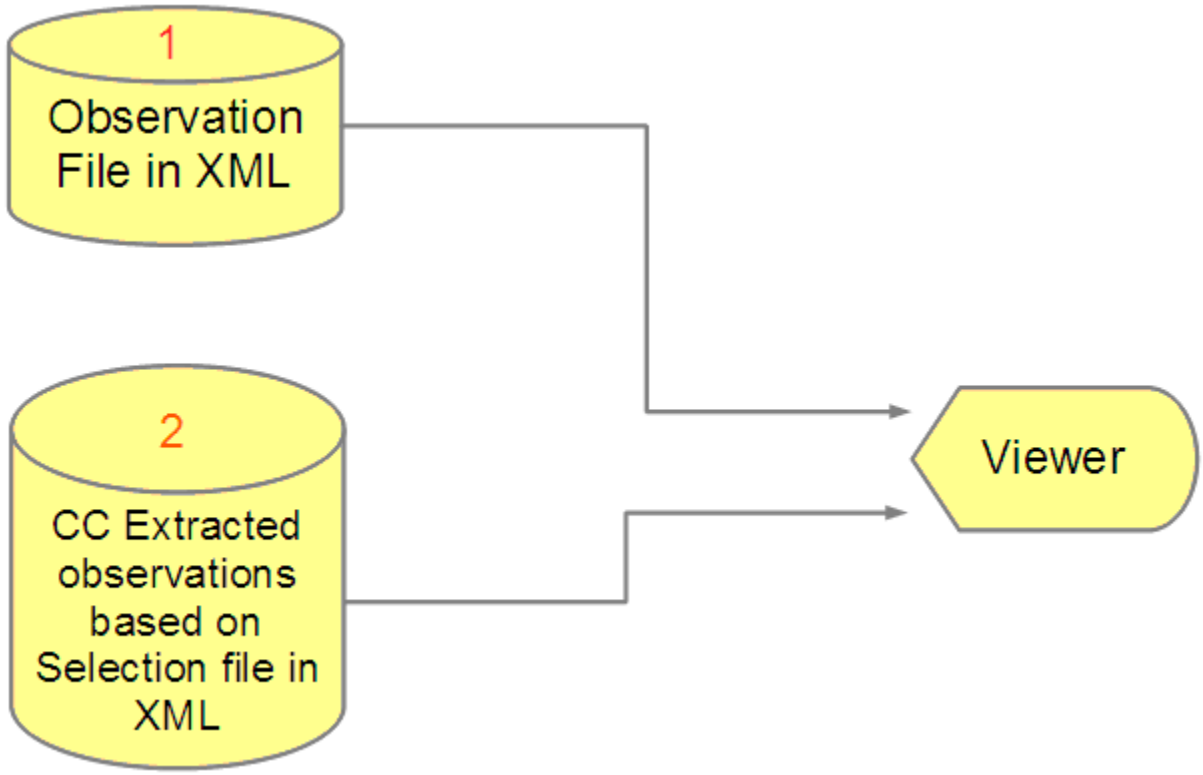


Figure 6. Step 4 The Viewer

- The Viewer is part of IBM z/OS Debugger Utilities. It is Option E.1 and allows the user to analyze the code coverage observations interactively.
- The Viewer processes either the Observation file created by z/OS Debugger (1 in the figure) or the code coverage extracted Observation file created by the code coverage Extraction Utility (2 in the figure).
- When you first select Option E.1, you are prompted to provide the name of the file that you want the Viewer to use.
- The Viewer provides the following functionality:
  - Sorting entries.
  - Viewing an annotated listing associated with an entry. When you are viewing an annotated listing, no selection criteria is applied. Every line of the listing is included and marked as executed or unexecuted as specified in the observation.

Code coverage by using z/OS Debugger

Setup

Preparing your program

One of the benefits of using this approach to create code coverage observations is that you can use the same load modules that you prepare for debugging your application with z/OS Debugger. Programs written in COBOL, PL/I and C and compiled with certain compiler options are supported.

Code Coverage is supported for Enterprise COBOL for z/OS and OS/390 Version 3 and Enterprise COBOL for z/OS Version 4, 5, and above. The following compiler options are required to ensure that the SYSDEBUG side file or program object contains the program source:

- Enterprise COBOL for z/OS and OS/390 Version 3 - TEST(SEPARATE) with NONE recommended but not required.

- Enterprise COBOL for z/OS Version 4 - TEST(SEPARATE) with NOHOOK recommended but not required.
- Enterprise COBOL for z/OS Version 5 and 6.1 - TEST(SOURCE), Version 6.2 and later - TEST(NOSEPARATE,SOURCE).

Code Coverage is supported for Enterprise PL/I for z/OS Version 4.2 and above in 31-bit mode. The following compiler options are required to ensure that the SYSDEBUG side file contains the complete expanded program source and statement table:

- TEST(SEPARATE) - the ALL and NOHOOK sub-options are also recommended but not required.
- GONUMBER(SEPARATE) - required to produce the statement table in the SYSDEBUG side file.
- MACRO or PP(MACRO) - required if there are %INCLUDE statements in the source. Using the MACRO suboption CASE(ASIS) will leave the case of the source unchanged.
- LISTVIEW(AFTERALL) - required if include files, EXEC CICS commands, or SQL code are in the source.

Code Coverage is supported for IBM z/OS XL C. The following compiler options and program preparation are required:

- You must run the following 2-stage compile process.

  The first stage preprocesses the program, so the IBM z/OS Debugger has access to fully expanded source. The second stage compiles the program.
- The first compile stage specifies compiler options PP(COMMENTS,NOLINES) to expand INCLUDEs and macros. The output is SYSUT10 DD. SYSUT10 DD is the expanded source file and is the input for the second compiler stage. Modify the SYSUT10 DD to enable z/OS Debugger, by saving it in a expanded source library and specify a member name that is equal to the primary entry point name or CSECT name of your application program.
- For the second compiler stage, use the DEBUG(FORMAT(DWARF)) option to place the debug data in a separate file in one of these ways:

  Use DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL,FILE(location)), or for better performance, use DEBUG(FORMAT(DWARF),NOHOOK,SYMBOL,FILE(location)).
- You cannot use an .mdbg file.
- You cannot use DEBUG(FORMAT(ISD)) or TEST.
- You cannot perform source extraction of a source stored on an HFS or zFS file.

For a full description of the compilers and the options, see Part 2, “Preparing your program for debugging,” on page 21.

EQAOPTS commands

EQAOPTS commands are used to provide data set names for the XML output and a list of program names that require code coverage.

CCOUTPUTDSN
Specifies the file name of an MVS sequential data set. The file contains code coverage output in XML format.

A write-only data set is created if required, opened for appending at z/OS Debugger termination, written with code coverage data collected, and then closed and freed.

CCOUTPUTDSNALLOC
Specifies the allocation parameters in BPXWDYN format if a new CCOUTPUTDSN data set is to be created.

CCPROGSELECTDSN
Specifies the file name of an MVS sequential data set. The data set contains a list of compile unit names and is normally created and edited with DTU option E.2. Code coverage data is collected when these compile units are run. The program name in the list can contain a wildcard; for example, PRG1* specifies that code coverage data is collected for all programs whose names begin with PRG1.

The data set is read-only and opened at the start of z/OS Debugger. After the program list is read, the file is closed and freed.

Example:

```
EQAXOPT  CCOUTPUTDSN,'&&USERID.DBGTOOL.CCOUTPUT'
EQAXOPT  CCOUTPUTDSNALLOC,'MGMTCLAS(STANDARD)                 +
      STORCLAS(DEFAULT) LRECL(255) BLKSIZE(0) RECFM(V,B)      +
      DSORG(PS) SPACE(2,2) CYL'
EQAXOPT  CCPROGSELECTDSN,'&&USERID.DBGTOOL.CCPRGSEL'
```

Notes:

- You can find this example in *hlq*.SEQASAMP(EQACCOPT).
- EQAOPTS commands must be contained in fixed-length, eighty-byte records.
- The continuation character "+" is in column 72.

For more information about EQAOPTS commands, see the chapter about EQAOPTS commands in *IBM z/OS Debugger Reference and Messages*.

EQA_STARTUP_KEY

The *EQA_STARTUP_KEY* is an environment variable. The format for specifying this environment variable is as follows: ENVAR("EQA_STARTUP_KEY=ACTION").

The values for the ACTION parameter are as follows:

CC
An unattended z/OS Debugger Code Coverage session is requested. In this case, an interactive debug session is not launched.

DCC
A combined z/OS Debugger session and Code Coverage session is requested. This allows the developer to have a debug session and concurrently create code coverage data. If you use this option and change the program logic path by using the GOTO and JUMPTO commands, the observation is flagged indicating that the debug override is ON.

z/OS Debugger uses the EQA_STARTUP_KEY environment variable and TEST runtime options to determine whether to activate an interactive debug session and code coverage session or not. The following table shows different combinations of the environment variable and TEST runtime options, and the resultant session activation.

Note: There are two different code coverage sessions: z/OS Debugger code coverage session and IBM Developer for z/OS code coverage session. z/OS Debugger handles the z/OS Debugger code coverage session. IBM Developer for z/OS handles the IBM Developer for z/OS code coverage session. The code coverage data format and presentation are different in the two sessions.

EQA_STARTUP_KEY	z/OS Debugger session device	z/OS Debugger interactive debug session	z/OS Debugger code coverage session	IBM Developer for z/OS code coverage session
CC	MFI (no 3270 terminal available[1])	No	Yes	No
CC	TCPIP/DBMDT	No	No	Yes
DCC	MFI/VTAM (target 3270 terminal provided)	Yes	Yes	No
DCC	TCPIP/DBMDT	Yes	Yes	No

Notes:

1. For CICS, the "no 3270 terminal available" restriction is bypassed if you follow the example in “Generating code coverage for CICS transactions” on page 485.

Examples:

- `'/TEST(ALL,*,PROMPT,MFI:*),ENVAR("EQA_STARTUP_KEY=CC")'`
  - Using DT MFI, and specifying CC.
  - Code Coverage observations are collected.
- `'/TEST(ALL,*,PROMPT,VTAM%userid:*),ENVAR("EQA_STARTUP_KEY=DCC")'`
  - Using z/OS Debugger MFI with the Terminal Interface Manager, and specifying DCC.
  - An interactive debug session is started and Code Coverage observations are collected while it is running.
- `'/TEST(ALL,*,PROMPT,TCPIP&nn.nn.nn.nn%8001:*),ENVAR("EQA_STARTUP_KEY=DCC")'`
  - Using z/OS Debugger TCPIP with IBM Developer for z/OS, and specifying DCC.
  - An interactive debug session is started, and Code Coverage observations are collected while the debug session is running.

Code coverage Options data set

The code coverage Options file contains information that is provided as input to the z/OS Debugger Code Coverage engine. The file contains the following XML tags. You can manually code the tags or use DTU option E.2 to create them.

- <GROUPID1>: Group ID 1

  If you want to group observations to form a set based on the characteristics of the applications, you can use this tag.
- <GROUPID2>: Group ID 2

  If you want a subgroup for the observation to form a subset based on the characteristics of the application, you can use this tag. During the analysis of the observations, the user can sort based on the grouping.
- <EXTNAME>: Name of the program (COBOL PROGRAM-ID, PL/I external procedure name or C short CU name) that is targeted for code coverage.

  You can use a wildcard (*) either at the end of the name string, or you can use only the wildcard if you want all programs in the application to be covered. The DTU option E.2 panel allows up to 8 names. You can hand code more in the Options data set if you need.

Here is an example of an Options file in XML rendering. In this example, the Options file indicates that z/OS Debugger collects code coverage observations for programs COB01A, COB01B, COB01C, and COB01D. z/OS Debugger marks the observations as part of GROUP ID 1 PAYROLL and GROUP ID 2 TEST02.

```
<GROUPID1>PAYROLL</GROUPID1>
<GROUPID2>TEST02</GROUPID2>
<EXTNAME>COB01A</EXTNAME>
<EXTNAME>COB01B</EXTNAME>
<EXTNAME>COBO1C</EXTNAME>
<EXTNAME>COBO1D</EXTNAME>
```

Generating code coverage extracted observations

Depending on the values that are provided in the Options file, z/OS Debugger gathers observations for all statements in the programs in the Options data set. The number of observations can be large, and depends on the number of programs and the statements in the programs.

To facilitate the evaluation of the observations, z/OS Debugger Code Coverage provides a mechanism to define a subset of the observations in the final report. This is done by providing a selection mechanism that allows you to only include in the report the extracted observations for those programs that you are interested in.

You can specify how z/OS Debugger selects such programs by providing a Selection file. You can create the Selection file by using IBM z/OS Debugger Utilities Option E.3 or by manually coding the file by following the Selection file XML DTD syntax.

Code Coverage selection data set

You use the selection data set to specify the criteria that is used in the evaluation of the code coverage observations to create a extracted observations data set and a set of statistics based on the selection provided. For example, you might want to see only the results for a specific group, or a specific program even if the Options data set indicated more than one program. This allows the user to define the granularity of the information.

There are two different types of selection criteria attributes. The first group selects the entries that are to be extracted from the observation data set that is created by z/OS Debugger. The other group operates from within the subset that is created after applying the first group of attributes. The second set of attributes is designed for further selection of the statements to be considered in the final statistical results based on the contents of the program source.

Observation selection criteria

The selection criteria is based on the attribute values of a code coverage observation. You can specify one or more attribute values and their associated comparison operators.

The comparison operators include: equal (E), greater than (G), less than (L), greater than or equal (GE), less than or equal (LE), and not equal (NE). If no value is entered for an attribute, it means that any value is valid and the selection process does not examine the attribute.

The following screen shows a list of observation attributes, comparison operators, and roll-up options that you can specify to select only the entries that you are interested in when you generate the code coverage extracted observations.

```
Attribute name            Value   Operator             Roll-up
------------------------ ------- ------------------  -------
Run date (YYYY/MM/DD)               (E,G,L,GE,LE,NE)
Run time (HH:MM:SS)                 (E,G,L,GE,LE,NE)
Group ID 1                COST     E (E,NE)            N (Y/N)
Group ID 2                BENEFIT  E (E,NE)            N (Y/N)
User ID                   USERIBM   E (E,NE)            Y (Y/N)
Load module name                    (E,NE)
Program name              COB01*   E (E,NE)
Compile date (YYYY/MM/DD)           (E,G,L,GE,LE,NE)
Compile time (HH:MM:SS)             (E,G,L,GE,LE,NE)
Debug override                      (E,NE)               (Y/N)
Total statements                    (E,G,L,GE,LE,NE)
Executed statements                 (E,G,L,GE,LE,NE)
```

Source statement selection

The source statement selection is used to select source statements based on special indicators in the source that indicate the lines that have been modified or added since the last check-in or promotion of the program source. You can define source markers to specify that the source line with the special indicator be included or excluded when the code coverage percentage is calculated.

Source markers

The source markers provide a way to select the source lines that are to be marked in the report file and called out in the statistics calculation for code coverage. These are based on the indicators in the source like a comment, numeric sequence, a range of statements, and a string at a specific place in the source

listing. An indicator marks a statement or section of statements that have been changed or added as a result of a defect fix or enhancement. A source marker definition consists of the following elements:

Marker type
Single source line or a section of source lines

- SINGLE
- SECTIONBEGIN
- SECTIONEND

Selection
INCLUDE or EXCLUDE

Start column
Marker search starts at this column in a source line

End column
Marker search ends at this column in a source line

Indicator
Character (*xxxx*) or hex (*X'nnnn'*)

Note:

- Multiple markers can be defined.
- Section source markers must come in pairs, such as SECTIONBEGIN and SECTIONEND.

The following table shows a sample of source markers:

Table 27. A sample of source markers

Marker type	Selection	Start column	End column	Indicator
SINGLE	INCLUDE	73	80	PMR12345
SINGLE	EXCLUDE	7	72	MOVE
SECTIONBEGIN	INCLUDE	7	80	DEFECT123BEGIN
SECTIONEND	INCLUDE	7	80	DEFECT123END

The first entry in the table indicates to mark as included in the report file and call out in the statistics calculation only statements that have the string PMR12345 in columns 73 - 80.

The second entry in the table indicates to mark as excluded in the report file and call out in the statistics calculation only statements that have the string MOVE in columns 7 - 72.

The third and fourth entries in the table indicate to mark as included only the statements beginning with the first statement that has the string DEFECT123BEGIN between columns 7 - 80 until the statement that has the string DEFECT123END between columns 7 - 80.

The following example corresponds to the values in the above table.

```
<SOURCEMARKER>
<MARKERTYPE>SINGLE</MARKERTYPE>
<SELECTION>INCLUDE</SELECTION>
<STARTCOLUMN>73</STARTCOLUMN>
<ENDCOLUMN>80</ENDCOLUMN>
<MARKERVALUE>C'PMR12345'</MARKERVALUE>
</SOURCEMARKER>
<SOURCEMARKER>
<MARKERTYPE>SINGLE</MARKERTYPE>
<SELECTION>EXCLUDE</SELECTION>
<STARTCOLUMN>7</STARTCOLUMN>
<ENDCOLUMN>72</ENDCOLUMN>
<MARKERVALUE>C'MOVE'</MARKERVALUE>
</SOURCEMARKER>
<SOURCEMARKER>
<MARKERTYPE>SECTIONBEGIN</MARKERTYPE>
<SELECTION>INCLUDE</SELECTION>
```

```
<STARTCOLUMN>7</STARTCOLUMN>
<ENDCOLUMN>80</ENDCOLUMN>
<MARKERVALUE>DEFECT123BEGIN</MARKERVALUE>
</SOURCEMARKER>
<SOURCEMARKER>
<MARKERTYPE>SECTIONEND</MARKERTYPE>
<SELECTION>INCLUDE</SELECTION>
<STARTCOLUMN>7</STARTCOLUMN>
<ENDCOLUMN>80</ENDCOLUMN>
<MARKERVALUE>DEFECT123END</MARKERVALUE>
</SOURCEMARKER>
```

Based on the Selection options in the example above, the report marks the sections of the source that matches the specified selection.

Source marker use case example

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7-|--+----8
00001 * ACCESS BY LOW LEVEL QUALIFIERS
00002      MOVE 'KY' TO STATE                                          PMR12345
00003      MOVE 'LEX' TO CITY
00004      MOVE 'VM ' TO OP-SYS
00005   PROGA.
00006      PERFORM LOOP1 UNTIL TAPARM1 = 0                             PMR12345
00007 * DEFECT123BEGIN
00008      IF TAPARM2 = 0 THEN
00009      PERFORM PROCA.
00010 * DEFECT123END
```

Based on the sample source markers above, the report shows the lines as follows:

- Line 2 is both included and excluded
- Lines 3 and 4 are excluded
- Lines 6, 8, and 9 are included

The Selection file can be created by using IBM z/OS Debugger Utilities Option E.3, or you can code the file manually by following the Selection file XML DTD syntax.

IBM z/OS Debugger Utilities Option E

The z/OS Debugger Code Coverage option in IBM z/OS Debugger Utilities provides facilities to complete the following tasks:

- View the observations and sort them
- Create and modify the Options file and the Selection file
- Extract observations after you apply Selection file
- Report creation

It includes five suboptions to perform such tasks.

- Use Code Coverage observation viewer to sort and view code coverage observations.
- Use Code Coverage Options file to create/modify the Options file.
- Use Code Coverage observation Selection file to create/modify the Selection file.
- Use Code Coverage observation selection to extract code coverage observations based on selection criteria.
- Use Code coverage report generation to create reports.

The following screen shows IBM z/OS Debugger Utilities Option E suboptions.

```
------------------------- z/OS Debugger
Code Coverage -------------------------
Option ===>

1  Observation viewer
   Browse code coverage observations.

2  z/OS Debugger options
   Create or modify the z/OS Debugger
code coverage options.

3  Observation selection criteria
   Create or modify the observation selection criteria and source markers.

4  Observation extraction
   Extract code coverage observations using selection criteria.

5  Report generation
   Create report.
```

Option E.1 Code Coverage Observation Viewer

The Viewer is Option E.1 in IBM z/OS Debugger Utilities. You can view the XML entries of the code coverage observations in a table format that facilities analysis. The Viewer uses either the original file of observations that were created by z/OS Debugger during a code coverage session, or the extracted Observation file after you apply the selection criteria.

After you select suboption 1, you are prompted to provide the name of the data set for the code coverage observations that you want to view. The following screen shows the panel with the name of the data set that the viewer uses.

```
------------- z/OS Debugger - Code Coverage Observation Viewer ---------------
Command ===>


Specify the name of a code coverage observation data set that you
want to browse.

The code coverage observation data set contains code coverage
observations generated from a z/OS Debugger
Code Coverage session.


Data Set Name:
  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT'
  Volume Serial . . .           (If not cataloged)


Press Enter to continue.
Press Exit or Cancel to exit.
```

After you specify the location of the file, press enter to move to the viewer where you can view the observations in table format. The following screen shows the entries for a group of observations. The viewer provides the following functions:

- Sort table entries.
- View an annotated listing that is associated with an entry.

```
------------ z/OS Debugger - Code Coverage Observation Viewe Row 1 to 6 of 11
Command ===>                                                 Scroll ===> PAGE

  Enter / to sort the table entries.

Enter (V)iew table entry command to view source listing.

  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB01           Prog Name:  COB01A
  Comp Date:  2013/05/07      Comp Time:  15:53:00        Debug override: N
  Tot Stmts:  17              Exec Stmts: 15              Percent:   88.23%
--------------------------------------------------------------------------
  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB01           Prog Name:  COB01B
  Comp Date:  2013/05/07      Comp Time:  15:53:00        Debug override: N
  Tot Stmts:  10              Exec Stmts: 9               Percent:   90.00%
--------------------------------------------------------------------------
  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB01           Prog Name:  COB01C
  Comp Date:  2013/05/07      Comp Time:  15:53:00        Debug override: N
  Tot Stmts:  14              Exec Stmts: 12              Percent:   85.71%
--------------------------------------------------------------------------
  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB02           Prog Name:  COB02A
  Comp Date:  2013/04/30      Comp Time:  10:51:00        Debug override: N
  Tot Stmts:  27              Exec Stmts: 24              Percent:   88.88%
--------------------------------------------------------------------------
  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB02           Prog Name:  COB02C
  Comp Date:  2013/04/30      Comp Time:  10:51:00        Debug override: N
  Tot Stmts:  14              Exec Stmts: 12              Percent:   85.71%
--------------------------------------------------------------------------
  Run Date :  2013/05/14      Run Time:   16:41:05
  Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
  Load Name:  COB31M          Prog Name:  COB31M
  Comp Date:  2013/04/29      Comp Time:  16:25:00        Debug override: N
 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

When you enter a forward slash (/) in the **sort the table entries** field, you are prompted with a pop-up panel where you can choose the sorting options to sort the table entries to your specifications. The following screen shows the Table Sort Pop-up panel.

```
 You can sort the table entries using column key number
 (1 - 13) and sort sequence (A or D).

 Attribute name          Key order  Sort sequence
 Run date                    1           A
 Run time                    2           A
 Group ID 1
 Group ID 2
 User ID
 Load module name            3           A
 Program name                4           A
 Compile date
 Compile time
 Debug override
 Total statements
 Executed statements
 Percent

  F1=Help       F2=Split      F3=Exit       F7=Backward
  F8=Forward    F9=Swap      F12=Cancel
```

You can view the annotated source listing for a program when it is available by entering V next to an entry. The source for the program associated with the entry is displayed. The source is annotated to show the code coverage. See “Annotated listing format” on page 479.

Option E.2 Code Coverage Options file

In this option, you can create the Options file that is used as an input to z/OS Debugger at the start of a code coverage session. In this file, you can specify the programs that you are interested in when the code coverage observations are collected. The information that you provide is then converted to XML format. As mentioned before, you can create this file yourself by hand coding the options following the Options file XML DTD syntax (See "XML Tags used in the Options file" on page 489).

After you choose Option E.2, you are prompted to provide the location of the file that will be used to save the Options. If the file has not been previously created, it will be created for you. In the following screen, you can see this panel.

```
---------------------- z/OS Debugger - Code Coverage Options ---------------------
Command ===>


Specify the name of a code coverage options data set name that you
want to create or edit.

The data set contains a list of program names and group IDs that are
used when collecting code coverage observations.


Data Set Name:
  Data Set Name . . . 'USERIBM.DBGTOOL.CCPRGSEL'
  Volume Serial . . .           (If not cataloged)


Press Enter to edit the data set.
Press Exit or Cancel to exit.
```

After the Options file is created, you can proceed to the Options panel. You can specify the programs that you want code coverage observations for and the group or subgroup to use to group such results. The panel is tailored after other z/OS Debugger panels that are used for creating debug profiles. The following screen shows the Options panel. You have two sections in this panel:

- Program selection.
  - In this section, you can specify up to 8 programs or you can use an asterisk (*) instead of a program name or you can end the name of a program with an asterisk (*) to create a template for a group of programs with the same prefix in the name.
- Group selection.
  - You can use Group ID 1 and Group ID 2 for grouping results.
  - If you want to provide a group during the observation selection, you should specify a group. If the group is in the Viewer, you can sort the entries in the Viewer by the group.
  - You can use a wildcard (*) or leave it blank if you do not want to use a group.

```
------------------- z/OS Debugger - Edit Code Coverage Options -----------------
Command ===>

Program name list for code coverage. (* is a valid wild card character,
  by itself, or as the last character of a name)

Name 1: COB01*      Name 2: COB02*      Name 3: IGYTCARA    Name 4:
Name 5:             Name 6:             Name 7:             Name 8:


Group ID is a container ID that allows you to catalog code coverage
observations.

Group ID 1:      COST
Group ID 2:      BENEFIT
```

After you exit this panel, the Options file is written with your options using the Options file XML DTD syntax (See "XML Tags used in the Options file" on page 489). As mentioned before, you can skip the use of Option E.2 and hand code the contents of the file.

Option E.3 Code Coverage observation Selection file

In this option, you can specify the selection criteria that you want to use to extract only the observations that you are interested in. When you first select this option, you provide the name of the data set that contains the Selection file. The following screen shows this panel.

```
---------- z/OS Debugger - Code Coverage Observation Selection Criteria ---------
Command ===>


Specify the name of a code coverage observation selection criteria
data set that you want to create or edit.

The data set contains selection criteria and source markers used to select
code coverage observations and percentage calculations.


Data Set Name:
  Data Set Name . . . 'USERIBM.DBGTOOL.CCOBSSEL'
  Volume Serial . . .           (If not cataloged)


Press Enter to edit the data set.
Press Exit or Cancel to exit.




 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

After you provide the name of the data set, press Enter to create or modify your Selection file. The following screen shows the selection attributes panel.

```
------------- z/OS Debugger - Edit Code Coverage Selection Criteria -------------
Command ===>

Specify code coverage observation selection criteria

Enter attribute value and comparison operator. Comparison operators
are (E)qual, (G)reater, (L)ess, (GE) greater than or equal,
(LE) less than or equal, and (NE) not equal.

Attribute name            Value        Operator             Rollup
Run date (YYYY/MM/DD)                     (E,G,L,GE,LE,NE)
Run time (HH:MM:SS)                       (E,G,L,GE,LE,NE)
Group ID 1                COST         E  (E,NE)            N (Y/N)
Group ID 2                BENEFIT      E  (E,NE)            N (Y/N)
User ID                   USERIBM       E  (E,NE)            Y (Y/N)
Load module name                          (E,NE)
Program name              COB01*       E  (E,NE)
Compile date (YYYY/MM/DD)                 (E,G,L,GE,LE,NE)
Compile time (HH:MM:SS)                   (E,G,L,GE,LE,NE)
Debug override                            (E,NE)               (Y/N)
Total statements                          (E,G,L,GE,LE,NE)
Executed statements                       (E,G,L,GE,LE,NE)


Specify source markers for code coverage percentage analysis

Marker type: SINGLE/SECTIONBEGIN/SECTIONEND
Selection:   INCLUDE/EXCLUDE

Marker type   Selection  Column Column String
                         Start  End
SINGLE        INCLUDE    73     75     PMR
SINGLE        EXCLUDE    73     80     PMR11114
SECTIONBEGIN  INCLUDE    7      80     DEFECT123BEGIN
SECTIONEND    INCLUDE    7      80     DEFECT123END


 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

The marker section allows only five markers to be specified. If you need more than five, you need to add the additional entries by hand using the Selection file XML DTD syntax (See “XML tags used in the Selection file” on page 489).

Most of the field above are self-explanatory or have been described before in this document. The following section describes the operators and the meaning of the Roll-Up fields.

Operators

E = Equal

G = Greater than

L = Less than

GE = Greater or Equal

LE = Less or Equal

NE = Not Equal

Roll-up

The roll-up is a merge process. The selected observations are grouped into subgroups with all observations that have the same load module name, program name, compile date and compile time. The roll-up is then performed within each subgroup and is based on four other attributes of the observations. Each of the four attributes has a 'roll-up' option with value Yes or No. If Yes, it means that the observations are qualified for merge when the attributes are the same or different. If No, it means that observations with different values of the attribute cannot be merged. However, if they have the same value, they are qualified. The test is performed on each of the four attributes. All tests must be positive before the merge takes place. The attributes of a observation that has the roll-up option are GroupID1, GroupID2, User ID, and DBGOV (Debug override). The merge of qualified observations is to combine the executed statement lists together for generating the code coverage extracted observations.

In the resultant observation after the merge process, the attributes that have the roll-up option = 'Y' show a value of '*' except the DBGOV attribute. This attribute shows a value of 'Y' if at least one of the merged observations has the DBGOV attribute = 'Y'. It shows a value of 'N' when all the merged observations have the DBGOV attribute = 'N'.

The qualified observations might come from different test cases; the executed statement lists might overlap; and, by combining together, the code coverage percentage might be improved.

Roll-up use case example

You can define the roll-up option of the four attributes as follows:

```
Attribute            Rollup option
----------           -------------
GroupID1               Y
GroupID2               Y
UserID                 Y
DbgOv                  Y
```

Here are two selected observations based on the selection criteria:

```
# GrpID GrpID User Lmod CSECT Comp       Comp     DO tot  exec  %
  ID1   ID2   ID        Name  Date       Time        stmt stmt
- ----- ----- ---- ---- ----- ---------- -------- -- ---- ---- ----
1 Pay1  Test1 USERIBM LMD1 PRG1  2013/04/08 10:10:20 Y  100  80   80%
2 Pay1  Test2 USERIBM LMD1 PRG1  2013/04/08 10:10:20 N  100  50   50%
```

The roll-up process merges #1 and #2 together even when the values of GroupID2 and DbgOv are different because the roll-up option of the two attributes is Yes.

After the two observations are merged, the code coverage percentage becomes 90% because the executed statements in #1 and #2 overlap.

Option E.4 Code Coverage observation extraction

With this option, you can create a file that contains the results from applying the selection file to the file that contains all observations created by a z/OS Debugger Code Coverage session. When you select this panel in the following screen, you are prompted to provide the following files:

- Input
  - The location of the file with the code coverage observations
  - The location of the file with the selection criteria
- Output
  - The location of the file that contains the extracted code coverage observation output

```
--------------- z/OS Debugger - Code Coverage Observation Selecton --------------
Command ===>


The observation selection function extracts observations that meet the
selection criteria from the observation data set. It writes the result
to the observation output data set.

Specify the name of a code coverage observation data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT'

Specify the name of a code coverage selection criteria data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage observation output data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT.SELECTED'

Press Enter to continue.
Press Exit or Cancel to exit.



 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

After you press Enter, you will get a confirmation message on the upper right corner, 'Observation extract OK'. If there is an error during the process, an error message is displayed. By pressing F1, a long message appears at the bottom of the panel.

```
--------------- z/OS Debugger - Code Coverage Observatio  Extract observations OK
Command ===>


The observation selection function extracts observations that meet the
selection criteria from the observation data set. It writes the result
to the observation output data set.

Specify the name of a code coverage observation data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT'

Specify the name of a code coverage selection criteria data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage observation output data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT.SELECTED'

Press Enter to continue.
Press Exit or Cancel to exit.



 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

Option E.5 Code Coverage report generation

When you select this option, a panel is displayed with three choices for the type of report that you want to create:

- Create report in XML format.
- Create report in Presentation format.
- Create and browse report in Presentation format.

In the same panel, you must provide the following information:

- The location of the code coverage extracted observation data set

- Code coverage selection criteria data set
- The location of output code coverage report data set

The following screen shows the Code Coverage Report Generation panel:

```
---------------- z/OS Debugger - Code Coverage Report Generation ----------------
Command ===>


The report generator adds marked source statements and code coverage
statistics to the extracted observations. It writes the result
to the report output data set along with the selection criteria.

Select a report action.

   1. Create report in XML format
   2. Create report in Presentation format
   3. Create and browse report in Presentation format

Specify the name of a code coverage extracted observation data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT.SELECTED'

Specify the name of a code coverage selection criteria data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage report data set.

  Data Set Name . . . 'USERIBM.DBGTOOL.CCOUTPUT.SELECTED.REPORT'

Press Enter to continue.
Press Exit or Cancel to exit.



 F1=Help      F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
 F12=Cancel
```

Annotated listing format

There are three formats of annotated listings:

Observation viewer
: The View table entry command builds an annotated listing in a temporary data set and internally issues the view command against that data set. The data set is deleted when view exits. Unlike the annotated listings described below, a viewed annotated listing is not subject to selection criteria. This means that the only annotation performed is marking the statements as executed or unexecuted. In other words, there are no included or excluded statements to annotate. Also because of this, the statistics in the viewed annotated listing lack the granularity of the statistics provided in the other annotated listings.

XML Report
: This creates an annotated listing with additional annotation (markers) for included and excluded lines. Each line in the source listing is encapsulated in <STMT> and </STMT> XML tags. The selection criteria source markers and statistics are encapsulated in their own XML tags as is the observation data and SYSDEBUG compile date and time.

Presentation Report
: The presentation format annotated listing is more viewer friendly and is nearly identical to an XML format report without the XML tags. The selection criteria source markers, the statistics, and the observation data are included in the tables that follow the annotated listing. The report also indicates whether the SYSDEBUG compile date and time does not match the compile date and time that is recorded in the observation.

The annotated listing begins with a table of information about the observations that is similar to an entry in the Viewer. After that, the source listing is displayed with annotation showing which executable lines were executed or not executed. XML and presentation reports also contain additional annotation for

included and excluded lines (as indicated by the source markers in the Selection file). The result is written to the specified output data set. If option 3 was requested, the output data set is browsed via ISPF but not deleted upon exit.

Below is a sample of a presentation format annotated listing report for a COBOL program. There are 8 header lines including 2 blank ones, the rollup history, a number of source lines, the selection criteria source markers, and the statistics. The header lines indicates the observation for which the report is generated. The rollup history indicates the origin of the observation.

```
1Rpt Date :  2013/05/11      Rpt Time:   10:32:45

 Run Date :  2013/05/10      Run Time:   09:22:49
 Group ID 1: COST            Group ID 2: BENEFIT         User ID:    USER1
 Load Name:  COB01           Prog Name:  COB01A
 Comp Date:  2013/05/07      Comp Time:  15:53:00        Debug override: N
 Tot Stmts:  17              Exec Stmts: 15              Percent:   88.23%

Rollup History:
Observation is not part of rollup.

               ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8
     1               * COB01A - COBOL EXAMPLE FOR DTCU
     2
     3                IDENTIFICATION DIVISION.
     4                PROGRAM-ID. COB01A.
     5               ********************************************************************
     6               * Licensed Materials - Property of IBM                            *
     7               *                                                                  *
     8               * 5655-M18: Debug Tool for z/OS                                   *
     9               * 5655-M19: Debug Tool Utilities and Advanced Functions           *
    10               * (C) Copyright IBM Corp. 1997, 2004 All Rights Reserved          *
    11               *                                                                  *
    12               * US Government Users Restricted Rights - Use, duplication or     *
    13               * disclosure restricted by GSA ADP Schedule Contract with IBM     *
    14               * Corp.                                                           *
    15               *                                                                  *
    16               ********************************************************************
    17
    18
    19                ENVIRONMENT DIVISION.
    20
    21                DATA DIVISION.
    22
    23                WORKING-STORAGE SECTION.
    24                01  TAPARM1      PIC 99 VALUE 5.
    25                01  TAPARM2      PIC 99 VALUE 2.
    26                01  COB01B       PIC X(6) VALUE 'COB01B'.
    27                01  P1PARM1      PIC 99 VALUE 0.
    28
    29                01  TASTRUCT.
    30                  05 LOC-ID.
    31                     10 STATE    PIC X(2).
    32                     10 CITY     PIC X(3).
    33                  05 OP-SYS     PIC X(3).
    34
    35                PROCEDURE DIVISION.
    36
    37               * THE FOLLOWING ALWAYS PERFORMED
    38
    39               * Defect456Begin
    40
    41                PROG.                                                      PMR11112
    42               * ACCESS BY TOP LEVEL QUALIFIER                             PMR11112
    43 I>                 MOVE 'ILCHIMVS' TO TASTRUCT                           PMR11112
    44
    45               * ACCESS BY MID LEVEL QUALIFIERS                            PMR11113
    46 I>                 MOVE 'ILSPR' TO LOC-ID                                PMR11113
    47 I>                 MOVE 'AIX' TO OP-SYS                                  PMR11113
    48
    49               * ACCESS BY LOW LEVEL QUALIFIERS                            PMR11114
    50 B>                 MOVE 'KY' TO STATE                                    PMR11114
    51 B>                 MOVE 'LEX' TO CITY                                    PMR11114
    52 B>                 MOVE 'VM ' TO OP-SYS                                  PMR11114
    53                    .                                                     PMR11114
    54
    55                PROGA.
    56  >                 PERFORM  LOOP1 UNTIL TAPARM1 = 0
    57
    58  >                 IF TAPARM2 = 0   THEN
    59               *   PROCA NOT EXECUTED                                     PMR12345
    60 I<                     PERFORM PROCA.                                    PMR12345
    61
    62
    63 I>                 PERFORM LOOP2 UNTIL TAPARM2 = 0                       PMR12345
    64                    .
    65  >                 STOP RUN
    66                    .
    67
    68                PROCA.
```

```
    69               *   PROCA NOT EXECUTED                                      PMR12345
    70 I<                 MOVE 10 TO P1PARM1                                    PMR12345
    71                    .                                                     PMR12345
    72                LOOP1.
    73  >                   IF TAPARM1 > 0 THEN
    74  >                     SUBTRACT 1 FROM TAPARM1.
    75  >                   CALL 'COB01B'
    76                    .
    77                LOOP2.                                                    PMR12345
    78 I>                  IF TAPARM2 > 0 THEN                                  PMR12345
    79 I>                    SUBTRACT 1 FROM TAPARM2.                           PMR12345
    80
    81               * Defect456End

                          Start  End
  Marker type   Selection  Column Column String
  ------------ ---------  ------ ------ ------------------------------
  SINGLE        INCLUDE        73     75 PMR
  SINGLE        EXCLUDE        73     80 PMR11114
  SECTIONBEGIN  INCLUDE         7     80 DEFECT123BEGIN
  SECTIONEND    INCLUDE         7     80 DEFECT123END

            Statements  Executed  Percentage
            ----------  --------  ----------
  Total             17        15       88.23
  Included           8         6       75.00
  Excluded           0         0        0.00
  Incl/Excl          3         3      100.00
```

Below is a sample of a presentation format annotated listing report for a PL/I program. The format is similar to other supported languages.

```
1Rpt Date :  2013/09/10      Rpt Time:   11:53:14

 Run Date :  2013/09/02      Run Time:   12:31:30
 Group ID 1: *               Group ID 2: BENEFIT         User ID:   USER1
 Load Name:  PLI01           Prog Name:  PLI01A
 Comp Date:  2013/09/02      Comp Time:  12:14:00   Debug override: N
 Tot Stmts:  14              Exec Stmts: 11              Percent:   78.57%

 Rollup History:

 Group ID 1      Group ID 2      Load Name       Prog Name
 --------------- --------------- --------------- ---------------
 COST            BENEFIT         PLI01           PLI01A
 COST            BENEFIT         PLI01           PLI01A

                ----+----1----+----2----+----3----+----4----+----5----+----6----+----7-|--+----8
      1          PLI01A:PROC OPTIONS(MAIN);         /* PL/I DTCU TESTCASE           */
      2          /*******************************************************************/
      3          /* Licensed Materials - Property of IBM                            */
      4          /*                                                                 */
      5          /* 5655-P14: Debug Tool for z/OS                                   */
      6          /* 5655-P15: Debug Tool Utilities and Advanced Functions           */
      7          /* (C) Copyright IBM Corp. 1997, 2005 All Rights Reserved          */
      8          /*                                                                 */
      9          /* US Government Users Restricted Rights - Use, duplication or     */
     10          /* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.*/
     11          /*                                                                 */
     12          /*******************************************************************/
     13
     14          DCL EXPARM1 FIXED BIN(31) INIT(5);
     15          DCL EXPARM2 FIXED BIN(31) INIT(2);
     16          DCL PARM2   FIXED BIN(31) INIT(2);
     17          DCL PLI01B EXTERNAL ENTRY;         /*                              */
     18  >       DO WHILE (EXPARM1 > 0);            /*  THIS DO LOOP EXECUTED 5 TIMES*/
     19  >          EXPARM1 = EXPARM1 -1;           /*                              */
     20 B>          CALL  PLI01B(PARM2);            /* PLI01B CALLED 5 TIMES        */
     21  >       END;
     22  >       IF (EXPARM2 = 0) THEN              /* THIS BRANCH ALWAYS TAKEN     */
     23  <         CALL PROC2A(EXPARM2);            /* PROC2A NEVER  CALLED         */
     24  >       DO WHILE (EXPARM2 > 0);            /* DO LOOP EXECUTED TWICE       */
     25  >         EXPARM2 = EXPARM2 - 1;
     26  >       END;
     27  >       RETURN;
     28
     29  <       PROC2A: PROCEDURE(P1PARM1);         /* THIS PROCEDURE NEVER EXECUTED */
     30          DCL P1PARM1 FIXED BIN(31);
     31  <       P1PARM1 = 10;
     32  <       END PROC2A;
     33 I>       END PLI01A;

                          Start  End
  Marker type   Selection  Column Column String
  ------------ ---------  ------ ------ ------------------------------
  SINGLE        INCLUDE         2     80 PLI01
  SINGLE        EXCLUDE         2     80 PLI01B

            Statements  Executed  Percentage
            ----------  --------  ----------
  Total             14        11       78.57
```

```
  Included          1         1      100.00
  Excluded          0         0        0.00
  Incl/Excl         1         1      100.00
```

Below is a sample of a presentation format annotated listing report for a C program. The format is similar to the other supported languages.

```
1Rpt Date :  2013/10/31      Rpt Time:   08:07:42

 Run Date :  2013/10/16      Run Time:   13:33:07
 Group ID 1: COST            Group ID 2: BENEFIT         User ID:   USERIBM
 Load Name:  C01             Prog Name:  C01A
 Comp Date:  2013/05/07      Comp Time:  15:53:00   Debug override: N
 Tot Stmts:  12              Exec Stmts: 8               Percent:   66.66%

 Rollup History:

 Observation is not part of rollup

               *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     1           main()
     2          /*******************************************************************/
     3          /* Licensed Materials - Property of IBM                            */
     4          /*                                                                 */
     5          /* 5655-W70: Debug Tool for z/OS                                   */
     6          /* Copyright IBM Corp. 1997, 2012 All Rights Reserved              */
     7          /*                                                                 */
     8          /* US Government Users Restricted Rights - Use, duplication or     */
     9          /* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.*/
    10          /*                                                                 */
    11          /*******************************************************************/
    12
    13             {                             /* DEFECT456BEGIN                */
    14 I>            int EXPARM1 = 5;                                               PMR13579
    15 I>            int EXPARM2 = 2;                                               PMR13579
    16               extern void C01B(void);
    17 I<            void PROCA(int);           /* function not called           */
    18 B>            while (EXPARM1 > 0)        /* execute loop 5 times          */ PMR12345
    19               {
    20 I>               EXPARM1 = EXPARM1 -1;
    21 I>               C01B();                 /* call C01B  5 times            */
    22               }
    23 I>            if (EXPARM2 == 0)          /* branch taken  DEFECT456END    */
    24  <              PROCA(EXPARM2);          /* not executed                  */
    25 I>            while (EXPARM2 > 0)        /* loop execute 2 times          */ PMR1
    26  >              EXPARM2 = EXPARM2 - 1;   /* executed twice                */
    27             }
    28  <          void PROCA(int P1PARM1)      /* function not called           */
    29             {
    30  <            P1PARM1 = 10;              /* not executed                  */
    31             }

                           Start  End
  Marker type   Selection  Column Column String
  ------------- ---------  ------ ------ ------------------------------
  SINGLE        INCLUDE        73     76 PMR1
  SINGLE        EXCLUDE        73     80 PMR12345
  SECTIONBEGIN  INCLUDE         8     72 DEFECT456BEGIN
  SECTIONEND    INCLUDE         8     72 DEFECT456END

            Statements  Executed  Percentage
            ----------  --------  ----------
  Total             12         8       66.66
  Included           7         6       85.71
  Excluded           0         0        0.00
  Incl/Excl          1         1      100.00
```

The following table shows the column layout for the source lines:

Table 28. The column layout for the source lines

Columns	Contents
1	Blank.
2 through 7	6 digit listing line number, right justified, leading zeros suppressed.

Table 28. The column layout for the source lines (continued)	
Columns	**Contents**
9	If executable and not using E.1 (V)iew then: • 'I' included • 'E' excluded • 'B' both included and excluded • ' ' neither included nor excluded If not executable or using E.1 (V)iew then: • ' '
10 through 15	1 column per statement on this line. For example, col 10 represents the 1st statement, column 11 represents the 2nd statement. These columns indicate whether the statement was executed (>), unexecuted (<), unspecified ( ) or specified multiple times (M, which likely indicates an internal error). Column 15 may contain a plus sign (+) if an executed or unexecuted tag value indicates a statement number that exceeds 6 for this line.
17 through 96	Source columns 1 - 80

Batch facilities

Extraction function

This function selects, from an input file, code coverage observations that are based on the selection criteria and writes to an output file in XML. The input of the calling interface is as follows:

You can run the code coverage Extraction Utility in batch by running the EQAXCCX2 REXX exec. You must specify the following DDNAMES:

EQACSINP
Location of Observation file.

EQACSSEL
Location of Selection file.

EQACSOUT
Location of output code coverage extracted observations file.

An example of using EQAXCCX2 in batch can be found in *hlq*.SEQASAMP(EQACCEXT).

All three files are allocated either as a sequential file or PDSE. The file format should be VB and LRECL=255. If it is a PDSE file, the data set name must include a member name.

Report functions

XML Report

This function composes a full XML file for reporting purpose. The file contains the data for a report writer to write a readable report or an HTML file for the browser. A full report XML file contains the following two sections:

- The observation section contains the selected observations. Each observation includes statements which might be marked as included or excluded and code coverage extracted observations XML tags that are generated from the report generator.
- The selection criteria section contains the selection criteria and source markers.

The code coverage Report Utility can be started in batch by starting the EQAXCCR2 REXX exec with the XML parameter. You must specify the following DDNAMES:

EQACRINP
Code coverage extracted observations that are based on selection criteria.

EQACRSEL
Code coverage Selection file.

EQACROUT
XML report output.

An example of using EQAXCCR2 in batch to generate a XML report can be found in *hlq*.SEQASAMP(EQACCXRP).

All three files are allocated either as a sequential file or PDSE. The file format is VB for the XML file, VBA for the Presentation file, and LRECL=255. If it is a PDSE file, the data set name must include a member name.

Presentation report

A Presentation report can also be generated. It contains the same data as the XML report, except it is presented in a viewer friendly format.

To generate a Presentation report, specify PFMT as the parameter to EQAXCCR2. An example of using EQAXCCR2 in batch to generate a Presentation report can be found in *hlq*.SEQASAMP(EQACCPRP).

Batch examples

You can find JCL samples in *hlq*.SEQASAMP for batch jobs. The JCL samples contain the steps to build a test case, and then to specify, gather, process, and document code coverage for the test case.

The following members contain JCL samples of gathering code coverage in batch jobs:

Table 29.

Member name	Compiler
EQACC1VZ	Enterprise COBOL for z/OS and OS/390 V3 Enterprise COBOL for z/OS V3 and V4
EQACC2VZ	Enterprise PL/I for z/OS V4.2 through V4.5 and V5
EQACC3VZ	z/OS XL C
EQACC4VZ	Enterprise COBOL for z/OS V5 and V6

The JCL samples consist of the following steps:

1. Compile procedure
2. Creating data sets
3. Compiling the source
4. Binding the output of the compiler to create a load module
5. Clearing out the CCOUTPUT file
6. Loading the CCPRGSEL Options file
7. Loading the CCOBSSEL Selection file
8. Running the load module, gathering code coverage data and writing out the CCOUTPUT file

9. Running DTU E.4 - Observation extraction
10. Running DTU E.5.1 - XML Report generation
11. Running DTU E.5.2 - Presentation Report generation

Generating code coverage for CICS transactions

This section shows a technique that you can use to generate a code coverage Observation file for a CICS transaction.

Prepare the following files outside of CICS:

- An Options file, as previously discussed.
- A sequential EQAOPTS file with the code coverage EQAOPTS commands as previously discussed.
- A z/OS Debugger commands file with a single GO command (containing the string GO; starting at column 8)

In CICS, run the DTCN transaction and press PF9 (OPTions) and fill it in as follows:

```
DTCN               z/OS Debugger CICS Control - Menu 2                  S07CICPB

 Select z/OS Debugger options

 Test Option     ==> TEST                Test/Notest
 Test Level      ==> ERROR               All/Error/None
 Commands File   ==> USERIBM.CC.CICS.GOCMD
 Prompt Level    ==> PROMPT
 Preference File ==> *

 EQAOPTS File    ==> USERIBM.CC.EQAOPTS

 Any other valid Language Environment options
 ==> ENVAR("EQA_STARTUP_KEY=CC")

 PF1=HELP 2=GHELP 3=RETURN
```

Then press PF3 (RETURN), on this screen, PF4 (SAVE) on the main DTCN screen, and PF3 (EXIT) to exit DTCN. Then run your transaction. Each transaction you run will append a new set of observations to the Observation file.

This will run the transaction in unattended mode. If you want to interact with z/OS Debugger while collecting observations, remove the Commands file from the Options panel shown above, and change the value of EQA_STARTUP_KEY to DCC.

Generating code coverage in IMS Transaction Isolation

To generate code coverage in IMS Transaction Isolation, you need to define an EQA_STARTUP_KEY environment variable and an EQAOPTS commands data set in the Manage Additional Libraries and Delay Debug panel (EQAPMPRG).

For more information about panel EQAPMPRG, see “Using IMS Transaction Isolation to create a private message-processing region and select transactions to debug” on page 337.

On panel EQAPMPRG, define the following settings for IMS Transaction Isolation:

- Add the EQA_STARTUP_KEY environment variable, for example, `ENVAR("EQA_STARTUP_KEY=CC")`, in the **Other run-time options** field.
- Add a data set that contains EQAOPTS commands configured for capturing z/OS Debugger code coverage, for example, `'USER.EQAOPTS.LOAD' SHR`, in the data set table at the bottom of the panel.

```
Manage additional libraries and delay debug options               Row 1 from 2
Command ===>                                    Scroll ===> PAGE

Your private message region will be set up to use delay debug
mode for processing debugging preferences.  Type / below to edit
your delay debug profile data set.

   Edit delay debug profile data set

Other run-time options: ENVAR("EQA_STARTUP_KEY=CC")


The following DD cards will be added to the top of the STEPLIB
concatenation for the private message region that z/OS Debugger
will launch.  You may add, edit or delete data sets from this
list before launching the message region for testing.

Cmd Seq C DD Information (DSN/Sysin/Sysout/Dummy)            DISP
          ****************** Top of Data ******************
     1    'USER.EQAOPTS.LOAD'                                 SHR
```

XML tags for code coverage

This section contains a set of XML tags for code coverage.

XML tags definition for the Observation file

The XML file contains the following XML tags and content:

- All tags have a corresponding end tag </XXXXX>. The table shows the end tag when it needs to be on the same line as the start tag.
- The tag name is upper case.
- The occurrence column shows the number of tags that are allowed in context.

Table 30. XML tags and the contents

XML tag	Description	Occurrence
<COMPILATIONUNIT>	Compilation unit container	>=1, per <LOADMODULE>
<COMPILEDATE>	Compile date container	1, per <COMPILATIONUNIT>
<COMPILETIME>	Compile time container	1, per <COMPILATIONUNIT>
<CSECT>	CSECT or program container	>=1, per <COMPILATIONUNIT>
<DAY>xxx</DAY>	Day	1, per <COMPILEDATE> or <RUNDATE>
<DBGOV>x</DBGOV>	Debug override (Y or N)	1, per <CSECT>
<DTCODECOVERAGEFILE>	z/OS Debugger code coverage data	>=1, per file
<DTCODECOVERAGEREPORT>	Code coverage report data	1 or 0, per file
<EXCEXECD>xxx</EXCEXECD>	Total number of excluded source statements executed	1, per <STATISTICS>
<EXCPRCNT>xxx</EXCPRCNT>	Percentage of excluded source statements executed	1, per <STATISTICS>
<EXCSTMTS>xxx</EXCSTMTS>	Total number of excluded source statements	1, per <STATISTICS>

Table 30. XML tags and the contents (continued)

XML tag	Description	Occurrence
<EXECUTED>x x</EXECUTED>	List of the statement or line numbers that were executed. Each number separated by a blank.	>=0, per <CSECT>
<EXTNAME>xxx</EXTNAME>	Name of CSECT or program	1, per <CSECT>
<GROUPID1>xxx</GROUPID1>	User provided group ID. Default is *.	1 or 0, per file
<GROUPID2>xxx</GROUPID2>	User provided group ID. Default is *.	1 0r 0, per file
<HOURS>xxx</HOURS>	Hours	1, per <COMPILETIME> or <RUNTIME>
<IECEXECD>xxx</IECEXECD>	Total number of included and excluded source statements executed	1, per <STATISTICS>
<IECPRCNT>xxx</IECPRCNT>	Percentage of included and excluded source statements executed	1, per <STATISTICS>
<IECSTMTS>xxx</IECSTMTS>	Total number of included and excluded source statements	1, per <STATISTICS>
<INCEXECD>xxx</INCEXECD>	Total number of included source statements executed	1, per <STATISTICS>
<INCPRCNT>xxx</INCPRCNT>	Percentage of included source statements executed	1, per <STATISTICS>
<INCSTMTS>xxx</INCSTMTS>	Total number of included source statements	1, per <STATISTICS>
<LOADMODULE>	Load module container	>=1, per file
<MARKEDSTMTS>	Container for marked statements	1, per <CSECT>
<MEMBERNAME>xxx</MEMBERNAME>	Name of the load module	1, per <LOADMODULE>
<MINUTES>xxx</MINUTES>	Minutes	1, per <COMPILETIME> or <RUNTIME>
<MONTH>xxx</MONTH>	Month	1, per <COMPILEDATE> or <RUNDATE>
<ORIGINALCOLLECTION>	Container for original observations that are rolled up	1, per <COMPILATIONUNIT>
<ORIGINALOBSERVATION>	Container for original observation that is merged	>=1, per <ORIGINALCOLLECTION>

Table 30. XML tags and the contents (continued)

XML tag	Description	Occurrence
<PROGRAMDSCOMPILEDATE> xxx </PROGRAMDSCOMPILEDATE>	The compile date container of data set that contains program source	1, per <COMPILATIONUNIT>
<PROGRAMDSCOMPILETIME> xxx </PROGRAMDSCOMPILETIME>	The compile time container of data set that contains program source	1, per <COMPILATIONUNIT>
<PROGRAMDSNAME>xxx</ PROGRAMDSNAME>	The name of data set that contains program source	1, per <COMPILATIONUNIT>
<PROGRAMDSTYPE>xxx</ PROGRAMDSTYPE>	The type of data set that contains program source. Valid types are: • 1 - COBOLSYSDEBUG (Enterprise COBOL for z/OS V3 and V4) • 2 - PLISYSDEBUG (Enterprise PL/I for z/OS V4.2 and above) • 4 - Program Object (Enterprise COBOL for z/OS V5 and above) • 5 - Source (z/OS XL C)	1, per <COMPILATIONUNIT>
<RUNDATE>	Date that the code coverage data was saved	1, per <DTCODECOVERAGEFILE> or <COVERAGEFILE >
<RUNTIME>	Time that the code coverage data was saved	1, per <DTCODECOVERAGEFILE> or <COVERAGEFILE>
<SECONDS>xxx</SECONDS>	Seconds	1, per <COMPILETIME> or <RUNTIME>
<STATISTICS>	Container for code coverage statistics	1, per <CSECT>
<STMT>xxx</STMT>	Marked source statement	>=1, per <MARKEDSTMTS>
<TOTEXECD>xxx</TOTEXECD>	Total number of source statements executed	1, per <STATISTICS>
<TOTPRCNT>xxx</TOTPRCNT>	Percentage of source statements executed	1, per <STATISTICS>
<TOTSTMTS>xxx</TOTSTMTS>	Total number of source statements	1, per <STATISTICS>
<UNEXECUTED>x x</UNEXECUTED>	List of the statement or line numbers that were not executed. Each number separated by a blank.	>=0, per <CSECT>

Table 30. XML tags and the contents (continued)		
XML tag	**Description**	**Occurrence**
<USERID>xxx</USERID>	User ID that generates the file. Default is *.	1 or 0, per file
<YEAR>xxx</YEAR>	Year	1, per <COMPILEDATE> or <RUNDATE>

XML tag hierarchy for the Observation file

The following sample XML output shows the hierarchical structure of the tags, the containers, and the tags within a container.

```
<DTCODECOVERAGEFILE>
<RUNDATE>
<YEAR>....</YEAR>
<MONTH>....</MONTH>
<DAY>....</DAY>
</RUNDATE>
<RUNTIME>
<HOURS>....</HOURS>
<MINUTES>....</MINUTES>
<SECONDS>....</SECONDS>
</RUNTIME>
<GROUPID1>....</GROUPID1>
<GROUPID2>....</GROUPID2>
<USERID>....</USERID>
<LOADMODULE>
<MEMBERNAME>....</MEMBERNAME>
<COMPILATIONUNIT>
<PROGRAMDSNAME>....</PROGRAMDSNAME>
<PROGRAMDSTYPE>....</PROGRAMDSTYPE>
<COMPILEDATE>
<YEAR>....</YEAR>
<MONTH>....</MONTH>
<DAY>....</DAY>
</COMPILEDATE>
<COMPILETIME>
<HOURS>....</HOURS>
<MINUTES>....</MINUTES>
<SECONDS>....</SECONDS>
</COMPILETIME>
<CSECT>
<EXTNAME>....</EXTNAME>
<DBGOV>....</DBGOV>
<EXECUTED>....</EXECUTED>
<UNEXECUTED>....</UNEXECUTED>
</CSECT>
</COMPILATIONUNIT>
</LOADMODULE>
</DTCODECOVERAGEFILE>
```

XML Tags used in the Options file

The following example shows the XML tags used in the Options file:

```
<GROUPID1></GROUPID1>
<GROUPID2></GROUPID2>
<EXTNAME></EXTNAME>
<EXTNAME></EXTNAME>
<EXTNAME></EXTNAME>
<EXTNAME></EXTNAME>
```

The three tags are defined in the table of *common tags* and *XML tags and the contents*.

XML tags used in the Selection file

The following table shows the description of XML tags used in the Selection file:

Table 31. Description of XML tags used for selection criteria

Tag	Description	Occurrence
<ATTRIBUTE>	A attribute criterion container	>=1, per selection criteria file
<NAME>xxx</NAME>	Name of selected attribute	1, per attribute criterion
<OPERATOR>xxx</OPERATOR>	Comparison operator used to see if the attribute of an observation compares successfully	1, per attribute criterion
<ROLLUP>xxx</ROLLUP>	Roll up characteristics of the attribute criterion. Valid values are as follows: • Y - Yes. Observations with different values of the attributes can be merged (rolled up). • N - No. Observations with different values of the attributes cannot be merged (rolled up).	1, per attribute with the following names: • GROUPID1 • GROUPID2 • USERID • DBGOV

Table 32. Description of XML tags used for source maker

Tag	Description	Occurrence
<ENDCOLUMN>xxx</ ENDCOLUMN>	The end column of a source statement when searching for source marker value.	1, per source marker
<MARKERTYPE>xxx</ MARKERTYPE>	Marker type. Valid types are as follows: • SECTIONBEGIN • SECTIONEND • SINGLE	1, per source marker
<MARKERVALUE>xxx</ MARKERVALUE>	A character string or hex value used to check if a source statement contains such string or hex value. Attribute criterion. Valid values are as follow: • Y - Yes. Observations with different values of the attributes can be merged (rolled up). • N - No. Observations with different values of the attributes cannot be merged (rolled up).	1, per attribute with the following names: • GROUPID1 • GROUPID2 • USERID • DBGOV

Table 32. Description of XML tags used for source maker (continued)

Tag	Description	Occurrence
<SECTION>xxx</SECTION>	Include or exclude the source statement that contains the source maker value when calculating the code coverage statistics. Valid values are as follows: • INCLUDE • EXCLUDE	1, per source marker
<SOURCEMARKER>	A source maker container. Selected attribute container.	>=1, per source marker file
<STARTCOLUMN>xxx</ STARTCOLUMN>	The start column of a source statement when searching for source marker value.	1, per source marker

Appendix F. Notes on debugging in batch mode

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

z/OS Debugger can run in batch mode, creating a noninteractive session.

In batch mode, z/OS Debugger receives its input from the primary commands file, the USE file, or the command string specified in the TEST run-time option, and writes its normal output to a log file.

Note: You must ensure that you specify a log data set.

Commands that require user interaction, such as PANEL, are invalid in batch mode.

You might want to run a z/OS Debugger session in batch mode if:

- You want to restrict the processor resources used. Batch mode generally uses fewer processor resources than interactive mode.
- You have a program that might tie up your terminal for long periods of time. With batch mode, you can use your terminal for other work while the batch job is running.
- You are debugging an application in its native batch environment, such as MVS/JES or CICS batch.

When z/OS Debugger is reading commands from a specified data set or file and no more commands are available in that data set or file, it forces a GO command until the end of the program is reached.

When debugging in batch mode, use QUIT to end your session.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Chapter 16, “Starting z/OS Debugger in batch mode,” on page 129

© Copyright IBM Corp. 1992, 2023

Appendix G. Using IMS message region templates to dynamically swap transaction class and debug in a private message region

Note: This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

You can use predefined IMS message region templates to debug a specific transaction in a private message region by using IBM z/OS Debugger Utilities option 4.3 Swap IMS Transaction Class and Run Transaction (panel EQAPMPRS). This panel and its sub-panels allow you to take the following actions:

1. Start a private message region from a predefined message region template. This template specifies a message class that is reserved for debug purposes.
2. Assign a transaction that you want to debug to the class for the private message region.
3. Schedule a message for the transaction.
4. After you have finished debugging the transaction and it completes, the transaction is assigned to its original class and the private message region is stopped.

To dynamically launch a private message region and run a specific transaction in that region, complete the following steps:

1. Start IBM z/OS Debugger Utilities. For detailed information, see "Starting IBM z/OS Debugger Utilities" on page 9.
2. In the IBM z/OS Debugger Utilities panel (EQA@PRIM), type 4 in the Option line and press Enter.
3. In the Manage IMS Programs panel (EQAPRIS), type 3 in the Option line and press Enter.
4. In the Debug IMS Transaction - Select Private Message Region panel (EQAPMPRS), type a forward slash (/) beside the template you want to use, and press Enter. You can choose from the following types of templates:
   - Predefined templates from a common z/OS Debugger Setup Utility data set
   - Templates previously customized and stored in a private z/OS Debugger Setup Utility data set

   If you use a member from a private z/OS Debugger Setup Utility data set, you can see the Create Private Message Regions - Edit Setup File panel (EQAPFORA). Enter the information to edit an existing setup file.
5. In the Specify Transaction and Additional Test Libraries panel (EQAPMPRT), type the transaction name that you want to launch in your private message region. You also need to enter any additional information to send when the message is scheduled.

   You might want to add data sets to the message region STEPLIB concatenation. To add a data set, type an I in the Cmd column of the data set table at the bottom of the panel. This adds an empty line to the table that you can fill in with a data set name and a disposition.

   Each data set in the table is added to the beginning of the STEPLIB concatenation for the message region, in the order specified in the table. You might change the relative position of the data sets in the table by modifying the values in the Seq column.

   For more advanced manipulation of the DD card, you can type a forward slash (/) in the Cmd column for a DD card and press Enter. A menu is displayed where you can change the allocation parameters, the DCB parameters, and other characteristics that are specified on the DD card for a data set.
6. To start the private message region and schedule the transaction, run the z/OS Debugger IMS Transaction Swap Utility (the EQANBSWT Batch Message Program, hereafter referred to as EQANBSWT). This can be done in one of the two following ways:
   - Press PF4 to run the transaction. This starts EQANBSWT in the foreground of your TSO session.

© Copyright IBM Corp. 1992, 2023

- Press PF10 to submit. This displays a JCL deck that runs the EQANBSWT program that you can submit to the Job Entry System by using the ISPF SUBMIT command.

EQANBSWT will start the private message region. By default, the TEST parameter will be the following:

```
TEST(ALL,*,PROMPT,VTAM%userid:*)
```

The *userid* is your TSO user ID.

If you want to use a different TEST parameter, type a forward slash (/) beside the **Enter / to modify parameters** field, and press Enter. The **EQAPFMTP** panel is displayed. Specify the TEST parameter sub-options and session type, and press PF3 to save.

EQANBSWT will also start a second private message region, by using the NOTEST parameter, and serving the same class. This region allows additional messages scheduled for the transaction to be processed when the transaction is being debugged in the TEST region at the same time.

EQANBSWT will then assign the transaction to the class served by the private message region and schedule the transaction.

When the transaction completes, EQANBSWT stops the private message regions and assigns the transaction to the class to which it was initially assigned.

The jobs that are started to run EQANBSWT and the two private message regions use the job card you specified in IBM z/OS Debugger Utilities option 0, Job Card. Each job name is replaced by values that you entered in Debug Utilities option 4.0, Set IMS Program Options. If you do not set personal defaults in option 4.0, system defaults are used.

In certain circumstances, EQANBSWT does not complete normally. To interrupt EQANBSWT, take one of the following steps:

- If you ran EQANBSWT in the foreground by using the Run command, press the ATTN or PA1 key and follow the prompts to stop the process.
- If you ran EQANBSWT as a batch job by using the Submit command, issue the STOP *jobname* MVS command, for example, by typing /P *jobname* in the Spool Display and Search Facility (SDSF).

7. When you want to leave the Specify Transaction and Additional Test Libraries panel (EQAPMPRT), you can save any changes you have made into a private message region template.
   - If you selected a predefined message template in step 4, type SAVE AS and press Enter. This displays the z/OS Debugger Foreground – Edit Setup File panel (EQAPFOR), where you can enter a data set name for your private copy of the template.
   - Otherwise, press PF3 to Exit. Your changes are saved to the private template you opened in step 4.

Appendix H. Displaying and modifying CICS storage with DTST

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The DTST transaction enables you to display, scan, and modify CICS storage. It is a BMS transaction and runs on a 3270 terminal.

Starting DTST

This topic describes the methods of starting DTST and gives examples.

Before you begin, if you need to modify storage, verify with your system programmer that you have the authority to modify CICS key storage, USER key storage, or both. "Authorizing DTST transaction to modify storage" in *IBM z/OS Debugger Customization Guide* describes the steps the system programmer must do to authorize you to modify CICS key storage, USER key storage, or both.

You can start the DTST transaction with or without specifying a base address. A base address can be any of the following items:

- A literal hexadecimal number (for example, 45CB00)
- A 64 bit address (for example, 48_40B00000)
- The name of a program (for example, MYPGM)
- An offset calculation or indirection (for example, 45CB00+40)

You can also specify that DTST take a specific action when it starts. You specify an action with one of the following characters:

- P, which means to page forward or backward.
- S, which means to search through storage until a specific target is found.

"Syntax of the DTST transaction" on page 501 describes all the parameters.

Examples of starting DTST

The following examples illustrate how to enter the DTST command with parameters.

Example: Starting DTST and specifying a literal hexadecimal number

To display storage at address 45CB00, enter the command `DTST 45CB00`.

The base address is 45CB00.

Example: Starting DTST and specifying a 64 bit address

To display storage at address 48_40B00000, enter the command `DTST 48_40B00000`.

The base address is 48_40B00000.

Example: Starting DTST and specifying a program name

To display program storage for program MYPROG, enter the command `DTST P=MYPROG`.

The base address is the address of the program in storage.

Example: Starting DTST and specifying an offset

To display storage at an negative offset of D0 bytes from address 45CB00, enter the command `DTST 45CB00 - D0`.

© Copyright IBM Corp. 1992, 2023

The result of the calculation (45CB00-D0) is the base address. In this example, the base address is 45CA30.

To display program storage at an positive offset of 28 bytes from the starting address of program MYPROG, enter the command `DTST P=MYPROG+28`.

If the starting address of program MYPROG is 8492A000, then the result of the calculation (8492A000+28) is the base address (8492A028).

If fullwords generate protection exceptions (for example, in fetch-protected storage), DTST displays question marks in the Storage Key field.

Example: Starting DTST with indirect addressing

To display storage by indirection, use an asterisk (*) to indicate 31-bit addressing or an at sign (@) to indicate 24-bit addressing. DTST uses the fullword at that address as the base address.

If you want to use the fullword at address 45CB00 as the base address, enter the command `DTST 45CB00*`.

You can combine multiple offset or levels of indirection. For example, if you enter the command `DTST 45CB00 + b* + 14** + 14*`, DTST calculates the base address in the following order:

1. Beginning with 45CB00, add B0. The result is 45CBB0.
2. Go to location 45CBB0 to obtain the address at that location. For this example, assume that the address is 29AD00.
3. Add 14 to 29AD00. The result is 29AD14.
4. Go to location 29AD14 to obtain the address at that location. For this example, assume that the address is 1838AD.
5. Go to location 1838AD to obtain the address at that location. For this example, assume that the address is 251936.
6. Add 14 to 251936 to get the result 25194A.
7. Go to location 25194A to obtain the address at that location. For this example, assume that the address is 3920AD. DTST opens the memory window and display the contents of storage beginning at 3920AD.

Example: Starting DTST with the BASE keyword

The BASE keyword can make it easier to write long command lines. The BASE keyword is assigned the value of the base address of the previous DTST command. For example, if you enter the command `DTST 45CB00+10*`, BASE is assigned the value of the result of 45CB00+10*. If you want to use the value of 45CB00+10* in a subsequent command, use the BASE keyword. For example, `DTST BASE+20*`.

Example: Starting DTST with a scan request

You can specify data that you are looking for by adding a scan request to the DTST command. For example, to find the data 'WORKAREA' starting at base address 45CB00, enter the command `DTST 45CB00,S='WORKAREA'`. The scan starts at the base address and continues for 4K bytes. To find the data 'WORKAREA' starting at base address 45CB00 at the beginning of every double word, enter the command `DTST 45CB00,S8='WORKAREA'`. You can specify that the scan be done in a negative direction, which means that addresses are decreasing in value.

Example: Starting DTST with a page number request

You can specify a page you want displayed by adding a page request to the DTST command. For example, to display storage that is 5 pages from the base address 45CB00, enter the command `DTST 45CB00,P=5`. This is equivalent to entering the command `DTST 45CB00`, then pressing the page down keys five times. If you enter the command `DTST 45CB00,P=-5`, it is equivalent to entering the command `DTST 45CB00`, then pressing the page up keys five times.

Modifying storage through the DTST storage window

After you start the DTST transaction, the storage window is displayed. You can modify the contents of storage being displayed in the storage window.

Before you begin, verify with your system programmer that you have the authority to modify CICS key storage, USER key storage, or both. "Authorizing DTST transaction to modify storage" in *IBM z/OS Debugger Customization Guide* describes the steps the system programmer must do to authorize you to modify CICS key storage, USER key storage, or both.

After you verify that the previous DTST command ran successfully, you can do the following steps to modify storage.

1. Press PF9 to enter modify mode. The command line becomes protected, and columns four through seven become unprotected.
2. Move your cursor to data you want to modify and type in the new data. You can modify several different locations at the same time.
3. Press Enter. DTST verifies that the data you entered is valid. DTST makes all modifications that contain valid data. If any word contains invalid data, the line contains that word is highlighted. You can correct the invalid data, then press Enter to verify the change.
4. Press any function key to end modify mode. However, you can not press any of the following keys:
   - PF10
   - PF11
   - the CLEAR key
   - the Enter key when you have typed in any modifications

Navigating through the DTST storage window

There are several ways to navigate through the DTST storage window.

After you enter the DTST command, do the following steps:

1. Choose one of the following methods to navigate through the window:
   - Use the PF7 or PF8 keys to move up or down a page, respectively.
   - Move your cursor to the command line and enter a new address. All spaces are ignored, except the one after the transaction name (DTST) and any within apostrophes (').
   - Move your cursor over any fullword displayed in column 4 or 6, then press Enter.
2. To close the DTST storage window, press the PF3 key.

DTST storage window

The DTST storage window is the interface you use to display and modify storage.

```
+-------------------------------------------------------------------------+
| Command  : DTST 00100000                                                |
| Response : Normal                                                       |
| Page     : HOME     Storage Key : USER                                  |
+-------------------------------------------------------------------------+
| 00100000  0000 00 | C4A3D983 826E6E6E A7E10888 A0050004 | DtRcb>>>x..h.... |
| 001 1 10  0 2  3 | 001 4 12 000 5 00 000 6 00 000 7 00 | ..... 8 ..... |
| 00100020  0020 02 | A7E09170 8009D150 A7E152D8 00000000 | x.j...J.x..Q.... |
| 00100030  0030 03 | 00000001 000C5258 00000000 00000000 | ................ |
| 00100040  0040 04 | A6BF6098 800A4968 800B01DB 00000000 | w.-q.......Q.... |
| 00100050  0050 05 | 00000000 00000000 800B30CB 80140C10 | ..........H.... |
| 00100060  0060 06 | 8074B6A0 80155CA8 80160818 801683C0 | ......*y......c{ |
| 00100070  0070 07 | A6BFD338 00000000 A6BFD190 00000000 | w.L.....w.J..... |
| 00100080  0080 08 | 00000000 00000000 00000000 00000000 | ................ |
| 00100090  0090 09 | 00000000 00000000 00000000 00000000 | ................ |
| 001000A0  00A0 10 | 00000000 00000000 00000000 00000000 | ................ |
| 001000B0  00B0 11 | 00000000 00000000 00000000 00000000 | ................ |
| 001000C0  00C0 12 | 00000000 00000000 00000000 00000000 | ................ |
| 001000D0  00D0 13 | 00000000 00000000 00000000 00000000 | ................ |
| 001000E0  00E0 14 | 00000000 00000000 00000000 00000000 | ................ |
| 001000F0  00F0 15 | 00000000 00000000 00000000 00000000 | ................ |
+-------------------------------------------------------------------------+
|  1=Hlp  2=Retrv  3=End  5=RepeatScan  7=Up  8=Down  9=Modfy  ENTER=ReCalc  |
+-------------------------------------------------------------------------+
```

The following list describes all the parts of the interface.

Command
The most recent command you entered.

Response
The result of the most recent command you entered. If the command was successful, the word `Normal` is displayed in this field. If the command was unsuccessful, a message indicating the type of error that occurred in the previous command is displayed.

Storage Key
Displays one of the following values:

`CICS`
Indicates that the CICS[hyphen]key storage is displayed.

`USER`
Indicates that the USER[hyphen]key storage is displayed.

`KEY`*n*
Indicates that Key *n* storage is displayed.

`????`
Indicates that the key is not recognized.

`!!!!`
Indicates that the key was not obtained.

Column 1
Displays the address of storage. The addresses are organized on a word boundary. If you enter an address that is not on a word boundary, the bytes preceding the address, up to the beginning of the word, are padded with blanks.

Column 2
Displays the offset of the address in column 1 from the base address. The offset is displayed in hexadecimal.

Column 3
Displays the line number (0 to 15) in the window. The line number is displayed in decimal.

Columns 4 through 7
Displays the contents of storage in hexadecimal. Each column represents four bytes.

Column 8
Displays the contents of storage contents in EBCDIC.

Some of the following PF keys work only if the previous operation was successful. If the previous operation was successful, the word `Normal` is displayed in the **Response** field.

PF1 (Help)
Displays the help screen. The help screens display command syntax with examples and lists all keywords.

PF2 (Retrieve)
Retrieves the previous command from the command history. DTST stores up to 10 commands in the command history, discarding the older commands to save newer commands.

PF3 (Exit)
Clears the screen and ends the transaction.

PF5 (RepeatScan)
Repeats the scan operation.

PF7 (Up)
Moves one page (256 bytes) back in storage. The base address is not recalculated.

PF8 (Down)
Moves one page (256 bytes) forward in storage. The base address is not recalculated.

PF9 (Modify)
Starts modify mode.

Enter
DTST does one of the following tasks:

- When the cursor is on a fullword, DTST uses that fullword as the base address for the next command.
- Recalculates the base address from the input string, even if it has not changed, then changes the memory window so that the new base address is shown at the top of the screen.

Navigation keys for help screens

DTST provides a number of online help screens. You can access these screens by pressing PF1 on the main screen (when you are not in modify mode), which displays the main help index. You can navigate through the help screens by using the PF keys described in this topic.

PF3
Close the help screen and return to the DTST storage window.

PF7
Display the previous screen.

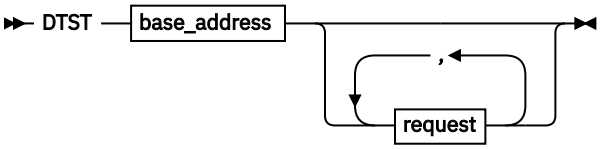
PF8
Display the next screen.

PF10
Display the main help index.

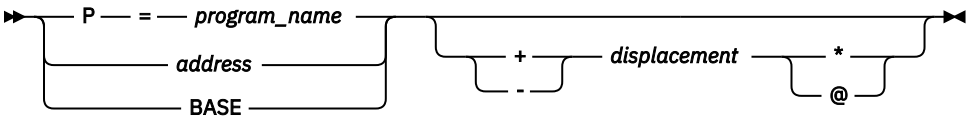
PF11
Display the last help screen.

Syntax of the DTST transaction

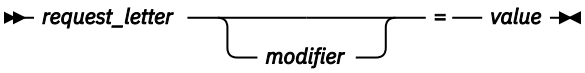
The DTST transaction displays storage in a memory window. You can navigate through the storage area and modify storage.



base_address



request



The following list describes the parameters:

address
A hexadecimal value for a 31-bit address (for example, 45CB0) or for a 64-bit address using underscore notation (for example 48_40B00000).

BASE
The value of the base address of the previously entered DTST command, which ran successfully.

displacement
A one to eight character hexadecimal value.

modifier
Indicates the direction in which to conduct the action. The default is forward, which means an increasing value. For the backward direction, use the negative sign (-).

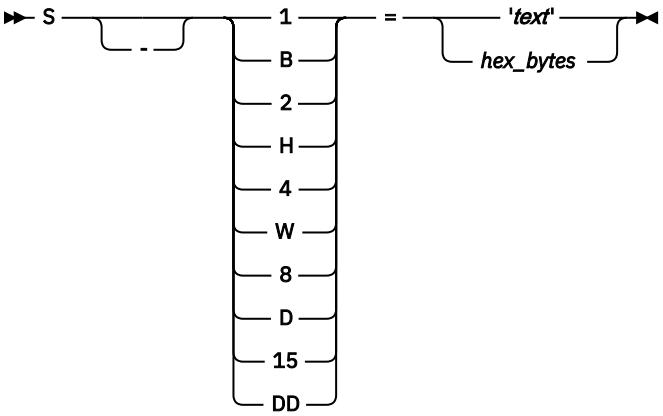
P
Indicates that you are specifying the name of a program and you want the starting address of that program to be used as the base address.

program_name
Name of a program.

request_letter
Indicates the action you want DTST to take. The *request_letter* can be one of the following characters:

P
Indicates that you want DTST to page up or down.

S
Indicates that you want DTST to search through storage and stop when it finds the target. The S request has the following syntax:



value
Hexadecimal or decimal value or a string enclosed in quotation marks (") or apostrophes ('). It is used to indicate the number of pages you want DTST to scroll or the target of a search.

Examples

To indicate that you want to display the fifth page (or screen) of memory after the address x'01000000', enter the command `DTST 01000000,P=5`. This is equivalent to entering `DTST 01000000`, then pressing PF8 five times.

To indicate that you want to find x'00404040' starting at address x'01000000', enter the command `DTST 01000000,S=00404040`.

Appendix I. z/OS Debugger Load Module Analyzer

Notes:

- Load Module Analyzer is deprecated and will be removed in a future version.
- This section is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

The z/OS Debugger Load Module Analyzer analyzes MVS load modules or program objects to determine the language translator (compiler or assembler) used to generate the object for each CSECT. This program can process all or selected load modules or program objects in a concatenation of PDS or PDSE data sets.

Choosing a method to start Load Module Analyzer

You can start the Load Module Analyzer in one of the following ways:

- Editing sample JCL provided in member EQAWLMA of data set *hlq*.SEQASAMP, and then submitting the JCL to run as a batch job.
- Selecting option 5 on the z/OS Debugger Utility ISPF panel.

Starting the Load Module Analyzer by using JCL

To start the Load Module Analyzer by using sample JCL, do the following steps:

1. Make a copy of member EQAWLMA in data set *hlq*.SEQASAMP.
2. Edit that copy, as instructed in the member.
3. Submit the JCL.
4. Review the results.

Starting the Load Module Analyzer by using IBM z/OS Debugger Utilities

To start the Load Module Analyzer by using IBM z/OS Debugger Utilities, do the following steps:

1. Start IBM z/OS Debugger Utilities.
2. Select option 5.
3. Enter the appropriate information into each field on the panel, keeping in mind the following behavior:
   - If you specify that you want a single load module or program object analyzed, Load Module Analyzer is run in the TSO foreground.
   - If you specify that you want an entire PDS or PDSE analyzed, JCL is generated to start Load Module Analyzer in MVS batch. Then, you must submit or save the generated JCL.

Description of the JCL statements to use with Load Module Analyzer

By default, the Load Module Analyzer program processes all members in the PDS or PDSE specified in the EQALIB DD statement. You can use control statements to instruct Load Module Analyzer to process only specific members of the data set concatenation.

The following information is included in the output for each CSECT:

- CSECT name
- Segment number (present only for a multi-segment module)

© Copyright IBM Corp. 1992, 2023

- CSECT offset in load module or segment
- CSECT length in hexadecimal
- Program-ID as contained in the binder IDR data
- Translator (compile or assembly) date
- Program description as supplied for the specified program ID.
- For OS/VS COBOL, PARM=RES or PARM=NORES.
  - PARM=RES indicates that one or more OS/VS COBOL CSECTs in the load module or program object were compiled with the RES compiler option.
  - PARM=NORES indicates that all OS/VS COBOL CSECTs in the load module or program object were compiled with the NORES compiler option.
- If you specify LEINFO, LESCAN, or CKVOLFPRS:
  - If a Language Environment prologue was detected, information is included in a string identified by `LEINFO=(….` This string contains the Language Environment entry name or an asterisk to indicate that the name is the same as the external symbol, Language Environment linkage type, source language, and translation date, time, and translator version.
  - If no Language Environment prologue was detected, but the prologue appears to be that of a known, non-Language Environment compiler, one of the following is included: C/C++, COBOL, or PL/I.
  - Otherwise, ASSEMBLER is included to indicate that the program is likely to be an assembler program.

Description of DD names used by Load Module Analyzer

Load Module Analyzer uses the following DD names:

EQALIB
Specifies a concatenation of PDS or PDSE data sets containing the load modules or program objects to be analyzed. If the same member is present in more than one of the concatenated data sets, only the first member is processed.

EQAPRINT
Specifies the output report. It can be in fixed block record format (RECFM=FBA) with a logical record length of 133 or more (LRECL >=133) or in variable block record format (RECFM=VBA) with a logical record length of 137 or more (LRECL >= 137).

EQAIN
Specifies the control statements. If you want only specific load modules or program objects to be processed, use the following syntax:

```
SELECT MEMBER=load_module_name
```

If you want all load modules to be processed, you can omit this DD statement, direct it to DUMMY, or direct it to empty data set. This file must be in fixed block record format (RECFM=FB) with a logical record length of 80 (LRECL=80). Each control statement must be on a separate line. The entries are free-form and you can use blanks before or after each keyword and operator. You can include comments by placing an asterisk in column 1.

EQASYSPF
Specifies a list of system prefixes. This is a list of prefixes of names of CSECTs that you want Load Module Analyzer to recognize as system routines. The list helps limit the amount of output displayed for these prefixes. This file must be in fixed block record format (RECFM=FB) with a logical record length of 80 (LRECL=80). z/OS Debugger provides data for this file in member EQALMPFX of the table library (SEQATLIB). See “Description of EQASYSPF file format” on page 508 for a description of this file.

EQAPGMNM
Specifies a list of program names corresponding to program IDs found in the load module IDR data. This file must be in fixed block record format (RECFM=FB) with a logical record length of 80 (LRECL=80). z/OS Debugger provides data for this file in member EQALMPGM of the table library

(EQATLIB). See “Description of EQAPGMNM file format” on page 509 for directions on how to add entries to this list.

Description of parameters used by Load Module Analyzer

You can specify parameters by using the PARM= keyword of the EXEC JCL statement. The parameter string passed to this program can consist of any of the following parameters, separated by commas or blanks:

CKVOLFPRS
Lists only CSECTs or entries that use at least one of the Additional Floating-Point Registers 8 through 15. You cannot specify this parameter with the OSVSONLY parameter. If you specify both, the last one specified is used.

COMPOPTS
Lists the compiler options known at run time for each compile unit. Note that some compiler options are not known at run time and, in some cases, only certain sub-options of a specific option might be known at run time.

Also, the options known at run time can vary depending on the release and version of each compiler.

This option can be specified with an operand. For example:

```
COMPOPTS=';'
```

In this case, the specified character is used to end each compiler option when it is listed; this makes scanning of the options simpler.

This option applies to the following compilers only:

- Enterprise COBOL
- COBOL for MVS & VM
- VS COBOL II
- Enterprise PL/I
- z/OS XL C/C++

DATEFMT=*dateformat*
Specifies how dates are to be formatted. If a date from the binder CSECT identification record (IDR) data does not appear to be a valid Julian date, it is not reformatted. Use one of the following values:

YYYYMMDD
Sort format: YYYY/MM/DD. (Default)

MMDDYYYY
U.S. standard format: MM/DD/YYYY.

DDMMYYYY
European standard format: DD/MM/YYYY.

LEINFO
Causes the text for each CSECT and external entry point to be inspected for a Language Environment footprint. If one is found, information about the Language Environment entry point name, linkage type, source language, and translation date and time is included in the output for the CSECT or entry. If no Language Environment footprint is found, the prologue code is inspected for known non-Language Environment prologue formats. If one is discovered, the corresponding language is included in the output. Otherwise, “ASSEMBLER” is output.

In addition, for OS/VS COBOL and VS COBOL II, a NON-LEINFO section is included that contains the compile date and time and (for VS COBOL II only) the version of the compiler used.

LESCAN
Causes the actions described under the LEINFO parameter. In addition, the text for each CSECT is scanned looking for “hidden” Language Environment entry points that do not correspond to an

external symbol. For example, these might be present for C static functions. If such “hidden” entry points are detected, the same output as described for LEINFO in generated.

LISTLD
Lists all label definition (LD) entries in addition to CSECT names.

LOUD
Specifies that the data read from the EQASYSPF and EQAPGMNM files is displayed in the output listing.

NATLANG=*language_code*
Specifies the national language. Use one of the following values:

ENU
Mixed-case English. (Default)

UEN
Upper-case English.

JPN
Japanese.

KOR
Korean.

OSVSONLY
Specifies that only CSECTs compiled with the OS/VS COBOL compiler are to be displayed in the output. Information about all other CSECTs is suppressed.

You cannot specify this parameter with the CKVOLFPRS parameter. If you specify both, the last one specified is used.

SHOWLIB
Specifies that the include indicator in the EQASYSPF file is to be ignored so that all CSECTs are listed.

SORTBY=*sort_option*
Specifies how to sort the names of the CSECTs in the output. Use one of the following values:

OFFSET
Sort by offset; the order shown in the linkage editor or AMBLIST output. (Default)

NAME
Sorts by CSECT name.

PROGRAM
Sort by the translator program ID.

LANGUAGE
Sorts by the source language and by the translator program ID.

DATE
Sorts by the translation date.

Description of EQASYSPF file format

This file contains a list of system prefixes. When Load Module Analyzer finds a CSECT that has a name prefixed by a name in this list and the entry for that prefix indicates that names beginning with that prefix are not to be included, Load Module Analyzer does not display an individual entry for that CSECT. Instead, a single line is displayed in the output for each prefix found that indicates that one or more CSECTs with the specified prefix was found.

z/OS Debugger supplies data for this file in member EQALMPFX of the table library (SEQATLIB). If you want to add entries to this file, do one of the following tasks:

- Update the EQALMPFX[15] member in *hlq*.SEQATLIB through the SMP/E USERMOD in *hlq*.SEQASAMP(EQAUMOD3).

[15] USERMOD EQAUMOD3 is provided for updating EQALMPFX. See "SMP/E USERMODs" in the *IBM z/OS Debugger Customization Guide* for an SMP/E USERMOD for this customization.

- Create a data set containing the new entries. Then, concatenate this data set to the one that ships with z/OS Debugger.

Each line in this file represents one entry. The entries are free-form; however, each item must be separated from the previous item by one or more blanks. You can include comments by placing an asterisk in column 1. Use the following syntax for each line:

```
prefix   I   L   description
```

prefix
A one to seven character prefix.

I
Include indicator. Specify a "1" to indicate that each CSECT beginning with this prefix is to be treated as an ordinary CSECT. Specify a "0" to indicate that CSECTs beginning with this prefix are not to be listed individually.

L
Language or system component indicator. Choose from one of the following characters:

B
COBOL

N
Enterprise COBOL for z/OS, Version 4 or later

V
OS/VS COBOL

P
PL/I

E
Enterprise PL/I

C
C/C++

A
Assembler

L
Language Environment

S
CICS

I
IMS

2
Db2

M
MVS

T
TCP/IP

*****
Unclassified.

description
A twelve-character description of the component owning the prefix.

Description of EQAPGMNM file format

This file contains a list of program names corresponding to program IDs found in the load module IDR data. These names are used in the output to describe the language translator used to generate the object for the corresponding CSECT.

z/OS Debugger provides data for this file in member EQALMPGM of the table library (SEQATLIB). If you want to add entries to this file, do one of the following tasks:

- Update the EQALMPGM[16] member in *hlq*.SEQATLIB through the SMP/E USERMOD in *hlq*.SEQASAMP(EQAUMOD4).
- Create a data set containing the new entries. Then, concatenate this data set to the one that ships with z/OS Debugger.

Each line represents one entry. The entries are free-form. The program number must begin in column 1 and each item must be separated from the previous item by one or more blanks. You can include comments by placing an asterisk in column 1. You cannot use sequence numbers in this file. Use the following syntax for each line:

```
program_name    L   program_description
```

program_name
A seven character program number.

L
Language or system component indicator. See "Description of EQASYSPF file format" on page 508 for a list of possible values.

program_description
A description of the program.

Description of program output created by Load Module Analyzer

The output for each load module or program object is displayed in the following order:

- All members of the first EQALIB concatenation with each load module or program object appearing in alphabetical order
- All members of the second EQALIB concatenation that are not duplicates of members in the previous concatenation, with each load module or program object appearing in alphabetical order
- All members of the next EQALIB concatenation that are not duplicates of members in the previous concatenation, with each load module or program object appearing in alphabetical order

Alias names are displayed in the following manner:

- If the primary member name exists, this name is displayed in the output in the order previously described. Before the output of the contents of that member, a list of alias names for the primary member name is given.
- If the primary member name is not present in the data set, the alias is displayed the order previously described.

Description of output contents created by Load Module Analyzer

Example: Output created by Load Module Analyzer for an OS/VS COBOL load module

The following is a fragment of output that might appear for an OS/VS COBOL load module:

[16] USERMOD EQAUMOD4 is provided for updating EQALMPGM. See "SMP/E USERMODs" in the *IBM z/OS Debugger Customization Guide* for an SMP/E USERMOD for this customization.

```
5724-T07 IBM z/OS Debugger 15.0.2 Load Module Analyzer   2020/08/04  03:58  Page   2
                                 Load Module  TSCODEL.CICS.TEST.LOAD(CICK512) AMODE(31),RMODE(ANY)

 CSECT    Sg   Offset  Length  Program-ID  Trn-Date    Program-Description
 $PRIV000010
                   28     C58   5688216    1996/12/31  AD/Cycle C/370
 $PRIV000011
                  D00    1CD0   5688216    1996/12/31  AD/Cycle C/370
 @@XINIT@        29E0       8   5688216    1996/12/31  AD/Cycle C/370
 @@INIT@         29E8     3D8   5688216    1996/12/31  AD/Cycle C/370
 EQADCRXT        2DC0     240   566896201  1995/05/15  Assembler H Version 1 Release 2, 3, OR 4
 @@C2CBL         3118      10   569623400  1995/08/03  High Level Assembler for MVS & VM & VSE Version 1
 @@FETCH         3138      10   569623400  1995/08/03  High Level Assembler for MVS & VM & VSE Version 1
 MEMSET          3148      10   569623400  1995/08/03  High Level Assembler for MVS & VM & VSE Version 1
 FPRINTF         3158      10   569623400  1995/08/03  High Level Assembler for MVS & VM & VSE Version 1
 CS9403          3168    3518   566895807  1995/08/15  VS COBOL II Version 1 Release 3
 STRLEN          7398      10   569623400  1995/08/03  High Level Assembler for MVS & VM & VSE Version 1
 CEE*                           (Multiple program ID's)
 DFH*                           5668962                Assembler H Version 1 Release 2, 3, OR 4
 EDC*                           5696234                High Level Assembler for MVS & VM & VSE Version 1
 IGZ*                           5668962                Assembler H Version 1 Release 2, 3, OR 4
```

Example: Compiler options output created by Load Module Analyzer

The following is an example of the output that might be generated when LEINFO and COMPOPTS=‘;’ are in effect:

```
LEINFO=(*,COBOL,V04R02M00 2011/09/12 07:23:06)
COMPOPTS:       ADV;              QUOTE;           ARITH(COMPAT);  NOAWO;          CODEPAGE(1140);
NOCURRENCY;     DATA(31);         NODATEPROC;      DBCS;           NODECK;         NODLL;
NODUMP;         NODYNAM;          NOEXPORTALL;     NOFASTSRT;      INTDATE(ANSI);  NOLIB;
LIST;           NOMAP;            NONAME;          NONUMBER;       NUMPROC(NOPFD); OBJECT;
NOOFFSET;       NOOPTIMIZE;       OUTDD(SYSOUT);   PGMNAME(COMPAT);                RENT;
RMODE(ANY);     SEQUENCE;         SIZE(MAX);       SOURCE;         NOSSRANGE;      NOTERM;
TEST(STMT,PATH,BLOCK,NOSEPARATE);                  NOTHREAD;       TRUNC(STD);     NOVBREF;
NOWORD;         YEARWINDOW(1900);                  ZWB;
```

Appendix J. Running NEWCOPY on programs by using DTNP transaction

Note: This chapter is not applicable to IBM Developer for z/OS (non-Enterprise Edition) or IBM Z and Cloud Modernization Stack (Wazi Code).

DTNP is a CICS transaction, supplied by z/OS Debugger, that runs the NEWCOPY batch command which loads a new copy of an application program into an active CICS region.

You can run the transaction in the following ways:

- Enter the transaction name (DTNP). The transaction displays the **z/OS Debugger - NEWCOPY Program** panel. Enter the name of the application program in the **Program Name** field. To process multiple application programs at once, append the wildcard character (*) to the name. For example, LYN* indicates that you want DTNP to process all programs that start with the letters "LYN". Press PF4.
- Enter the transaction name (DTNP), followed by the name of the program. To process multiple application programs at once, append the wildcard character (*) to the name. For example, LYN* indicates that you want DTNP to process all programs that start with the letters "LYN".

The transaction displays the results in the **z/OS Debugger - NEWCOPY Program** panel. If the NEWCOPY action fails, the transaction runs the PHASEIN action, so CICS uses a new copy of the application for all new transaction requests.

Refer to the following topics for more information related to the material discussed in this topic.

Related tasks

Description of the CEMT SET PROGRAM command in *CICS Transaction Server for z/OS: Supplied Transactions*, SC34-7004.

© Copyright IBM Corp. 1992, 2023

Appendix K. Using the IBM Debug Tool plug-ins

Notes:

- This section is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).
- All the Debug Tool plug-ins are deprecated and will be removed in the next release.
- As a system administrator, you can use z/OS Debugger Profile Management to manage CICS (DTCN) profiles from all users. For more information, see "Managing profiles with z/OS Debugger Profile Management" in *IBM z/OS Debugger Customization Guide*.

The same functionality, in general, is now available in the remote IDE via these methods:

- DTCN profiles, or CICS profiles, can now be created and managed in the z/OS Debugger Profiles view.
- DTSP profiles, or non-CICS profiles, can now be created and managed in the z/OS Debugger Profiles view.
- JCL generation can be accomplished by using property groups, where available, and the various JCL generation actions in both the Remote Systems view, and the z/OS Project navigator. Users can also use the z/OS Batch Application with existing JCL launch configuration to dynamically instrument and submit JCL to the host.
- Code Coverage can now be accomplished by using the various launches or instrumenting JCL. The Code Coverage Results view allows users to work with compiled code coverage results, Java code coverage results, and supports importing code coverage results in the Debug Tool format.

The DTCN Profile Manager, DTSP Profile Manager, Instrument JCL for Debugging, Debug Tool Code Coverage, and Load Module Analyzer plug-ins are available to download from IBM Mainframe DEV as part of the IBM Developer for z/OS install offering. These plug-ins add the following views to the **Debug** perspective of the remote debugger:

- The **DTCN Profiles** view, which helps you create and manage DTCN profiles for CICS on your z/OS system.
- The **DTSP Profile** view, which helps you create and manage the TEST runtime options data set (EQAUOPTS) on your z/OS system.
- The **Instrument JCL for Debugging** view, which guides you through the process of filling out information that it uses to instrument JCL to start z/OS Debugger for batch jobs.
- The **Debug Tool Code Coverage** view, which guides you through the process that measures test coverage in application programs.
- The **Load Module Analyzer** view, which helps you determine the language translator (compiler and assembler) used to generate each CSECT in a load module or program object.

To install these plug-ins, follow the steps found in the website at Download Eclipse Tools of IBM DeveloperWorks Mainframe Development. Use the IBM Installation Manager method to install the plug-ins, and select **IBM Debug for z/OS** > **Debug Tool Plugins** when you install or modify IBM Developer for z/OS.

1. Verify that your system administrator has completed the following tasks described in the *IBM z/OS Debugger Customization Guide*:
   - "Adding support for the DTCN Profiles view and APIs"
   - "Adding support for the DTSP Profile view"
2. Restart your Eclipse-based application.

Establishing a connection between the DTCN Profiles view for CICS and your z/OS system

Specify the settings needed to establish a connection between the **DTCN profiles** view and your z/OS system by taking the following steps:

1. Select **Window** > **Show view** > **Other**.

© Copyright IBM Corp. 1992, 2023

2. Type "DTCN" in the text box at the top of the window. Select **DTCN Local profiles**, **DTCN Server Profiles**, and click **OK**.
3. Select **Window** > **Show view** > **Other**.
4. Type "Host Connections" in the text box at the top of the window. Select **Host Connections** and click **OK**.
5. In the **Host Connections** view, select **DTCN** and click **Add** to create a connection to DTCN.
6. Specify the settings in the following fields and click **Save and Close**:

   Name
   The name of the connection. It is autofilled by combining the host name and port number that you specified with ":".

   Host name
   The TCP/IP name or address of the z/OS system as described in "Defining the CICS TCPIPSERVICE resource" in the *IBM z/OS Debugger Customization Guide*.

   Port number
   The port number of the z/OS system as described in "Defining the CICS TCPIPSERVICE resource" in the *IBM z/OS Debugger Customization Guide*.

   Connection type
   If the server is not enabled with SSL as described in "Establishing a secured communication between the DTCN profile view for CICS and your z/OS system" in the *IBM z/OS Debugger Customization Guide*, select NON-SSL. Default value is NON-SSL.

   Inactivate profile
   Select Yes if you want your profile to be inactive during workbench shutting down. Default value is Yes.
7. Select the DTCN connection you created, and click **Connect**.
8. In the DTCN Signon window, specify the settings in the following fields, or select **Use existing Credentials** if you have at least one credential defined, and click **OK**.

   Credentials Name
   The name of the credential. You can leave it blank for default.

   User ID
   The ID that you use to log on to the CICS system.

   Password or Passphrase
   The password or passphrase that you use to log on to the CICS system.

   The connection is successful when you see a green icon for the DTCN connection. Otherwise, review the information you entered, correct any mistakes, and try the connection test again. You can also review the trace file (see "Locating the trace file of the DTCN Profile, the DTSP Profile, Instrument JCL for Debugging, Code Coverage, and Load Module Analyzer view" on page 524) for diagnostic information that can help identify a mistake.
9. In the **DTCN Local Profiles** view, right click **DTCN Local Profiles**, then click on **Create context** menu to create local profiles. These profiles are saved on your local workspace. The color highlighted local profile means that it is the same as server profile.

Establishing a connection between the DTSP Profile view and your z/OS system

Specify the settings needed to establish a connection between the **DTSP Profile** view and your z/OS system by taking the following steps:

1. Select **Window** > **Show view** > **Other**.
2. Type "DTSP" in the text box at the top of the window. Select **DTSP Local Profiles**, **DTSP Server Profiles**, and click **OK**.
3. Select **Window** > **Show view** > **Other**.
4. Type "Host Connections" in the text box at the top of the window. Select **Host Connections** and click **OK**.

5. In the **Host Connections** view, select **Application Delivery Foundation for z/OS** and click **Add** to create a connection to the ADFz Common Components Server.
6. Specify the settings in the following fields and click **Save and Close**:

   Name
   The name of the connection. It is autofilled by combining the host name and port number that you specified with ":".

   Host name
   The TCP/IP name or address of the z/OS system, which is set by the system administrator according to the instructions in "Installing the server components for IBM Debug Tool DTCN and DTSP Profile Manager" in the *IBM z/OS Debugger Customization Guide*.

   Port number
   The port number of the z/OS system, which is set by the system administrator according to the instructions in "Installing the server components for IBM Debug Tool DTCN and DTSP Profile Manager" in the *IBM z/OS Debugger Customization Guide*.

   Default encoding
   The default encoding is "cp037". If you use a different encoding scheme, specify it in this field.
7. Click **Window** > **Debug Tool** > **DTSP (non-CICS)** in the navigation pane.
8. In the Preferences window, select the Problem Determination Tools connection you created from the **Connection** list and click **Connect**.
9. If this is the first time you are connecting to the ADFz Common Components Server, click **Yes** in the Certificate Information window.
10. In the Application Delivery Foundation for z/OS Signon window, specify the settings in the following fields, or select **Use existing Credentials** if you have at least one credential defined, and click **OK**:

    Credentials Name
    The name of the credential. You can leave it blank for default.

    User Id
    The ID that you use to log on to the z/OS system. The **DTSP Profile** substitutes this ID for the `&userid` token in the **Profile name pattern** field.

    Password or Passphrase
    The password or passphrase that you use to log on to the z/OS system.

    If you see a message that indicates the test was successful, click **OK** to close the Preferences window. Otherwise, review the information you entered, correct any mistakes, and try the connection test again. You can also review the trace file (see "Locating the trace file of the DTCN Profile, the DTSP Profile, Instrument JCL for Debugging, Code Coverage, and Load Module Analyzer view" on page 524) for diagnostic information that can help identify a mistake.
11. In the **DTSP Local Profiles** view, right click **DTSP Local Profiles**, then click on **Create context** menu to create local profiles. These profiles are saved on your local workspace. The color highlighted local profile means that it is the same as server profile.

In the views, you can right click anywhere to see a list of actions available. If you need to change your connection settings, you can right click in any area of the **DTSP Profile** view and select **Preferences**.

Migrating to the z/OS Debugger Profiles view

The DTSP views, such as DTSP Local Profiles and DTSP Remote Profiles views, and DTCN views, such as the DTCN Local Files and DTCN Remote Profiles views, are replaced by the **z/OS Debugger Profiles** view.

About this task

When you start the product using an existing workspace, the DTCN or DTSP profiles in the workspace are automatically migrated to the **z/OS Debugger Profiles** view. When you connect to the Application Delivery Foundation for z/OS server and access one of the DTSP or DTCN views for the first time in a workspace, you are prompted to switch to the new **z/OS Debugger Profiles** view.

Procedure

To migrate your profiles, complete the following steps:

1. Open the **z/OS Debugger Profiles** view.

   Your profiles in the workspace are automatically migrated.
2. Select any of the DTCN or DTSP views.

   You are prompted with a message to switch to the **z/OS Debugger Profiles** view.

   - Select **Yes** to close the associated deprecated view and use the **z/OS Debugger Profiles** view. You might need to edit the profile before activating it if some options were not automatically determined. To restore the old view, go to the **General** > **Capabilities** preference page and select the appropriate IBM Debug Tool Plug-ins entry.
   - Select **No** to close the message and continue using the old view. If you did not select **Remember my decision**, you will be prompted again when you restart the product and click on the deprecated view.

   Note: After the message is displayed, any changes to the existing profiles or new profiles that are created with the deprecated views might not be migrated.

Instrument JCL for Debug Tool Debugging plug-in

Note: The Instrument JCL for Debug Tool Debugging plug-in is deprecated and will be removed in the next release. Use **z/OS Batch Application with existing JCL** launch configurations instead to instrument JCL and launch in debug mode. Settings from the Instrument JCL for Debug Tool Debugging view will be migrated to the **JCL Generation preference** page and used as defaults when you create new launch configurations.

The Instrument JCL for Debug Tool Debugging plug-in provides a UI that guides you through the process of filling out information that it uses to instrument JCL to start z/OS Debugger for batch jobs.

You can access the Instrument JCL for Debug Tool Debugging plug-in by taking the following steps:

1. Select **Window** > **Show view** > **Other**.
2. Type “Instrument JCL for Debug Tool Debugging” in the text box at the top of the window or scroll down until you find this entry in the drop-down menu. Select and click **OK**. The view contains the following options:

   User Settings
   Modify job card, and specify the names of the commands and preferences files.

   System Settings
   Specify library location that contains specific z/OS Debugger and Language Environment components.

   Prepare and Start Debug session
   Specify a JCL and start debug session.

   FTP Connection Settings
   Specify host name, user ID, and password for connection to server.

Customize Instrument JCL for debugging by taking the following steps:

1. In the Instrument JCL for Debug Tool Debugging view, double click **FTP Connection Settings**. Specify the settings in the corresponding fields, then click the **Test Connection** button to verify the settings, and then click **OK**.

   Host name
   The TCP/IP name or address of the z/OS system.

   User ID
   The z/OS system user id.

   Password
   The z/OS system password.

2. In the Instrument JCL for Debug Tool Debugging view, double click **User Settings** to open the User Settings view. In this view, provide the job card, session type, invocation method, and the location of the commands file and preferences file.

   Job card
   A job card image is used when there is no job card in the JCL while modifying the JCL for job submission. There is no default setting. You must enter a valid job card here.

   Session type
   Session type is the type of the display device where the debug session is displayed when z/OS Debugger starts.

   Invocation method
   Invocation method controls the method that the utility implements in the JCL to start the debugger.

   Commands file and preferences file
   They are part of the TEST run-time option string.

   After you have filled in the information, click on the **Save** button in the upper left corner of the view to save the settings.

3. In the Instrument JCL for Debug Tool Debugging view, double click **System Settings** to open the System Settings view. In this view, provide the z/OS Debugger load module library and Language Environment CEEBINIT load module library.

   z/OS Debugger Library
   If the text field is not blank, it is added to the STEPLIB DD concatenation.

   Language Environment CEEBINIT Module
   A partitioned data set that contains a member, CEEBINIT, module that has been link-edited with the z/OS Debugger version (EQAD3CXT) of the Language Environment CEEBXITA user exit.

   After you have filled in the information, click on the **Save** button in the upper left corner of the view to save the settings.

4. In the Instrument JCL for Debug Tool Debugging view, double click **Debug Information, Source and Listing Files**. In this view, provide a user level file list and an installation file list.

   User level file list
   The files listed in the data set are added to the top of the EQADEBUG DD concatenation.

   Installation file list
   The files listed in the data set are added to the bottom of the EQADEBUG DD concatenation.

   After you have filled in the information, click on the **Save** button in the upper left corner of the view to save the settings.

5. In the Instrument JCL for Debug Tool Debugging view, double click **Prepare and Start Debug Session**. A wizard guides you through a set of steps required for the JCL Instrumentation for debugging. Specify the settings in the following wizard pages.

   Wizard Page 1
   Specify a partially qualified data set name in the field as a filter, then click the **Select** button to retrieve a list of data set names. Only partitioned or sequential data set names are supported. After the JCL data set name is selected, click **Next**.

   Wizard Page 2
   If the JCL data set is a partitioned data set, select one of the members on this page and click **Next**. If it is a sequential data set from the previous page, Wizard page 3 displays.

   Wizard Page 3
   A list of steps in the selected JCL is displayed. Session type and invocation method are displayed on top of the page. A **Find** icon is available to locate a character string in the step list. Select one of the steps, session type and invocation method, then click **Next**.

   Wizard Page 4 GUI
   If the session type selected is GUI from the previous page, a GUI page is displayed with the IP address of your workstation and the default port number. You can enter a different work station's

IP address and port number if you want. The workstation must have the remote debugger installed and listen on the port number. Click **Next**.

Wizard Page 4 TIM
If the session type is TIM from the previous page, the TIM page displayed. TIM is the z/OS Debugger TERMINAL INTERFACE MANAGER. Enter the user id to login the TIM terminal, then click **Next**. The default value is the FTP user id.

Wizard Page 5
The updated JCL is displayed and ready to submit. To assist viewing the added lines, all insertions are enclosed in the comment lines:

```
//*>>> The JCL lines below were inserted by z/OS Debugger. <<<
//*>>> The JCL lines above were inserted by z/OS Debugger. <<<
```

Wizard Page 6
Enter a data set name if you want to save the updated JCL, then Click **Next**.

Last Wizard Page
JCL job is submitted. Job id is displayed.

Usage notes

1. JCLs that use PROC are not supported.
2. If the invocation method described above in **Prepare and Start Debug Session** Wizard 3 uses the z/OS Debugger Language Environment user exit EQAD3CXT, see the *Specifying the TEST runtime options through the Language Environment user exit* chapter in the *IBM z/OS Debugger User's Guide* or the *IBM z/OS Debugger Customization Guide* for further details about how to use the user exit.
3. The plug-in's default is to start a debug session by using the Debug perspective in the remote IDE. However, you can direct the debug session to a Terminal Interface Manager session. See Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal for more information on using Terminal Interface Manager.

Debug Tool Code Coverage plug-in

Note: The Debug Tool Code Coverage plug-in is deprecated and will be removed in the next release. The legacy results can be imported into the **Code Coverage Results** view and you can use profiles to configure code coverage collection.

Debug Tool Code Coverage plug-in is a UI application that guides you through the process that measures test coverage in application programs which are written in COBOL, PL/I and C and are compiled with certain compilers and compiler options. With this UI application in PD Tool Studio, you can test your application and generate reports to determine which code statements have been executed and unexecuted.

You can access Debug Tool Code Coverage plug-in by taking the following steps:

- Select **Window** > **Show view** > **Other**.
- Type `Debug Tool Code Coverage` in the text box at the top of the window or scroll down until you find this entry in the drop-down menu. Select and click **OK**. The view contains the following options:

  Debug Tool Code Coverage Option Files
  Modify the z/OS Debugger code coverage options.

  Code Coverage Report Generation
  Create code coverage reports.

Establishing a connection between the Code Coverage view and your z/OS system

1. Select "Host connections view".
2. In the **Host Connections** view, select **Application Delivery Foundation for z/OS** and click **Add** to create a connection to the ADFz Common Components Server.
3. Specify the settings in the following fields and click **Save and Close:**

Name
The name of the connection. It is auto filled in by combining the host name and port number that you specified with a ":".

Host name
The TCP/IP name or address of the z/OS system, which is set by the system administrator according to the instructions in "Server Overview" in the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

Port number
The port number of the z/OS system, which is set by the system administrator according to the instructions in "Server Overview" in the *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

Default encoding
The default encoding is "cp037". If you use a different encoding scheme, specify it in this field.

4. If this is the first time you are connecting to the ADFz Common Components Server, click Yes in the Certificate Information window.
5. In the Problem Determination Tools Signon window, specify the settings in the following fields, or select **Use existing Credentials** if you have at least one credential defined, and click **OK**:

Credentials Name
The name of the credential. You can leave it blank for the default.

User Id
The ID that you use to log on to the z/OS system.

Password or Passphrase
The password or passphrase that you use to log on to the z/OS system.

Generate code coverage reports by taking the following steps:

1. In the Debug Tool Code Coverage view, double click **Code Coverage Options File** to open the option file editor. Specify the settings in the corresponding fields, then click **Create | Update**.

Data set name
You can enter an existing or new data set name, then click the **Select...** button to retrieve a data set list. Before you click the button, specifying a partially qualified data set name is highly recommended. Retrieving all data sets from the server may be time-consuming. The partially qualified data set name used as a filter must begin with first qualifier of the data set you are looking for.

Program name
The name of program targeted for code coverage. Up to 8 names are allowed. You can use a wild card either at the end the name string or standalone if you want all programs in the application.

Group ID
Group ID 1: If you want to group observations to form a set based on the characteristics of the applications, you can use this field.

Group ID 2: If you want a subgroup for the observation to form a subset based on the characteristics of the application, you can use this field. During the analysis of the observations the user can sort based on the grouping.

2. In the Debug Tool Code Coverage view, double click **Code Coverage Report Generation**. A wizard guides you through code coverage report generation. Specify the settings in the following wizard pages.

Wizard Page1
Specify a partially qualified data set name in the field as a filter, then click the **Select...** button to retrieve a list of data set names. Before you click the **Select...** button, specifying a partially qualified data set name is highly recommended. Retrieving all data sets from the server maybe time-consuming. The partially qualified data set name used as a filter must begin with first qualifier of the data set you are looking for. After the output data set name is selected, click **Next**.

Wizard Page 2
In the Observation selection criteria page, all fields are pre-populated based on the file you selected from the previous page. The selection wizard page is where you specify the criteria that should be used in the evaluation of the code coverage observations in order to create a set of statistics based on the selection provided. You might want to see only the results for a specific group, or a specific program even if the Options file indicated more than one program. This allows the user to define the granularity of the information. You can specify one or more attribute values and their associated comparison operator. After specify selection criteria, click **Next** or **Finish**.

Wizard Page 3
This Source Marker Page is optional. The source markers provide a way to select source lines that are to be included or excluded in the statistics calculation for code coverage. It is based on the indicators in the source listing like a comment, numeric sequence, a range of statements, or a string at a specific place in the source listing. An indicator marks a statement or section of statements that are changed/added as a result of a defect fix or enhancement. Click **Finish** after you have done with it.

3. In the Debug Tool Code Coverage report view, there are four menu icons on top of the view and three context menu items.

Expand All
Expand the tree view to the lowest level.

Collapse All
Collapse the tree view to the top level.

Export to XML
Save the report as XML.

Export to PDF
Save the report as a PDF. This report includes the program source.

View/Update Selection Criteria
Right click on the top level of the tree view to display the context menu **View/Update Selection Criteria**. Click this menu item to display the Observation Selection Criteria wizard page, where you can update your selections again. Notice that all fields are pre-populated based on your previous selections.

View Annotated Source
Right click on the lowest level of tree item to display two menu items. One of them is View Annotated Source. Executed lines are highlighted with green color, and unexecuted lines are highlighted with red color.

View Statistics
Right click on the lowest level of tree item to display two menu items. Another one is View Annotated Source.

Load Module Analyzer plug-in

Note: The Load Module Analyzer plug-in is deprecated and will be removed in the next release.

Load Module Analyzer plug-in (LMA) is a UI application that is used to determine the language translator (compiler and assembler) used to generate each CSECT in a load module or program object. In addition it can display the compiler options for high-level languages and a variety of other information.

You can access the Load Module Analyzer plug-in by taking the following steps:

1. Select **Window** > **Show view** > **Other**.
2. Type “Load Module Analyzer” in the text box at the top of the window or scroll down until you find this entry in the drop-down menu. Select and click **OK**. The view contains the following options:

Load Module Analyzer Report Generation
Create load module analyzer reports.

Refresh Current User
Display available reports for the current logged in user.

Establishing a connection between the Load Module Analyzer view and your z/OS system (refer to Code Coverage Plug-in section)

Generate Load Module Analyzer reports by taking the following steps:

1. In the Load Module Analyzer view, click **Launch Report Generating Wizard**. A wizard guides you through the Load Module Analyzer report generation. Specify the settings in the following wizard.

   Wizard Page1
   Specify a partially qualified data set name in the field as a filter, then click the **Select...** button to retrieve a list of data set names. Before you click the **Select...** button, specifying a partially qualified data set name is highly recommended. Retrieving all data sets from the server maybe time-consuming. The partially qualified data set name used as a filter must begin with first qualifier of the data set you are looking for. After the output data set name is selected, click **Next**.

   Wizard Page 2
   Report Preferences page, fill in the designated fields you would like for your report, then click **Next**. There are several fields:

   Display prefix and program data
   Allows you to see the list of system prefixes and program names known by the Load Module Analyzer program.

   Show information for all compiler/system library routines
   Allows you to see information about all system and library routines instead of a summary by prefix.

   Show all label definitions
   Allows you to show all external names including both CSECT's and label definitions.

   Show compiler options
   Allows you to show all the compiler options known at run-time for CU's generated by certain compilers.

   OS/VS COBOL only
   Allows you to limit output to only OS/VS COBOL programs.

   CKVOLFPRS
   Allows you to limit the output to only programs that may contain references to volatile floating point registers.

   Show language environment information
   Allows you to show information extracted from the Language Environment prologue blocks.

   Scan for language environment information
   Allows you to show information extracted from the Language environment prologue blocks and to scan for Language Environment entry points that do not correspond to external names.

   Sort by
   Allows you to sort the output for each load module by OFFSET in the load module, CU NAME, PROGRAM ID, LANGUAGE (COBOL, C/C++, PL/I, etc.), or translation DATE.

   Date format
   This option specifies the date format to be used in program output.

   Wizard Page 3
   The page is used to display a list of the members of a partitioned data set. Select the members whose contents you would like to view. You can select individually or select all, click **Next**.

   Wizard Page 4
   Confirmation page, there are several sections.

   Report name field
   Modify or use prefilled name.

Report summary
Display selections from previous wizard pages.

Report preferences
Display selections from previous wizard pages.

Save report to local directory
Click checkbox “Save to Local Files” will enable “Select File Directory” button.

2. In the Load Module Analyzer view, you can see the report list for current user by pressing **Refresh Current user** button. You can open them either by double clicking the reports from list or clicking **Open Selection(s) in Report View**. To remove reports from the list, select reports, then click **Remove Selection(s) from User Reports**.

Locating the trace file of the DTCN Profile, the DTSP Profile, Instrument JCL for Debugging, Code Coverage, and Load Module Analyzer view

When you do actions in the **DTCN Profiles**, **DTSP Profile**, **Instrument JCL for Debugging**, **Code Coverage**, or **Load Module Analyzer** view, the views save information about the actions and results of the actions in the following files:

- `.com.ibm.pdtools.debugtool.dtcn.trace.log` for the **DTCN Profiles** view
- `.com.ibm.pdtools.debugtool.dtsp.trace.log` for the **DTSP Profiles** view
- `.com.ibm.pdtools.debugtool.bjfd.trace.log` for the **Instrument JCL for Debugging** view
- `.com.ibm.pdtools.debugtool.dtcc.trace.log` for the **Code Coverage** view
- `.com.ibm.pdtools.debugtool.dtlma.trace.log` for the **Load Module Analyzer** view

The views save these files in the `\.metadata` folder of your workspace. (To find the path name of your workspace, click **File>Switch Workspace>Other...** in your Eclipse-based application.) The example below shows the file information about a common action and the action result.

Example: .debugtool.dtcn.trace file

```
<?xml version="1.0" encoding="UTF-8"?>
<profilerecord>
<profileid>DCRAGGS</profileid>
<activation>A</activation>
<program>
<loadname>LOAD1</loadname>
<pgmname>COMP1</pgmname>
</program>
<program>
<loadname>LOAD2</loadname>
<pgmname>COMP2</pgmname>
</program>
<program>
<loadname>LOAD3</loadname>
<pgmname>COMP3</pgmname>
</program>
<transactionid>T67</transactionid>
<userid>DCRAGG</userid>
<netname>net</netname>
<clientip>123.123</clientip>
<commareaoffset>0</commareaoffset>
<containeroffset>0</containeroffset>
<urmdeb>Y</urmdeb>
<trigger>TEST</trigger>
<level>ALL</level>
<sesstype>TCP</sesstype>
<sessaddr>DBM</sessaddr>
<sessport>8001</sessport>
<commandfile>TEST</commandfile>
<preferencefile>*</preferencefile>
<promptlevel>PROMPT</promptlevel>
<pmplatform>PlatName</pmplatform>
</profilerecord>
```

The last line of the trace is one line; however, the line is wrapped in this example so that you can see the entire contents of the line.

Examples: .debugtool.dtsp.trace files

The following example shows what the file might contain after you click on **Test Connection** in the **DTSP (non-CICS) Preferences** page:

```
Test Connection button clicked -----
getSocketIO parameters are below.
Host: tlba07me.torolab.ibm.com
Port: 5555
UserId: vikram
Pattern: &userid.dbgtool.eqauoptsStart Service successful. The message was:
 Connected to DebugToolProvider DTSP query response: File exists.
 Connection was successful ---
```

The following example shows what the file might contain after you click on **Finish** in the update wizard:

```
---- DTSP Finish button clicked ----
Profile data set: vikram2.dbgtool.eqauopts
UEWizard: Read successful.
DT_Update request worked fine. ------
Retrieving Profile -----
GetOtherProfiles: Socket is good -----
GetOtherProfiles: Hashmap contains {otheropts=sto(ff), sessport=8002,
 sessaddr=9.65.111.33, level=ERROR, preferencefile=*, commandfile=*,
 trigger=TEST, sesstype=TCPIP, profiledata set=vikram2.dbgtool.eqauopts}
```

Examples: .debugtool.bjfd.trace files

The following example shows what the file might contain after you click on **Save** Icon in the Setting Editor view:

```
[2013-09-26 10:18:11,633] 590320 INFO
-- logging in to FTP server
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.connect(FTPJobManager.java:74) ]
[2013-09-26 10:18:12,144] 590831 INFO
-- login succeeded
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.connect(FTPJobManager.java:79) ]
[2013-09-26 10:18:12,145] 590832 DEBUG
-- Buffer Size:1024
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:153) ]
[2013-09-26 10:18:12,146] 590833 DEBUG
-- Buffer Size:1048576
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:155) ]
[2013-09-26 10:18:12,147] 590834 DEBUG
-- 230 JSMITH is logged on.  Working directory is "/home/jsmith".
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:161) ]
[2013-09-26 10:18:12,147] 590834 DEBUG
-- Filter specified:JSMITH.EOI.FILE*
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:166) ]
[2013-09-26 10:18:12,349] 591036 DEBUG
-- 250 "JSMITH.EOI." is the working directory name prefix.
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:180) ]
[2013-09-26 10:18:13,167] 591854 DEBUG
-- 250 List completed successfully.
--[ com.ibm.pdtools.bjfd.controller.ftp.FTPJobManager.getDataSet(FTPJobManager.java:183) ]
[2013-09-26 10:18:17,718] 596405 DEBUG
-- Setting is added to setting manager
--[ com.ibm.pdtools.bjfd.model.setting.SettingManager.addSetting(SettingManager.java:76) ]
```

The following example shows what the file might contain after you click on the **Next** button in the **Prepare and Start Debug Session** wizard page 4:

```
[2013-09-26 10:29:47,516] 1286203 DEBUG
-- Text to be inserted:TCPIP&9.65.131.12%8001:
--[ com.ibm.pdtools.bjfd.ui.wizards.DebugSessionWizardPage5.setVisible
(DebugSessionWizardPage5.java:105) ]
[2013-09-26 10:29:47,519] 1286206 DEBUG
-- working directory:C:\Apps\workSpace\eclipse421\runtime-New_configuration3\parser\
--[ com.ibm.pdtools.bjfd.ui.wizards.DebugSessionWizardPage5.setVisible
(DebugSessionWizardPage5.java:113) ]
```

```
[2013-09-26 10:29:47,636] 1286323 DEBUG
-- Extracting text
--[ com.ibm.pdtools.bjfd.ui.actions.FileControlManager.extractText(FileControlManager.java:95) ]
[2013-09-26 10:29:47,647] 1286334 DEBUG
-- writing to JCL file
--
[ com.ibm.pdtools.bjfd.ui.actions.FileControlManager.writeJCLFile(FileControlManager.java:161) ]
[2013-09-26 10:29:47,648] 1286335 DEBUG
-- Populdated steplib text://* >>>The JCL lines above were inserted by Debug Tool.<<<
--[ com.ibm.pdtools.bjfd.ui.actions.UIManager.populateStepLibText(UIManager.java:270) ]
[2013-09-26 10:29:47,649] 1286336 DEBUG
-- Modifiled text needs to be inserted:
//GO  EXEC PGM=ECOB420
//STEPLIB  DD DISP=SHR,DSN=JSMITH.TEST.LOAD
//* >>>The JCL lines above were inserted by Debug Tool.<<<
//         DD DISP=SHR,DSN=ESFLINT.CEEV1RDZ.SCEERUN
//         DD DISP=SHR,DSN=ESFLINT.CEEV1RDZ.SCEERUN2
//SYSPRINT     DD SYSOUT=*
//SYSUDUMP     DD DUMMY
//SYSOUT       DD SYSOUT=*
/*
//* >>>The JCL lines below were inserted by Debug Tool.<<<
//CEEOPTS DD *,DLM='/*'
TEST(ALL,'-JSMITH.EOI.INSPIN(EOI1)',PROMPT,
TCPIP&9.65.131.12%8001:-JSMITH.EOI.INSPPREF)
//INSPLOG DD SYSOUT=*
//* >>>The JCL lines above were inserted by Debug Tool.<<<
--
[ com.ibm.pdtools.bjfd.ui.actions.FileControlManager.writeJCLFile(FileControlManager.java:244) ]
[2013-09-26 10:29:47,652] 1286339 DEBUG
-- Writing to output file
--
[ com.ibm.pdtools.bjfd.ui.actions.FileControlManager.writeJCLFile(FileControlManager.java:262) ]
```

Appendix L. Debugging a program processed by the Automatic Binary Optimizer for z/OS

Note: This chapter is not applicable to IBM Z and Cloud Modernization Stack (Wazi Code).

For programs complied with Enterprise COBOL for z/OS Version 5 or later and optimized by Automatic Binary Optimizer for z/OS (ABO), you can debug them in the same manner as the programs that are not optimized by ABO. LangX COBOL support cannot be used for those programs.

For programs complied with Enterprise COBOL for z/OS Version 4 or earlier, you can use the LangX COBOL support in z/OS Debugger to debug (with restrictions) a load module or program object generated by the ABO.

Debugging an ABO-optimized program with LangX COBOL support

Before you debug, you must use the IPVLANGO utility to create a new LangX file. For more information about the IPVLANGO Automatic Binary Optimizer LangX file update utility, see *IBM Application Delivery Foundation for z/OS Common Components Customization Guide and User Guide*.

ABO shuffles or removes instructions, which might result in moving or removing one or more statements, or collapsing several statements into one single statement. As a result, the debugging behavior might not be predictable and a visual random stepping can be experienced during the debug session. In addition, it may not be obvious when a variable is actually set.

For an example of this potential unpredictable behavior, see Example of potential unpredictable behavior when debugging an Automatic Binary Optimizer (ABO) optimized COBOL load module (http://www.ibm.com/support/docview.wss?uid=swg21971749).

To debug an ABO-processed program, complete the following steps:

1. Compile the source using Enterprise COBOL for z/OS Version 3 or Version 4 with the options needed for LangX COBOL support.

   For more information about the LangX COBOL support, see Chapter 6, “Preparing a LangX COBOL program,” on page 67.
2. Link or bind your program.
3. Run IPVLANGX against the COBOL listing from Step 1 to create a LangX file.

   Steps 1, 2, and 3 are the normal LangX COBOL program preparation.
4. Run ABO against the output from Step 2 to generate an ABO listing and an optimized program.
5. Run IPVLANGO against the LangX file from Step 3 and the listing output from Step 4 to create a new LangX file.
6. Debug the program generated by ABO in Step 4 with the LangX file from Step 5.

© Copyright IBM Corp. 1992, 2023

Appendix M. Limitations of 64-bit support in Debug Tool compatibility mode

Debug Tool compatibility mode, a subset of remote debug mode, supports 64-bit COBOL and C/C++ programs with some limitations. For 64-bit PL/I programs, use standard mode, another subset of remote debug mode, instead.

Subsystems IMS, Db2, and CICS are not supported.

The following functions are not supported for 64-bit COBOL and C/C++ programs in Debug Tool compatibility mode:

- The EQAUEDAT user exit, EQA_DBG_PATH environment variable, and EQA_SRC_PATH environment variable
- Language Environment user exit (CEEBXITA)
- Compiled in hooks

In addition to all the commands strictly related to the unsupported subsystems and functions mentioned above, the following commands are not supported for 64-bit COBOL and C/C++ programs in Debug Tool compatibility mode:

- z/OS Debugger commands:
  - CALL %FA
  - SET DYNDEBUG
- EQAOPTS commands:
  - EQAQPP
  - DLAYDBGCND
  - DLAYDBGXRF
  - DYNDEBUG
  - SVCSCREEN

© Copyright IBM Corp. 1992, 2023

Appendix N. Debugging programs compiled with IBM Open Enterprise SDK for Go

IBM Open Enterprise SDK for Go is an industry-standard Go compiler that you can use to build Go programs for the z/OS platform. IBM z/OS Debugger supports debugging programs compiled with IBM Open Enterprise SDK for Go 1.17.

Note: Go programs are currently not supported in IBM Z Open Debug that is provided in Wazi for VS Code and Wazi for Dev Spaces.

`go build` command

With each `go build` action, the Go compiler creates a single load module that includes all runtime package functions required to run the application packages. The Go compiler and assembler produce DWARF debugging data by default, which is embedded as NOLOAD (no load) classes in the resulting program object. You cannot produce a separate DWARF side file.

The `go build` command passes arguments to the underlying compiler, assembler, and linker tools by using the following options:

-gcflags Compiler flags

`-dwarf=true` is required and specified as the default.

`-dwarflocationlists=true` is required and specified as the default.

-asmflags Assembler flags

No assembler options affect the debugger.

-ldflags Linker flags

Do not specify `-s` because it disables the generation of the DWARF symbol table.

Do not specify `-w` because it disables the embedding of DWARF information in the program object.

Debugging a Go program

The Go build process produces a Language Environment-enabled program object with entry point `_rt0_s390x_zos`. To debug a Go program object, you must use the Language Environment TEST option.

Go programs only run under UNIX System Services.

1. You can use the following shell script called `dtrun` to debug:

   ```
   export STEPLIB=EQAW.SEQAMOD:$STEPLIB
   export GOMAXPROCS=1
   export _CEE_RUNOPTS="TEST(ERROR,*,PROMPT,DBMDT:*)"
   $@
   ```

   Note: This shell script assumes that you have logged on to the z/OS system with the Remote System Explorer, and that Debug Manager is activated on the system. If Debug Manager is not available, you can code the TEST option as `TEST(ERROR,*,PROMPT,TCPIP&`*`your.ip.address%your-debug-port`*`:*)`.

2. Invoke the debugger for your Go program. You can use the following command to run the shell script:

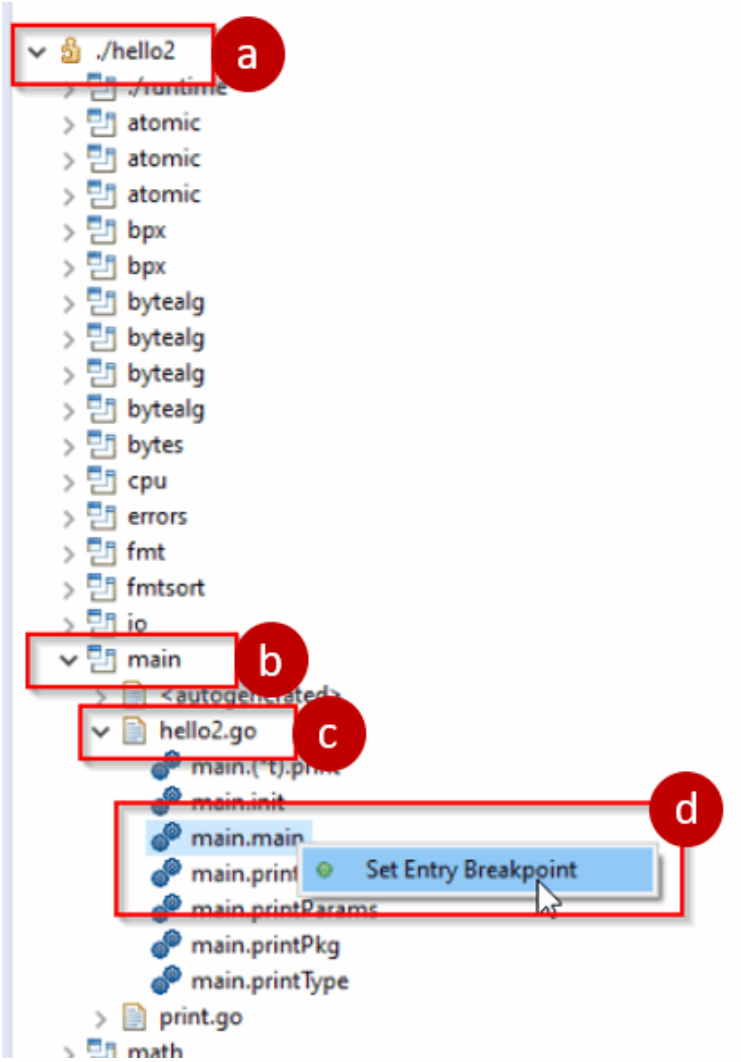
   ```
   dtrun hello
   ```

   The debugger starts at the main Language Environment entry point. For all Go programs, the entry point is the function `runtime._rt0_s390x_zos`.

3. To proceed to your application code, set an entry breakpoint and click **Resume**. You can use the **Modules** view to set the breakpoint.

© Copyright IBM Corp. 1992, 2023

For example, to set an entry breakpoint for `main.main` in the Go program object `hello2`, complete the following steps:



a. Expand the Go program object.

b. Expand the `main` compile unit.

c. Expand the source file where the main function is defined.

d. Right-click `main.main` and select **Set Entry Breakpoint**.

Debug features that are not supported for Go programs

- Compiled code coverage
- Delay debug
- Debug console commands
- Conditional breakpoints
- Load breakpoints
- Source entry breakpoints
- Watch breakpoints
- Actions on breakpoints
- Stop at all function entries
- Run to location

- Jump to location
- Hover
- Automonitor filter
- Playback
- Visual debugging

Go features that are not supported in debugging

- cgo
- Go assembler
- Debugging artifacts of `go test`
- Channel expressions
- Expressions containing Go functions
- Go application code that runs in more than one thread. If you set `GOMAXPROCS=1`, z/OS Debugger can debug your whole application. Otherwise, any code running in a thread other than the initial one is not debugged.

Known issues and limitations

- Expression evaluation for non-top stack frame local (auto) variables is not supported.
- STEP OVER might behave like STEP INTO when the callee is inlined.

For more information about known issues and limitations for IBM Open Enterprise SDK for Go, see Known issues and limitations.

Appendix O. Support resources and problem solving information

This section shows you how to quickly locate information to help answer your questions and solve your problems. If you have to call IBM support, this section provides information that you need to provide to the IBM service representative to help diagnose and resolve the problem.

Accessing the IBM Support portal

You must be a registered user on the IBM Support portal to download fixes and to submit a problem online to the IBM Support community.

If you need to look beyond IBM Documentation to answer your question or resolve your problem, you can use one or more of the following approaches:

- Open the IBM Support portal.
- Click to log in using your IBM.com username.
- On the IBM Support portal, you can do the following tasks:
  - Search for known issues, documentation, and support forums.
  - Open a case or view cases you opened.
  - Open a chat window with a Support representative.
  - Open Fix Central to view product downloads and updates.
  - Access product documentation and support forums.
  - Manage your support account, including notifications, invoices, orders, contracts, and warranties. For more information about notifications, see “Subscribing to support updates” on page 535.

Getting fixes

A product fix might be available to resolve your problem. To determine what fixes and other updates are available, select a link from the following list:

- Latest PTFs for z/OS Debugger
- Latest PTFs for IBM Developer for z/OS Enterprise Edition
- Latest PTFs for ADFz Common Components

When you find a fix that you are interested in, click the name of the fix to read its description and to optionally download the fix.

Subscribe to receive e-mail notifications about fixes and other IBM Support information as described in Subscribing to Support updates.

Subscribing to support updates

To receive automatic updates when IBM publishes new support content for your products, subscribe to weekly email updates or RSS feeds. Support content might include information about new releases, fixes, technotes, APARs, and support flashes.

To sign up for email updates, you must be a registered user on the IBM Support community website.

To subscribe to Support updates, follow the steps below.

1. Open the IBM Support portal website.
2. Click to log in using your IBM.com username.

© Copyright IBM Corp. 1992, 2023

3. Click **Manage support account** > **Notifications** to view your notifications.
4. Type the product name in the search field or click **Browse for a product**.
5. Type the product name in the **Product lookup** field, or click **Browse for a product**.
6. Click **Subscribe** beside your product, and in the **Select document types** window, select the types of documents for which you want to receive information. Click **Submit**.
7. Optionally, you can click the RSS/Atom feed by clicking **Links**. Then, copy and paste the link into your feeder.
8. To see any notifications that were sent to you, click **View**.

Contacting IBM Support

To submit your problem to IBM Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online on the Passport Advantage website.

- To learn more about Passport Advantage, see the Passport Advantage FAQs.
- For further assistance, contact your IBM representative.

To submit your problem online (from the IBM website) to IBM Support:

- Be a registered user on the IBM Support website. For details about registering, see Registering on the IBM Support website.
- Be listed as an authorized caller in the service request tool.

Determine the business impact of your problem

When you report a problem to IBM, you are asked to supply a severity level. Therefore, you must understand and assess the business impact of the problem that you are reporting.

Severity 1
The problem has a *critical* business impact: You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.

Severity 2
This problem has a *significant* business impact: The program is usable, but it is severely limited.

Severity 3
The problem has *some* business impact: The program is usable, but less significant features (not critical to operations) are unavailable.

Severity 4
The problem has *minimal* business impact: The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

Gather diagnostic information

To save time, if there is a MustGather document available for the product, refer to the MustGather document and gather the information specified. MustGather documents contain specific instructions for submitting your problem to IBM and gathering information needed by the IBM support team to resolve your problem. To determine if there is a MustGather document for this product, go to the product support page and search on the term MustGather. At the time of this publication, the following MustGather documents are available:

- MustGather: Read first for problems encountered with z/OS Debugger: https://www.ibm.com/support/pages/node/89125
- MustGather: Read first for problems encountered with code coverage: https://www.ibm.com/support/pages/node/6561317

If the product does not have a MustGather document, provide answers to the following questions:

- What software versions were you running when the problem occurred?
- Do you have logs, traces, and messages that are related to the problem symptoms? IBM Software Support is likely to ask for this information.
- Can you re-create the problem? If so, what steps were performed to re-create the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, and so on.
- Are you currently using a workaround for the problem? If so, be prepared to explain the workaround when you report the problem.

Submit the problem to IBM Support

You can submit your problem to IBM Support in the following ways:

- Online: Open the IBM Support community website. Click **Open a case** to open a service request and describe the problem in detail.
- By phone: For the phone number to call in your country or region, see the IBM Directory of worldwide contacts and click the name of your country or geographic region.
- Through your IBM Representative: If you cannot access IBM Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at IBM Directory of worldwide contacts.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Support website daily, so that other users who experience the same problem can benefit from the same resolution.

Appendix P. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The accessibility features in z/OS provide accessibility for z/OS Debugger.

The major accessibility features in z/OS enable users to:

- Use assistive technology products such as screen readers and screen magnifier software
- Operate specific or equivalent features by using only the keyboard
- Customize display attributes such as color, contrast, and font size

IBM Documentation, and its related publications, are accessibility-enabled. The accessibility features of the information center are described at https://www.ibm.com/docs.

Using assistive technologies

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

Keyboard navigation of the user interface

Users can access z/OS user interfaces by using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Volume 1* for information about accessing TSO/E and ISPF interfaces. 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.

Accessibility of this document

Information in the following format of this document is accessible to visually impaired individuals who use a screen reader:

- HTML format when viewed from IBM Documentation
- PDF format

Syntax diagrams start with the word Format or the word Fragments. Each diagram is preceded by two images. For the first image, the screen reader will say "Read syntax diagram". The associated link leads to an accessible text diagram. When you return to the document at the second image, the screen reader will say "Skip visual syntax diagram" and has a link to skip around the visible diagram.

© Copyright IBM Corp. 1992, 2023

Notices

This information was developed for products and services offered in the U.S.A. IBM might 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 in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
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 the 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 might 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.

Copyright license

This information contains sample application programs in source language, which illustrate 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year).

© Copyright IBM Corp. 1992, 2023

Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies", and "the IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Programming interface information

This document is intended to help you debug application programs. This publication documents intended Programming Interfaces that allow you to write programs to obtain the services of z/OS Debugger.

Trademarks and service marks

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.

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

Glossary

This glossary defines technical terms and abbreviations used in *IBM z/OS Debugger User's Guide* documentation. If you do not find the term you are looking for, refer to the *IBM Glossary of Computing Terms*, located at the IBM Terminology web site:

```
http://www.ibm.com/ibm/terminology
```

A

active block
The currently executing block that invokes z/OS Debugger or any of the blocks in the CALL chain that leads up to this one.

active server
A server that is being used by a remote debug session. Contrast with *inactive server*. See also *server*.

alias
An alternative name for a field used in some high-level programming languages.

animation
The execution of instructions one at a time with a delay between each so that any results of an instruction can be viewed.

attention interrupt
An I/O interrupt caused by a terminal or workstation user pressing an attention key, or its equivalent.

attention key
A function key on terminals or workstations that, when pressed, causes an I/O interrupt in the processing unit.

attribute
A characteristic or trait the user can specify.

Autosave
A choice allowing the user to automatically save work at regular intervals.

B

batch
Pertaining to a predefined series of actions performed with little or no interaction between the user and the system. Contrast with *interactive*.

batch job
A job submitted for batch processing. See *batch*. Contrast with *interactive*.

batch mode
An interface mode for use with the MFI z/OS Debugger that does not require input from the terminal. See *batch*.

block
In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it.

breakpoint
A place in a program, usually specified by a command or a condition, where execution can be interrupted and control given to the user or to z/OS Debugger.

C

CADP
A CICS-supplied transaction used for managing debugging profiles from a 3270 terminal.

© Copyright IBM Corp. 1992, 2023

century window (COBOL)
The 100-year interval in which COBOL assumes all windowed years lie. The start of the COBOL century window is defined by the COBOL YEARWINDOW compiler option.

command list
A grouping of commands that can be used to govern the startup of z/OS Debugger, the actions of z/OS Debugger at breakpoints, and various other debugging actions.

compile
To translate a program written in a high level language into a machine-language program.

compile unit
A sequence of HLL statements that make a portion of a program complete enough to compile correctly. Each HLL product has different rules for what comprises a compile unit.

compiler
A program that translates instructions written in a high level programming language into machine language.

condition
Any synchronous event that might need to be brought to the attention of an executing program or the language routines supporting that program. Conditions fall into two major categories: conditions detected by the hardware or operating system, which result in an interrupt; and conditions defined by the programming language and detected by language-specific generated code or language library code. An example of a hardware condition is division by zero. An example of a software condition is end-of-file. See also *exception*.

conversational
A transaction type that accepts input from the user, performs a task, then returns to get more input from the user.

currently qualified
See *qualification*.

D

data type
A characteristic that determines the kind of value that a field can assume.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

date field
A COBOL data item that can be any of the following:

- A data item whose data description entry includes a DATE FORMAT clause.
- A value returned by one of the following intrinsic functions:

  DATE-OF-INTEGER
  DATE-TO-YYYYMMDD
  DATEVAL
  DAY-OF-INTEGER
  DAY-TO-YYYYDDD
  YEAR-TO-YYYY
  YEARWINDOW
- The conceptual data items DATE and DAY in the ACCEPT FROM DATE and ACCEPT FROM DAY statements, respectively.
- The result of certain arithmetic operations.

The term date field refers to both *expanded date field* and *windowed date field*. See also *nondate.*.

date processing statement
A COBOL statement that references a date field, or an EVALUATE or SEARCH statement WHEN phrase that references a date field.

DBCS
See *double-byte character set*.

debug
To detect, diagnose, and eliminate errors in programs.

DTCN
z/OS Debugger Control utility, a CICS transaction that enables the user to identify which CICS programs to debug.

z/OS Debugger procedure
A sequence of z/OS Debugger commands delimited by a PROCEDURE and a corresponding END command.

z/OS Debugger variable
A predefined variable that provides information about the user's program that the user can use during a session. All of the z/OS Debugger variables begin with %, for example, %BL0CK or %CU.

debugging profile
Data that specifies a set of application programs which are to be debugged together.

default
A value assumed for an omitted operand in a command. Contrast with *initial setting*.

double-byte character set (DBCS)
A set of characters in which each character is represented by two bytes. Languages such as Japanese, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires two bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support these characters.

dynamic
In programming languages, pertaining to properties that can only be established during the execution of a program; for example, the length of a variable-length data object is dynamic. Contrast with *static*.

dynamic link library (DLL)
A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously. See also *load module*.

E

enclave
An independent collection of routines in Language Environment, one of which is designated as the MAIN program. The enclave contains at least one thread and is roughly analogous to a program or routine. See also *thread*.

entry point
The address or label of the first instruction executed on entering a computer program, routine, or subroutine. A computer program can have a number of different entry points, each perhaps corresponding to a different function or purpose.

exception
An abnormal situation in the execution of a program that typically results in an alteration of its normal flow. See also *condition*.

execute
To cause a program, utility, or other machine function to carry out the instructions contained within. See also *run*.

execution time
See *run time*.

execution-time environment
See *run-time environment*.

expanded date field
A COBOL date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year
In COBOL, four digits representing a year, including the century (for example, 1998). Appears in expanded date fields. Compare with *windowed year*.

expression
A group of constants or variables separated by operators that yields a single value. An expression can be arithmetic, relational, logical, or a character string.

eXtra Performance LINKage (XPLINK)
A new call linkage between functions that has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing nonessential instructions from the main path. When all functions are compiled with the XPLINK option, pointers can be used without restriction, which makes it easier to port new applications to z/OS.

F

file
A named set of records stored or processed as a unit. An element included in a container: for example, an MVS member or a partitioned data set. See also *data set*.

frequency count
A count of the number of times statements in the currently qualified program unit have been run.

full-screen mode
An interface mode for use with a nonprogrammable terminal that displays a variety of information about the program you are debugging.

H

high level language (HLL)
A programming language such as C, COBOL, or PL/I.

HLL
See *high level language*.

hook
An instruction inserted into a program by a compiler when you specify the TEST compile option. Using a hook, you can set breakpoints to instruct z/OS Debugger to gain control of the program at selected points during its execution.

I

inactive block
A block that is not currently executing, or is not in the CALL chain leading to the active block. See also *active block*, *block*.

index
A computer storage position or register, the contents of which identify a particular element in a table.

initial setting
A value in effect when the user's z/OS Debugger session begins. Contrast with *default*.

interactive
Pertaining to a program or system that alternately accepts input and then responds. An interactive system is conversational; that is, a continuous dialog exists between the user and the system. Contrast with *batch*.

I/O
Input/output.

L

Language Environment
An IBM software product that provides a common run-time environment and common run-time services for IBM high level language compilers.

library routine
A routine maintained in a program library.

line mode
An interface mode for use with a nonprogrammable terminal that uses a single command line to accept z/OS Debugger commands.

line wrap
The function that automatically moves the display of a character string (separated from the rest of a line by a blank) to a new line if it would otherwise overrun the right margin setting.

link-edit
To create a loadable computer program using a linkage editor.

linkage editor
A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single relocatable load module.

listing
A printout that lists the source language statements of a program with all preprocessor statements, includes, and macros expanded.

load module
A program in a form suitable for loading into main storage for execution. In this document this term is also used to refer to a Dynamic Load Library (DLL).

logical window
A group of related debugging information (for example, variables) that is formatted so that it can be displayed in a physical window.

M

minor node
In VTAM, a uniquely defined resource within a major node.

multitasking
A mode of operation that provides for concurrent performance, or interleaved execution of two or more tasks.

N

network identifier
In TCP/IP, that part of the IP address that defines a network. The length of the network ID depends on the type of network class (A, B, or C).

nonconversational
A transaction type that accepts input, performs a task, and then ends.

nondate
A COBOL data item that can be any of the following:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A reference modification of a date field
- The result of certain arithmetic operations that may include date field operands; for example, the difference between two compatible date fields.

The value of a nondate may or may not represent a date.

O

Options
A choice that lets the user customize objects or parts of objects in an application.

offset
The number of measuring units from an arbitrary starting point to some other point.

P

panel
In z/OS Debugger, an area of the screen used to display a specific type of information.

parameter
Data passed between programs or procedures.

partitioned data set (PDS)
A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

path point
A point in the program where control is about to be transferred to another location or a point in the program where control has just been given.

PDS
See *partitioned data set*.

physical window
A section of the screen dedicated to the display of one of the four logical windows: Monitor window, Source window, Log window, or Memory window.

prefix area
The eight columns to the left of the program source or listing containing line numbers. Statement breakpoints can be set in the prefix area.

primary entry point
See *entry point*.

procedure
In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. A set of related control statements. For example, an MVS CLIST.

process
The highest level of the Language Environment program management model. It is a collection of resources, both program code and data, and consists of at least one enclave.

Profile
A choice that allows the user to change some characteristics of the working environment, such as the pace of statement execution in the z/OS Debugger.

program
A sequence of instructions suitable for processing by a computer. Processing can include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it.

program unit
See *compile unit*.

program variable
A predefined variable that exists when z/OS Debugger was invoked.

pseudo-conversational transaction
The result of a technique in CICS called pseudo-conversational processing in which a series of nonconversational transactions gives the appearance (to the user) of a single conversational transaction. See *conversational* and *nonconversational*.

Q

qualification
A method used to specify to what procedure or load module a particular variable name, function name, label, or statement id belongs. The `SET QUALIFY` command changes the current implicit qualification.

R

record
A group of related data, words, or fields treated as a unit, such as one name, address, and telephone number.

record format
The definition of how data is structured in the records contained in a file. The definition includes record name, field names, and field descriptions, such as length and data type. The record formats used in a file are contained in the file description.

reference
In programming languages, a language construct designating a declared language object. A subset of an expression that resolves to an area of storage; that is, a possible target of an assignment statement. It can be any of the following: a variable, an array or array element, or a structure or structure element. Any of the above can be pointer-qualified where applicable.

run
To cause a program, utility, or other machine function to execute. An action that causes a program to begin execution and continue until a run-time exception occurs. If a run-time exception occurs, the user can use z/OS Debugger to analyze the problem. A choice the user can make to start or resume regular execution of a program.

run time
Any instant when a program is being executed.

run-time environment
A set of resources that are used to support the execution of a program.

run unit
A group of one or more object programs that are run together.

S

SBCS
See *single-byte character set*.

semantic error
An error in the implementation of a program's specifications. The semantics of a program refer to the meaning of a program. Unlike syntax errors, semantic errors (since they are deviations from a program's specifications) can be detected only at run time. Contrast with *syntax error*.

sequence number
A number that identifies the records within an MVS file.

session variable
A variable the user declares during the z/OS Debugger session by using Declarations.

single-byte character set (SBCS)
A character set in which each character is represented by a one-byte code.

Single Point of Control
The control interface that sends commands to one or more members of an IMSplex and receives command responses.

source
The HLL statements in a file that make up a program.

Source window
A z/OS Debugger window that contains a display of either the source code or the listing of the program being debugged.

SPOC
See Single Point of Control.

statement
An instruction in a program or procedure.

In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static
In programming languages, pertaining to properties that can be established before execution of a program; for example, the length of a fixed-length variable is static. Contrast with *dynamic*.

step
One statement in a computer routine. To cause a computer to execute one or more statements. A choice the user can make to execute one or more statements in the application being debugged.

storage
A unit into which recorded text can be entered, in which it can be retained, and from which it can be retrieved. The action of placing data into a storage device. A storage device.

subroutine
A sequenced set of instructions or statements that can be used in one or more computer programs at one or more points in a computer program.

suffix area
A variable-sized column to the right of the program source or listing statements, containing frequency counts for the first statement or verb on each line. z/OS Debugger optionally displays the suffix area in the Source window. See also *prefix area*.

syntactic analysis
An analysis of a program done by a compiler to determine the structure of the program and the construction of its source statements to determine whether it is valid for a given programming language. See also *syntax checker, syntax error*.

syntax
The rules governing the structure of a programming language and the construction of a statement in a programming language.

syntax error
Any deviation from the grammar (rules) of a given programming language appearing when a compiler performs a syntactic analysis of a source program. See also *syntactic analysis*.

T

session variable
See *session variable*.

thread
The basic line of execution within the Language Environment program model. It is dispatched with its own instruction counter and registers by the system. Threads can execute, concurrently with other threads. The thread is where actual code resides. It is synonymous with a CICS transaction or task. See also *enclave*.

thread id
A small positive number assigned by z/OS Debugger to a Language Environment task.

token
A character string in a specific format that has some defined significance in a programming language.

trigraph
A group of three characters which, taken together, are equivalent to a single special character. For example, ??) and ??(are equivalent to the left (<) and right (>) brackets.

U

utility
A computer program in general support of computer processes; for example, a diagnostic program, a trace program, or a sort program.

V

variable
A name used to represent a data item whose value can be changed while the program is running.

VTAM
See Virtual Telecommunications Access Method.

Virtual Telecommunications Access Method (VTAM)
IBM software that controls communication and the flow of data in an SNA network by providing the SNA application programming interfaces and SNA networking functions. An SNA network includes subarea networking, Advanced Peer-to-Peer Networking (APPN), and High-Performance Routing (HPR). Beginning with Release 5 of the OS/390 operating system, the VTAM for MVS/ESA function was included in Communications Server for OS/390; this function is called Communications Server for OS/390 - SNA Services.

An access method commonly used by MVS to communicate with terminals and other communications devices.

W

windowed date field
A COBOL date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year
In COBOL, two digits representing a year within a century window (for example, 98). Appears in windowed date fields. See also *century window (COBOL)*.

Compare with *expanded year*.

word wrap
See *line wrap*.

X

XPLINK
See eXtra Performance LINKage (XPLINK).

IBM z/OS Debugger publications

You can access the IBM z/OS Debugger publications by visiting any of the following pages:

- IBM Debug for z/OS:
  - IBM Documentation: https://www.ibm.com/docs/SSVSZX_15.0.0/com.ibm.debug.z.doc/topics/pdf.html
  - Library page: https://www.ibm.com/support/pages/node/713283
- IBM Developer for z/OS:
  - IBM Documentation: https://www.ibm.com/docs/SSQ2R2_15.0.0/com.ibm.debug.z.doc/topics/pdf.html
  - Library page: https://www.ibm.com/support/pages/node/713179
- IBM Z and Cloud Modernization Stack:
  - IBM Documentation: https://www.ibm.com/docs/SSV97FN_latest/com.ibm.debug.z.doc/topics/pdf.html

© Copyright IBM Corp. 1992, 2023

Index

Special Characters

__ctest() function 126
./E, BTS Environment command 97
.mdbg
    how z/OS Debugger locates 412
.mdbg file 403
.mdbg file, how to create 40, 45
&PGMNAME 97
&userid 517
&USERID 97
#pragma
    specifying TEST compiler option 43
    specifying TEST run-time option with 114
%CONDITION variable
    for PL/I 285
%PATHCODE variable
    for C and C++ 295
    for PL/I 284
    values for COBOL 271

A

ABEND 4038 384
abnormal end of application, setting breakpoint at 375
accessing PL/I program variables 287
ALL suboption of TEST compiler option (PL/I), effect of 39
ALL, how Enterprise COBOL for z/OS, Version 4, handles 34
ALLOCATE command
    managing file allocations 192
allocating z/OS Debugger
        files
    example of 134
allocating z/OS Debugger load library data
        set
    example of 134
ALTER PROCEDURE statement, example of 80
applications 357
Applid 85
assembler
    debugging a program in full-screen mode
        displaying variable or storage 248
        finding storage overwrite errors 250
        getting a function traceback 249
        modifying variables or storage 248
        multiple CUs in single assembly 246
        stopping at assembler routine call 248
        stopping when condition is true 249
    debugging non-reentrant 320
    defining CU as 245
    how z/OS Debugger locates EQALANGX files 411
    loading debug data of 245
    QUERY LOCATION 248
    reappearing 246
    restrictions
        assembler code using instructions as data 322
assembler *(continued)*
    restrictions *(continued)*
        detectable self-modifying 323
        non-detectable self-modifying 324
        non-Language Environment 322
        self-modifying 323
        SET DEFAULT VIEW NOMACGEN 325
        while debugging MAIN program 322
        with STORAGE run-time option 322
    sample program for debugging 243
    self-modifying code, restrictions 329
assembler program
    loading debug information 315
    locating EQALANGX 315
    making assembler CUs known to z/OS Debugger 315
assembler programs
    assembling, requirements 71
    requirements for debugging 71
    using z/OS Debugger Utilities to assemble and create 72
assembler, definition of xxi
assembling your program, requirements for 71
assigning values to variables 269, 295
AT commands
    AT CALL
        breakpoints, for C++ 311
    AT ENTRY
        breakpoints, for C++ 310
    AT EXIT
        breakpoints, for C++ 310
attention interrupt
    effect of during Dynamic Debug 195
    effect of during interactive sessions 194
    how to initiate 194
    required Language Environment run-time options 194
attributes of variables 377
automatic saving and restoring of settings, breakpoints, and monitor specifications 180
automatic saving and restoring of settings, breakpoints, and monitor specifications; disabling 181
available only with programs compiled with
    L prefix command 15
    M prefix command 16

B

base address, how to specify for MEMORY command 170
base address, using in Memory window 170
batch mode
    debugging Db2 programs in 333
    debugging IMS programs in 341
    description of 5
    for non-Language Environment programs 134
    starting z/OS Debugger in 129
    using z/OS Debugger in 493
binder APIs 393

blanks, significance of 264
BLOCK suboption of TEST compiler option (PL/I), effect of 38
BLOCK, how Enterprise COBOL for z/OS, Version 4, handles 34
blocks and block identifiers
  using, for C 304
boundaries, setting for searches 168
breakpoint
  clearing 17
  implicit 108
  setting, introduction to 14
  skipping 17
  using DISABLE and ENABLE 17
breakpoints
  before calling a NULL function
    in C 229
    in C++ 241
  before calling an invalid program, in COBOL 205
  before calling an undefined program, in PL/I 220
  halting if a condition is true
    in C 225
    in C++ 236
    in COBOL 201
    in LangX COBOL 210
    in PL/I 217
  halting when certain COBOL routines are called 200
  halting when certain functions are called
    in C 224
    in C++ 234
    in PL/I 216
  halting when certain LangX COBOL routines are called 210
  placing in IMS programs 347
  recording, using SET AUTOMONITOR 173
  setting a line 174
  setting, in C++ 310
breakpoints, setting in load modules that are not loaded 174
breakpoints, setting in programs that are not active 174
browse mode
  enabling and disabling 53
  introduction to 51
  list of commands not permitted 51, 52
  remote debug mode
    list of actions not permitted 52
BTS Environment command (./E), when to use 97

C

C
  compiling with c89 or c++ 62
  DEBUG compiler option, what it controls 40
  debugging a program in full-screen mode
    calling a C function from z/OS Debugger 226
    capturing output to stdout 226
    debugging a DLL 227
    displaying raw storage 227
    displaying strings 227
    finding storage overwrite errors 228
    finding uninitialized storage errors 229
    getting a function traceback 227
    halting on line if condition true 225
    halting when certain functions are called 224
    modifying value of variable 224
    setting breakpoint to halt 229
C *(continued)*
  debugging a program in full-screen mode *(continued)*
    tracing run-time path for code compiled with TEST 227
    when not all parts compiled with TEST 225
  GONUMBER compiler option 42
  LP64 versus ILP32 41
  OPT(1) or OPT(2) compiler options 42
  OPTIMIZE 41
  possible prerequisites 40–42
  preparing, programs to debug 39
  sample program for debugging 221
  TEST compiler option, what it controls 41
  user defined functions 41
  when to Dynamic Debug facility with 41, 42
C and C++
  AT ENTRY/EXIT breakpoints 310
  blocks and block identifiers 305
  choosing between TEST and DEBUG compiler option 39, 44
  commands
    summary 293
  equivalents for Language Environment conditions 300
  function calls for 298
  notes on using 261
  reserved keywords 299
  when to use FORMAT(DWARF) 39, 44
C/C++ file produced by DEBUG(FORMAT(DWARF)), how z/OS Debugger locates 412
C/C++ source files, how z/OS Debugger locates 412
C++
  AT CALL breakpoints 311
  DEBUG compiler option, what it controls 45
  debugging a program in full-screen mode
    calling a C++ function from z/OS Debugger 238
    capturing output to stdout 237
    debugging a DLL 238
    displaying raw storage 238
    displaying strings 238
    finding storage overwrite errors 240
    finding uninitialized storage errors 240
    getting a function traceback 238
    halting on a line if condition true 236
    modifying value of variable 235
    setting a breakpoint to halt 234, 241
    tracing the run-time path 239
    viewing and modifying data members 236
    when not all parts compiled with TEST 236
  examining objects 311
  GONUMBER compiler option 47
  LP64 versus ILP32 46
  OPT(1) or OPT(2) compiler options 47
  OPTIMIZE 46
  overloaded operator 310
  possible prerequisites 45
  preparing, programs to debug 44
  sample program for debugging 231
  setting breakpoints 310
  stepping through C++ programs 309
  template in C++ 310
  TEST compiler option, what it controls 46
  user defined functions 46
  using slashes to enter comments 264

C++ *(continued)*
when to Dynamic Debug facility with 46
CADP
how to start z/OS Debugger with 141
how to use 92
CAF (call access facility), using to start Db2 program 334
call access facility (CAF), using to start Db2 program 334
capturing output to stdout
in C 226
in C++ 237
CC...CC, Monitor prefix command 161
CCCA 57
CEE3CBTS 391
CEEBXITA
description of how it works 97
CEEBXITA, comparing two methods of linking 100
CEEBXITA, specifying message display level in 99
CEEBXITA, specifying naming pattern in 98
CEEROPT, using
for IMS programs 95
CEETEST
description 119
examples, for C 122
examples, for COBOL 123
examples, for PL/I 124
Starting z/OS Debugger with 119
using 341
CEEUOPT runtime options module 76
CEEUOPT to start z/OS Debugger under CICS, using 141
CEEUOPT, using
for IMS programs 95
changing how Monitor window displays values 183
changing physical window layout in the session panel 252
changing the value of a variable, introduction to 17
character set 261
characters, searching 167
CICS
breakpoints, pattern-match 351
CADP, how to use 92
choosing a debugging mode for 50
DPL 50
DTCN profile, creating a 82
DTCN profiles, displaying list of 85
DTCN, fields on Advanced Options 91
DTCN, fields on Menu 2 90
DTCN, fields on Primary Menu 86
DTST transaction, description of storage window 500
DTST transaction, navigating through DTST storage window 499
DTST transaction, starting the 497
DTST transaction, syntax of the 501
DTST transaction, using to modify storage 499
list of general tasks to complete for 81
non-Language Environment programs, passing runtime parameters to 92
non-Language Environment programs, starting z/OS Debugger for 92
pseudo-conversational program 353
region, reloading programs into an active 513
requirements for using z/OS Debugger in 349
restoring breakpoints 353
restrictions for debugging 354
CICS *(continued)*
saving breakpoints 353
starting the log file 354
starting z/OS Debugger under 139
WAIT option 50
CICS debugging
RLIM processing 354
closing automonitor section of Monitor window 186
closing z/OS Debugger physical windows 252
COBOL
CCCA 57
command format 267
debugging a program in full-screen mode
capturing I/O to system console 202
displaying raw storage 203
finding storage overwrite errors 205
generating a run-time paragraph trace 204
modifying the value of a variable 200
setting a breakpoint to halt 200
setting breakpoint to halt 205
stopping on line if condition true 201
tracing the run-time path 203
when not all parts compiled with TEST 202, 211
debugging COBOL classes 277
debugging VS COBOL II programs
finding listing 278
EJPD suboption 27
Enterprise, L prefix command only available with 15
Enterprise, M prefix command only available with 16
FACTORY 277
how z/OS Debugger locates separate debug file 411
how z/OS Debugger locates source file 410
list of effect of ALL compiler option 34
list of effect of BLOCK compiler option 34
list of effect of NOSYM compiler option 33
list of effect of NOTEST compiler option 31
list of effect of PATH compiler option 33
list of effect of STMT compiler option 33
Load Module Analyzer 56
non-Language Environment, QUERY LOCATION 210
NONE and NOHOOK with optimized programs 32
note on using H constant 265
notes on using 262
OBJECT 277
OPT compiler option 27, 28
optimized programs, debugging 362
paragraph names, finding 169
paragraph trace, generating a COBOL run-time 204
possible prerequisites 30
QUERY LOCATION 200
reserved keywords 268
RESIDENT compiler option 30
restrictions on accessing, data 178
run-time options 113
sample program for debugging 197, 207
SOURCE compiler option 30
TEST compiler option, what suboptions to specify 27
variables, using with z/OS Debugger 268
when to Dynamic Debug facility with 27
why you need to specify SYM 30
Working-Storage Section, displaying 184
COBOL listing, data set 401
COBOL, reusable runtime environments 367
coexistence of z/OS Debugger with other debuggers 379

coexistence with unsupported HLL modules 379
colors
changing in session panel 253
columnar format, displaying value in Monitor window in 189
command
syntax diagrams xxi
command format
for COBOL 267
command line, z/OS Debugger 159
Command pop-up window, changing size of 155
command sequencing, full-screen mode 160
commands
abbreviating 262
DTSU, using to debug Db2 program 333
for C and C++, z/OS Debugger subset 293
for PL/I, z/OS Debugger subset 283
getting online help for 265
interpretive subset
description 372
multiline 263
PLAYBACK 18
prefix, using in z/OS Debugger 161
truncating 262
TSO, using to debug Db2 program 334
commands (system), entering in z/OS Debugger 160
commands file
example of specifying 129
using log file as 172
using session log as 109
Commands File
in DTCN, description of 90
commands file, how to create a 171
commands, z/OS Debugger
COBOL compiler options in effect 268
entering on the session panel 158
entering using program function keys 162
order of processing 160
retrieving with RETRIEVE command 163
that resemble COBOL statements 267
COMMANDSDSN, EQAOPTS command 171, 404
Commarea data 91
Commarea offset 91
comments, inserting into command stream 264
Common pop-up window, how to enter commands in 164
compile unit
general description 373
name area, z/OS Debugger 159
qualification of, for C and C++ 307
compile units known to z/OS Debugger, displaying list of 194
compiler options
COBOL 27
how to choose, for PL/I 35
suggested 26, 27
which options to use for COBOL 27
compiling
a C program on an HFS or zFS file system 63
a C++ program on an HFS or zFS file system 63
an OS/VS COBOL program 56
Enterprise PL/I program on HFS or zFS file system 62
programs, introduction to 11
condition
handling of 285, 375
Language Environment, C and C++ equivalents 300
considerations
when using the TEST run-time option 107
constants
entering 264
HLL 372
PL/I 289
using in expressions, for COBOL 274
z/OS Debugger interpretation of HLL 371
constructor, stepping through 309
Container data 91
Container name 91
Container offset 91
continuation character
for COBOL 267
using in full-screen 263
continuing lines 263
continuous display 184
copying
JCL into a setup file using DTSU 116
CREATE PROCEDURE statement, example of 80
creating
setup file using z/OS Debugger Utilities 115
CRTE 50
CSECT, debugging multiple, in one assembly 247
CSECT, loading multiple, in one assembly 247
CU(s) 85
CURSOR command
using 164, 165
cursor commands
CLOSE 253
CURSOR 165
FIND 167
OPEN 253
SCROLL 151, 165
SIZE 253
using in z/OS Debugger 162
WINDOW ZOOM 253
customer support 536
customizing
PF keys 251
Profile panel 107
profile settings 254
session settings 251
CWI, Language Environment 391

D

data only modules, debugging 394
DATA parameter
restrictions on accessing COBOL data 178
data sets
COBOL listing 401
PL/I listing 402
PL/I source 402
separate debug file 402
specifying 172
used by z/OS Debugger 401
data type of variable, displaying in Monitor window the 185
Db2
assembling with assembler programs 76

Db2 *(continued)*
compiling with C or C++ programs 76
compiling with COBOL programs 75
compiling with PL/I programs 75
Db2 programs for debugging 75
linking programs 76
using z/OS Debugger with 333
Db2 programs
what files to keep 75
Db2 programs, binding 77
Db2 stored procedures
compiling or assembling options to use 79
debugging modes supported 79
restrictions 119
specifying TEST runtime options through EQAD3CXT 80
starting z/OS Debugger from 145
using z/OS Debugger with 335
what to do before debugging 79
DBCS
using with C 261
using with COBOL 271
using with z/OS Debugger commands 261
DEBUG and TEST compiler option, choosing between 39, 44
DEBUG compiler options 40, 45
debug mode
delay 397, 399
debug session
ending 195
recording 153
starting 143
starting your program 143
debuggers, coexistence with other 379
debugging
CICS programs 349
CICS programs, choosing mode 50
COBOL classes 277
Db2 programs 333
Db2 stored procedures 335
DLL
in C 227
in C++ 238
IMS programs, choosing mode 51
in full-screen mode 149
ISPF applications 357
multithreading programs 381
non-Language Environment programs 367
UNIX System Services programs 365
debugging profiles
how to create one with DTCN 82
declared data type, displaying characters in their 189
declared data type, modifying characters that cannot be displayed in their 189
declaring session variables
for C 296
for COBOL 272
deferred, description of 246
deferring an LDD command 209
DESCRIBE ALLOCATIONS command
managing file allocations 192
DESCRIBE command
using 307
description of how z/OS Debugger locates CICS tasks to debug 140
destructor, stepping through 309
diagnostics, expression, for C and C++ 300
DISABLE command 351
disassembly
changing program in disassembly view 330
differences between SET ASSEMBLER and SET DISASSEMBLY 315, 327
displaying registers 330
displaying storage 330
modifying registers 330
modifying storage 330
performing single-step operations 329
restrictions on what you can debug 330
self-modifying code, restrictions 329
setting breakpoints 329
what you can do is disassembly view 327
disassembly view, description of 328
disassembly view, how to start 328
Display Id
in DTCN, description of 89
displaying
environment information 307
halted location 169
lines at top of window, z/OS Debugger 166
raw storage
in C 227
in C++ 238
in COBOL 203
in PL/I 218
source or listing file in full-screen mode 156
strings
in C 227
in C++ 238
value of variable one time 183
values of COBOL variables 270
variable value 182
variables or storage
in LangX COBOL 210
displaying list of known compile units 194
displaying prefixes 394
displaying the value of a variable, introduction to 15
displaying variable value 182
displaying Working-Storage Section 184
DLL debugging
in C 227
in C++ 238
documents, licensed xvii
DOWN, SCROLL command 165
DTCN
creating a profile 82
data entry verification 84
defining COMMAREA 84
description of 139
description of columns 85
description of Session Type 89
do not link to EQADCCXT with particular COBOL compilers 81
do not link to EQADCCXT with particular PL/I compilers 81
migrating from versions earlier than V10 87
modifying Language Environment options 91
using repository profile items 140
DTCN Profiles 515, 518

DTCNFORCEFORCEIP, how Transaction Id in DTCN works with 89
DTCNFORCELOADMODID, how Transaction Id in DTCN works with 88
DTCNFORCENETNAME, how Transaction Id in DTCN works with 88
DTCNFORCETERMID, how Terminal Id in DTCN works with 86
DTCNFORCETRANID, how Transaction Id in DTCN works with 86
DTCNFORCEUSERID, how Transaction Id in DTCN works with 88
DTNP 513
DTSC 50
DTSP Profile 515, 518
DTST
  syntax of 501
DTST transaction
  description of storage window 500
  modifying storage after starting 499
  navigating through storage window 499
  starting the 497
  syntax of the 501
DWARF suboption of FORMAT compiler option, when to use 39, 44
Dynamic Debug
  attention interrupts, support for 195
Dynamic Debug facility, how it works 48

E

editing
  setup file using z/OS Debugger Setup Utility 115
elements, unsupported, for PL/I 291
ENABLE command 352
enclave
  multiple, debugging interlanguage communication application in 389
  non-Language Environment 119
  starting 383
ending
  debug session 195
  z/OS Debugger within multiple enclaves 383
English, specifying 10
English, specifying uppercase 10
entering
  commands on session panel 158
  file allocation statements into setup file 116
  program parameters into setup file 116
  runtime option into setup file 116
entering long command with Command pop-up window 164
entering multiline commands without continuation 263
entering PL/I statements, freeform 286
Enterprise COBOL
  compiler options to use 68
Enterprise PL/I
  restrictions 291
Enterprise PL/I, definition of xxi
ENU 10
EQAD3CXT
  comparing Db2 RUNOPTS to 97
EQADCCXT 81
EQADCCXT user exit 109
EQADEBUG DD statement 157
EQALANGX
  creating for LangX COBOL 68
EQALANGX file
  how to create 72
EQALANGX files, how z/OS Debugger locates 407, 408, 411
EQALMPFX 508
EQALMPGM 510
EQALOAD 393
EQANMDBG
  example 137
  methods for starting z/OS Debugger with 135
  passing parameters to
    using only EQANMDBG DD statement 136
    using only PARM 136
EQAOPTS file, format options 404
EQAOPTS file, where to specify, in DTCN 91
EQASET
  when to run 97
EQASTART, entering command 9
EQASYSPF 508
EQAUEDAT user exit 157
EQAUOPT
  how to create with IBM z/OS Debugger Utilities 103
  how to create with TIM 102
EQAWLMA 505
EQUATE, SET command
  description 251
error numbers in Log window 193
evaluating expressions
  COBOL 273
  HLL 371
evaluation of expressions
  C and C++ 300
examining C++ objects 311
examples
  assembler
    sample program for debugging 243
  C
    sample program for debugging 221
  C and C++
    assigning values to variables 295
    blocks and block identifiers 306
    expression evaluation 297
    monitoring and modifying registers and storage 312
    referencing variables and setting breakpoints 305
    scope and visibility of objects 306
  C++
    displaying attributes 311
    sample program for debugging 231
    setting breakpoints 311
  CEETEST calls, for PL/I 124
  CEETEST function calls, for C 122
  CEETEST function calls, for COBOL 123
  changing point of view, general 374
  COBOL
    %HEX function 274
    %STORAGE function 275
    assigning values to COBOL variables 269
    changing point of view 276
    displaying results of expression evaluation 273
    displaying values of COBOL variables 270

examples *(continued)*
COBOL *(continued)*
qualifying variables 275
sample program for debugging 197
using constants in expressions 274
code coverage 111
declaring variables, for COBOL 272
displaying program variables 294
full-screen mode 112
modifying setup files by using IBM z/OS Debugger Utilities 415
OS/VS COBOL
sample program for debugging 207
PL/I
in PL/I 216
sample program for debugging 213
PLITEST calls for PL/I 125
preparing programs by using IBM z/OS Debugger Utilities 415
remote debug mode 110
specifying TEST run-time option with #pragma 114
TEST run-time option 112
using #pragma for TEST compiler option 43
using constants 265
using continuation characters 263
using qualification 307
exception handling for C and C++ and PL/I 376
excluding programs 396
EXEC CICS RETURN
under CICS 352
explicit debug mode 395
expressions
diagnostics, for C and C++ 300
displaying values, for C and C++ 294
displaying values, for COBOL 273
evaluation for C and C++ 296, 300
evaluation for COBOL 273
evaluation of HLL 371
evaluation, operators and operands for C 299
for PL/I 289
using constants in, for COBOL 274

F

feedback codes, when to use 121
FIND command
using with windows 167
FIND command, setting boundaries with 168
finding
characters or strings 167
storage overwrite errors
in assembler 250
in C 228
in C++ 240
in COBOL 205
in LangX COBOL 211
in PL/I 219
uninitialized storage errors
in C 229
in C++ 240
finding COBOL paragraph names, example of 169
fixes, getting 535
FREE command
managing file allocations 192
freeform input, PL/I statements 286
full-screen mode
CICS, additional terminals 50
continuation character, using in 263
CURSOR 162
CURSOR command 165
debugging in 149
description of 5
example screen 12
examples of 112
introduction to 11
PANEL COLORS 253
PANEL LAYOUT 252
PANEL PROFILE 254
SCROLL 165
which why type of programs to use 49
WINDOW CLOSE 253
WINDOW OPEN 253
WINDOW SIZE 253
WINDOW ZOOM 253
full-screen mode using the Terminal Interface Manager
description of 6
starting a debugging session 131
function calls, for C and C++ 298
function, calling C and C++ from z/OS Debugger
C 226
C++ 238
function, unsupported for PL/I 291
functions
PL/I 289
functions, z/OS Debugger
%HEX
using with COBOL 274
%STORAGE
using with COBOL 275
using with COBOL 274

G

global data 312
global preferences file 404
global scope operator 312
GPFDSN, EQAOPTS command 404

H

H constant (COBOL) 264
halted location, displaying 169
header fields, z/OS Debugger session panel 150
help, online
for command syntax 265
hexadecimal format, displaying values in 190
hexadecimal format, how to display value of variable 190
hexadecimal format, how to monitor value of variable 190
hexadecimal format, monitoring values in 190
HFS or zFS, compiling a C program on 63
HFS or zFS, compiling a C++ program on 63
HFS or zFS, compiling Enterprise PL/I program on 62
highlighting, changing in z/OS Debugger session panel 253
history area of Memory window 170
history, z/OS Debugger command
retrieving previous commands 163

hooks
compiling with 48
compiling with, PL/I 34
compiling without, COBOL 48
removing from application 359, 360
rules for placing in C programs 44
rules for placing in C++ programs 47, 48
how to choose 40, 45

I

I/O, COBOL
capturing to system console 202
IBM Support Assistant, searching for problem resolution 535
IBM z/OS Debugger Utilities
brief description of Load Module Analyzer 8
brief description on preparing assembler 7
creating and managing setup files 7
creating setup file for IMS program 345, 495
Delay Debug Profile 8
how to start 9
IMS Transaction and User ID Cross Reference Table 9
instructions for compiling or assembling 416
list of all utilities in 7
managing debugging profiles 8
Non-CICS Debug Session Start and Stop Message Viewer 9
overview of IMS BTS Debugging 8
overview of IMS program preparation tasks 8
overview of JCL file conversion 8
overview of JCL for Batch Debugging 8
overview of Job Cards 7
overview of program preparation tasks 7
IBM z/OS Debugger Utilities, general instructions on how to use 61
ignoring programs 395
improving performance in multi-enclave environments 182
improving z/OS Debugger performance 359
IMS
choosing a debugging mode for 51
choosing method to specify TEST runtime options 95
JCL, sample doing replace link edit of CEEBXITA into CEEBINIT 96
making a user exit application-specific 96
making a user exit available installation-wide 96
making a user exit available region-wide 96
programs, debugging interactively 340
programs, non-Language Environment 340
transaction isolation 337, 339
IMS MPP
debugging 341
preparing to debug 341
INCLUDE files, how to automonitor variables in, while in remote debug mode 35
INCLUDE files, how to debug PL/I 35
information, displaying environmental 307
initial programs, non-Language Environment
CICS assembler 367
non-Language Environment COBOL 367
input areas, order of processing, z/OS Debugger 160
INSPLOG
creating the log file 172
example of using 134
INSPPREF
INSPPREF *(continued)*
example of using 134
INSPSAFE
example of using 134
instructions on how to compile a program with IBM z/OS Debugger Utilities 61
interfaces
batch mode 5
full-screen mode 5
full-screen mode using the Terminal Interface Manager 6
remote debug mode 6
interfaces, description of 5
interLanguage communication (ILC) application, debugging 389
interlanguage programs, using with z/OS Debugger 376
Internet
searching for problem resolution 535
interpretive subset
general description 372
of C and C++ commands 293
of COBOL statements 267
of PL/I commands 283
INTERRUPT, Language Environment run-time option 194
IP Name/Addr 85
IP Name/Address
in DTCN, description of 89
IPv6 format (TCP/IP) 343
ISPF
starting 161

J

Japanese, specifying 10
Java 391
JCL sample, linking CEEBXITA into your program 100
JCL sample, runs z/OS Debugger in batch mode 129
JCL to create EQALANGX file 72
JCL, list of changes to make to 59
JNI 391
JPN 10

K

keywords, abbreviating 262
KOR 10
Korean, specifying 10

L

Language Environment
conditions, C and C++ equivalents 300
EQADCCXT user exit 109
runtime options, precedence 109
user exit, link, into private copy of Language Environment runtime module 101
user exit, link, into your program 100
user exits, how to prepare 98
user exits, methods to modify sample assembler 98
Language Environment user exit, create and manage data set used by 101
language, specifying national 10
LangX COBOL

LangX COBOL *(continued)*
%PATHCODE values 281
debugging a program in full-screen mode
displaying raw storage 210
finding storage overwrite errors 211
setting a breakpoint to halt 210
stopping on line if condition true 210
when not all parts compiled with TEST 211
how to prepare a 67
loading debug information for 279
session panel's appearance 279
LDD command, example 315
LEFT, SCROLL command 165
licensed documents xvii
line breakpoint, setting 174
line continuation
for C 263
for COBOL 263
link-edit assembler program
how to, by using z/OS Debugger Utilities 73
linking
Db2 programs 76
EQADCCXT 81
LIST %HEX command 190
LIST command
use to display value of variable one time 183
LIST commands
LIST STORAGE
using with PL/I 286
List pop-up window, description of 155
listing
find, OS PL/I 291
find, VS COBOL II 278
listing files, how z/OS Debugger locates 407, 408, 410
literal constants, entering 264
LLA 393
Load Module Analyzer 56, 505
LoadMod::>CU(s)
in DTCN, description of 87
LoadMod(s) 85
LOCATION, description of 151
log file
creating 172
using 172
using as a commands file 172
log file, saving automonitor section to 187
Log window
description 153
error numbers in 193
retrieving lines from 164
log, session 109
LOGDSN, EQAOPTS command 172, 405
LOGDSNALLOC, EQAOPTS command 172
low-level debugging 312
LRECL 401

M

MAIN Db2 stored procedures 79
managing file allocations 192
manual restoring of settings, breakpoints, and monitor specifications 181
mdbg
how z/OS Debugger locates 412
mdbg file 403
MDBG, EQAOPTS command 40, 41, 45, 46
memory
displaying, introduction to 16
MEMORY command, using 192
Memory window
description of 154
displaying with base address 192
history area, navigating with 170
opening an empty 169
Memory window, addresses that span two columns 170
Memory window, entering multiple commands in 162
message display level, how to specify, in Language Environment user exit 99
modifying
value of variable by typing over 191
value of variable by using command 190
modifying value of a C variable 224
MONITOR command
viewing output from, z/OS Debugger 152
MONITOR LIST command, using to monitor variables 184
MONITOR LIST TITLED WSS 184
Monitor window
description 152
opening and closing 192, 253
Monitor window, adding variables to 186
Monitor window, replacing variables in 185
monitoring 184
monitoring storage in C++ 312
more than one language, debugging programs with 376
moving around windows in z/OS Debugger 164
moving the cursor, z/OS Debugger 165
moving to new level of Language Environment 101
multilanguage programs, using with z/OS Debugger 376
multiline commands
continuation character, using in 263
without continuation character 263
multiple commands, entering in Memory window 162
multiple enclaves
ending z/OS Debugger 383
interlanguage communication application, debugging 389
starting 383
multithreading
restrictions 381
MVS
starting z/OS Debugger using TEST run-time option 143
MVS POSIX programs, debugging 365
MVS, starting z/OS Debugger under 133

N

name (default) of data set that saves settings, breakpoints, and monitors specifications 179
NAMES 393
NAMES command
using EQAOPTS 396
NAMES EXCLUDE 396
naming conflicts 393

naming pattern, how to specify, in Language Environment user exit 98
national language, specifying 10
NATLANG parameter 10
navigating session panel windows 164
Netname 85
NetName
  in DTCN, description of 88
NOHOOK suboption of TEST compiler option (PL/I), effect of 37
NOMACGEN 319
non-Language Environment
  CICS
    passing runtime parameters 92
    Starting z/OS Debugger 92
  defining as 209
  how z/OS Debugger locates EQALANGX files 411
  loading debug information 209
  restrictions 281
non-Language Environment initial programs
  CICS assembler 367
  non-Language Environment COBOL 367
non-Language Environment programs
  debugging 367
  starting z/OS Debugger 134
non-reentrant
  breakpoints 320
  debugging, assembler 320
  variables 320
NONE suboption of TEST compiler option (PL/I), effect of 37
NOSYM suboption of TEST compiler option (C), effect of 43
NOSYM suboption of TEST compiler option (PL/I), effect of 38
NOTEST compiler option (C), effect of 42
NOTEST compiler option (C++), effect of 47
NOTEST compiler option (PL/I), effect of 37
NOTEST suboption of TEST run-time option 108

O

objects
  C and C++, scope of 303
opening Memory window with base address 192
opening z/OS Debugger physical windows 252
operators and operands for C 299
OPT
  C compiler option 42
  C++ compiler option 47
  COBOL compiler option 27, 28
OPTIMIZE, C compiler option 41
OPTIMIZE, C++ compiler option 46
optimized applications, debugging large 394
optimized COBOL programs, modifying variables in 191, 269, 270
optimized programs, compiling COBOL with NONE and NOHOOK 32
optimized programs, debugging COBOL 362
options module, CEEUOPT runtime 76
OS PL/I programs, debugging 291
OS PL/I, compiling 36
OS PL/I, finding list for 291
OS/VS COBOL
  compiler options to use 67
OS/VS COBOL *(continued)*
  restrictions 280
output
  C, capturing to stdout 226
  C++, capturing to stdout 237
overloaded operator 310
overwrite errors, finding storage
  in assembler 250
  in C 228
  in C++ 240
  in COBOL 205
  in LangX COBOL 211
  in PL/I 219

P

panel
  header fields, session 150
  Profile 254
PANEL command (full-screen mode)
  changing session panel colors and highlighting 253
PANEL PROFILE command 157
paragraph trace, generating a COBOL run-time 204
PATH, how Enterprise COBOL for z/OS, Version 4, handles 33
performance
  enhancing z/OS Debugger 76
performance, improving z/OS Debugger 359
PF keys
  defining 251
  using 162
PF4 key, using 183
PHASEIN 513
physical
  opening and closing windows 252
physical window, enlarging 166
PL/I
  AFTERALL 35
  AFTERCICS 35
  AFTERMACRO 35
  AFTERSQL 35
  built-in functions 289
  compiler options to use to automonitor variables in INCLUDE files while in remote debug mode 35
  compiler options to use when you want to debug INCLUDE files 35
  condition handling 285
  constants 289
  debugging a program in full-screen mode
    displaying raw storage 218
    finding storage overwrite errors 219
    getting a function traceback 218
    halting on line if condition is true 217
    modifying value of variable 216
    setting a breakpoint to halt 216
    setting breakpoint to halt 220
    tracing run-time path for code compiled with TEST 218
    when not all parts compiled with TEST 217
  debugging OS PL/I programs
    finding listing 291
  Enterprise, L prefix command only available with 15
  Enterprise, M prefix command only available with 16
  Enterprise, restrictions 291

PL/I *(continued)*
expressions 289
how to choose compiler options for 35
how z/OS Debugger locates separate debug file 411
notes on using 262
PLIBASE 36
possible prerequisites 36
preparing a program for debugging 34
QUERY LOCATION 216
run-time options 113
sample program for debugging 213
session variables 286
SIBMBASE 36
statements 283
structures, accessing 287
TEST compiler option, what it controls 34
when to Dynamic Debug facility with 36
PL/I for MVS & VM, compiling 36
PL/I listing, data set 402
PL/I source, data set 402
PL/I, definition of xxi
PLAYBACK commands
introduction to 18
PLAYBACK BACKWARD
using 178
PLAYBACK DISABLE
using 178
PLAYBACK ENABLE
using 176
PLAYBACK FORWARD
using 178
PLAYBACK START
using 177
PLAYBACK STOP
using 178
PLIBASE 36
PLITEST 125
plug-ins
how to install 515
plug-ins for remote debugger 515, 518
point of view, changing
description 374
for C and C++ 308
with COBOL 276
POPUP command 155
positioning lines at top of windows 166
precompiling Db2 programs 75
preference file 91, 107
preferences file
customizing z/OS Debugger with 256
Preferences File
in DTCN, description of 91
preferences files, how to create a 156
prefix area
z/OS Debugger 159
Prefix area, description of 152
prefix commands
prefix area on session panel 159
using in z/OS Debugger 161
prepare an assembler program, steps to 71
preparing
a PL/I program for debugging 34
preparing *(continued)*
C programs for debugging 39
C++ programs for debugging 44
to replay recorded statements using PLAYBACK START command 177
prerequisites
for COBOL, possible 30
previous commands, retrieving 163
problem determination
describing problems 536
determining business impact 536
submitting problems 537
Profile name pattern 517
profile settings, changing in z/OS Debugger 254
program
CICS, choosing debugging mode for 50
CICS, debugging 349
Db2, debugging 333
hook
compiling with, PL/I 34
removing 359, 360
rules for placing in C 44, 48
rules for placing in C++ 47
IMS, choosing debugging mode for 51
loaded from LLA 393
multithreading, debugging 381
preparation
considerations, size and performance 359, 360
TEST compiler option, for PL/I 34
TEST compiler option, for VS COBOL II 30
reducing size 359
source, displaying with z/OS Debugger 152
stepping through 175
that z/OS Debugger ignores when explicit debug mode is active 395
UNIX System Services, debugging 365
variables
accessing for C and C++ 294
variables, accessing for COBOL 269
Program IDs, specifying correct for C/C++ and Enterprise PL/I programs 87
programming language neutral, how to write commands that are 171
pseudo-conversational program, saving settings 353
PX constant (PL/I) 264

Q

qualification
description, for C and C++ 307
general description 373
qualifying variables
with COBOL 275
QUERY LOCATION
assembler 248
COBOL 200
LangX COBOL 210
PL/I 216

R

RACF access, combinations of EQAOPTS BROWSE command and 53

RECFM 401
recording
  breakpoints using SET AUTOMONITOR 173
  number of times each source line runs 173
  restrictions on, statements 178
  session with the log file 172
  statements, introduction to 18
  statements, using PLAYBACK ENABLE command 176
  stopping, using PLAYBACK DISABLE command 178
recording a debug session 153
referencing variables, implications of 49
reloading programs into an active CICS region 513
remote debug mode
  commands not allowed while browse mode is active 53
  description of 6
  examples of 110
  plug-ins for 515, 518
  where to find list of z/OS Debugger commands supported by 6
remote debug mode, PL/I, debugging INCLUDE files 35
removing statement and symbol tables 360
replacing variables in Monitor window 185
replaying
  statements, introduction to 18
replaying recorded statements 177
replaying statements
  changing direction of 178
  direction of 177
  restrictions on 178
  stopping using PLAYBACK STOP command 178
  using PLAYBACK commands 176
  using PLAYBACK START command 177
requirements
  for debugging CICS programs 349
reserved keywords
  for C 299
  for COBOL 268
RESLIB 30
restoring, manually; of settings, breakpoints, and monitor specifications 181
restrictions
  accessing COBOL data, for 178
  arithmetic expressions, for COBOL 273
  debugging OS PL/I programs 291
  debugging VS COBOL II programs 277
  expression evaluation, for COBOL 273
  location of source on HFS or zFS 62, 63
  modifying variables in Monitor window 191
  recording and replaying statements, for 178
  string constants in COBOL 274
  when debugging multilanguage applications 381
  when debugging under CICS 354
  when using a continuation character 268
  while debugging assembler programs 320
  while debugging Enterprise PL/I 291
RETRIEVE command
  using 163
retrieving commands
  with RETRIEVE command 163
retrieving lines from Log or Source windows 164
RIGHT, SCROLL command 165
RLIM processing, CICS 354
RUN subcommand 334
run time
run time *(continued)*
  environment, displaying attributes of 307
  option, TEST(ERROR, ...), for PL/I 286
  options module, CEEUOPT 76
run-time options
  specifying the STORAGE option 113
  specifying the TRAP(ON) option 113
  specifying with COBOL and PL/I 113
running a program 175
running in batch mode
  considerations, TEST run-time option 108
running your program, introduction to 14
RUNOPTS (Db2)
  comparing EQAD3CXT to 97
RUNTO command
  using, to replay recorded statements 177

S

save breakpoints file 405
save monitor specifications file 405
save settings file 405
SAVEBPDNSALLOC, EQAOPTS command 406
SAVEBPDSN, EQAOPTS command 405
SAVEBPS 405
SAVESETDSN, EQAOPTS command 405
SAVESETDSNALLOC, EQAOPTS command 405
SAVESETS 405
saving
  breakpoints 178
  monitor specifications 178
  settings 178
  setup file using z/OS Debugger Utilities 117
saving (automatically) settings, breakpoints, and monitor specifications 180
saving and restoring customizations 257
saving and restoring settings, how to improve performance in environment with multiple enclaves 182
saving, disabling automatic of settings, breakpoints, and monitor specifications 181
scenarios
  list of C, debugging 40, 41, 46
  list of C++, debugging 45
  list of COBOL, debugging 27
  list of PL/I, debugging 35
scope of objects in C and C++ 303
screen control mode, what is 50
scroll area, z/OS Debugger 159
SCROLL command
  using 164
search string, syntax of 167
searching for characters or strings 167
searching, how z/OS Debugger searches for 167
SELECT statement, example of 80
self-modifying code, restrictions for debugging 329
separate debug file
  COBOL and PL/I, how z/OS Debugger locates the 411
separate debug file files, how z/OS Debugger locates 407, 408
separate debug file, attributes to use for 76
separate debug file, data set 402
separate terminal mode, what is 50

service, when you apply to Language Environment 101
session
  variables, for PL/I 286
session panel
  changing colors and highlighting in 253
  changing physical window layout 252
  command line 159
  description 149
  header fields 150
  navigating 164
  order in which z/OS Debugger accepts commands from 160
  PF keys
    initial settings 163
    using 162
  while debugging LangX COBOL 279
  windows
    scrolling 165
session panel, while debugging assembler 316
session settings
  changing in z/OS Debugger 251
session variables
  declaring, for COBOL 272
SET AUTOMONITOR ON BOTH command, how it works 188
SET AUTOMONITOR ON command, example 188
SET AUTOMONITOR ON command, how it works 187
SET AUTOMONITOR ON PREVIOUS command, how it works 187
SET commands
  SET AUTOMONITOR
    using to record breakpoints 173
    viewing output from 152
  SET AUTOMONITOR ON
    monitoring values of variables 186
  SET DEFAULT SCROLL
    using 151
  SET EQUATE
    using 251
  SET INTERCEPT
    using with C and C++ programs 301
  SET PFKEY
    using in z/OS Debugger 162
  SET QUALIFY
    using with COBOL 276
    using, for C and C++ 308
  SET REFRESH
    using 357
  SET SCROLL DISPLAY OFF
    using 151
  SET WARNING
    using with PL/I 290
SET DEFAULT LISTINGS command 157
SET EXPLICITDEBUG 395
SET QUALIFY
  with multiple enclaves 383
SET SOURCE command 157
set up
  overall steps to, debugging session 25
SET WARNING OFF, how to use 174
setting
  line breakpoint 174
setting breakpoints, in C++ 310
setting breakpoints, introduction to 14
settings
  changing z/OS Debugger profile 254
  changing z/OS Debugger session 251
setup file
  copying JCL into, using DTSU 116
  creating, using z/OS Debugger Utilities 115
  editing, using DTSU 115
  saving, using z/OS Debugger Utilities 117
setup files
  overview of 7
SIBMBASE 36
single terminal mode, what is 50
size, reducing program 359
sizing physical windows 253
skipping programs 395
Software Support
  contacting 536
  describing problems 536
  determining business impact 536
  receiving updates 535
  submitting problems 537
Source display area, description of 152
source file in window, changing 157
source files, how z/OS Debugger locates 407, 408, 410
Source window
  changing source files 157
  description 152
  displaying halted location 169
  retrieving lines from 164
SOURCE, PL/I compiler option 36
source, program
  displaying with z/OS Debugger 152
SQLCODE 335
Sta 85
STANDARD 319
starting
  a debugging session in full-screen mode using the Terminal Interface Manager 131
  IBM z/OS Debugger Utilities 9
  your program from z/OS Debugger Utilities 117
  z/OS Debugger from Db2 stored procedures 145
  z/OS Debugger in full-screen mode, introduction to 12
starting a debug session 143
starting interactive function calls
  in C 226
starting your program 143
starting z/OS Debugger
  __ctest(), using 126
  batch mode 129
  Db2 program with TSO 334
  from a Language Environment program 119
  under CICS 139, 142
  under CICS, using CEEUOPT 141
  under MVS in TSO 133
  using the TEST run-time option 107
  with PLITEST 125
  with the CEETEST function call 119
  within an enclave 383
Starting z/OS Debugger
  at different points 108

statement tables, removing 360
statements
PL/I 283, 286
recording and replaying, introduction to 18
stdout, capturing output to
in C 226
in C++ 237
STEP command
using, to replay recorded statements 177
stepping
through a program 175
through C++ programs 309
stepping, introduction to 14
STMT suboption of TEST compiler option (PL/I), effect of 38
STMT, how Enterprise COBOL for z/OS, Version 4, handles 33
stopping
z/OS Debugger session
19
storage
classes, for C 304
displaying, introduction to 16
LangX COBOL, displaying 210
storage errors, finding
overwrite
in assembler 250
in C 228
in C++ 240
in COBOL 205
in LangX COBOL 211
in PL/I 219
uninitialized
in C 229
in C++ 240
STORAGE run-time option, specifying 113
storage, raw
C, displaying 227
C++, displaying 238
COBOL, displaying 203
PL/I, displaying 218
stored procedures
Db2, debugging 335
string
syntax for searching 167
string substitution, using 251
strings
C, displaying 227
C++, displaying 238
searching for in a window 167
SUB Db2 stored procedures 79
substitution, using string 251
SUBSYS
with C programs, what to do about 64
Suffix area, description of 152
suppressing the display of warning messages 175
SWAP command compared to scroll commands 165
SWAP command, when to use 165
SYM suboption of TEST compiler option (PL/I), effect of 37
symbol tables, removing 360
syntax diagrams
how to read xxi
SYSCDBG 403
SYSDEBUG 403
system commands, issuing, z/OS Debugger 160

T

TCP/IP, specifying for IMS programs (IPv4 or IPv6 formats)
343
template in C++ 310
temporary storage queue
how z/OS Debugger uses
82
temporary storage queue, comparing VSAM with 82
Term 85
Terminal Id
in DTCN, description of 86
Terminal Interface Manager
example of 130
how to start 131
terminal mode, selecting correct Display ID for each type of
90
terminology, z/OS Debugger xx
TEST compiler option
C, how to choose 41, 46
COBOL, how to choose 27
debugging C when only a few parts are compiled with
225
debugging C++ when only a few parts are compiled with
236
debugging COBOL when only a few parts are compiled
with 202
debugging LangX COBOL when only a few parts are
compiled with 211
debugging PL/I when only a few parts are compiled with
217
for PL/I 34
PL/I, how to choose 35
specifying NUMBER option with 30
using #pragma statement to specify 43
versus DEBUG runtime option (COBOL) 30
TEST compiler option (C), effect of 43
TEST compiler option (C++), effect of 47
TEST run-time option
as parameter on RUN subcommand 334
different ways to specify 53
for CICS programs, how to specify 55
for Db2 programs, how to specify 55
for Db2 stored procedures, how to specify 55
for IMS programs, how to specify 55
for JES batch programs, how to specify 54
for PL/I 286
for TSO programs, how to specify 54
for UNIX System Services programs, how to specify 54
specifying with #pragma 114
suboption processing order 108
TEST runtime option
example of 110
TEST suboptions, redefining at runtime 108
this pointer, in C++ 236
TIM
use to create TEST runtime options data set 102
trace file for DTCN Profiles or DTSP Profile 524
trace, generating a COBOL run-time paragraph 204
traceback, COBOL routine 203
traceback, function
in assembler 249
in C 227
in C++ 238

traceback, function *(continued)*
in PL/I 218
traceback, LangX COBOL routine 211
tracing run-time path
in C 227
in C++ 239
in COBOL 203
in PL/I 218
Tran 85
Transaction Id
in DTCN, description of 86
TRAP, Language Environment run-time option 194, 375
TRAP(ON) run-time option, specifying 113
trigraph 262
trigraphs
using with C 261
TSO
starting z/OS Debugger using TEST run-time option 143
TSO command
using to debug Db2 program 334
TSO, starting z/OS Debugger under 133
TSQ 82

U

UEN 10
uninitialized storage errors, finding
in C 229
in C++ 240
UNIX System Services
compiling a C program on 63
compiling a C++ program 63
compiling a Enterprise PL/I program on 62
using z/OS Debugger with 365
unsupported
HLL modules, coexistence with 379
PL/I language elements 291
UP, SCROLL command 165
URM debugging 91
USE file 108
User ID
in DTCN, description of 88

V

values
assigning to C and C++ variables 295
assigning to COBOL variables 269
variable
automonitor 16
changing value of 17
continuous display 15
displaying value of 15
modifying value
in C 224
in C++ 235
in COBOL 200
in PL/I 216
one-time and continuous display 16
one-time display 15
using SET AUTOMONITOR ON command to monitor value of 186
variable *(continued)*
value, displaying 182
variable, displaying data type of 185
variables
accessing program, for C and C++ 294
accessing program, for COBOL 269
assigning values to, for C and C++ 295
assigning values to, for COBOL 269
compatible attributes in multiple languages 377
displaying, for C and C++ 294
displaying, for COBOL 270
HLL 372
qualifying 373
session
declaring, for C and C++ 296
session, for PL/I 286
viewing and modifying data members in C++ 236
VS COBOL II
compiler options to use 68
VS COBOL II programs, additional preparation steps for 30
VS COBOL II programs, debugging 277
VS COBOL II, finding list for 278
VSAM, comparing CICS temporary storage queue with 82
VTAM
starting a debugging session through a, terminal 131

W

warning, for PL/I 290
window
description of Memory 154
window id area, z/OS Debugger 159
window, error numbers in 193
windows, z/OS Debugger
physical
changing configuration 252
opening and closing 252
resizing 253
windows, z/OS Debugger session
panel
opening and closing 253
zooming 253
Working-Storage Section, displaying 184

X

XPLINK
restriction on applications that use 330

Z

z/OS Debugger
C and C++ commands, interpretive subset 293
COBOL commands, interpretive subset 267
commands, subset 372
condition handling 375
data sets 401
enhancing performance of 76
evaluation of HLL expressions 371
exception handling, for C and C++ and PL/I 376
interfaces 5
interpretation of HLL variables 371
list of supported compilers 3

z/OS Debugger *(continued)*
list of supported subsystems 4
multilanguage programs, using 376
PL/I commands, interpretive subset 283
starting at different points 108
starting under CICS 139
starting under MVS in TSO 133
starting your program with 143
starting, by using z/OS Debugger Utilities 115
stopping, session 19
terminology xx
using in batch mode 493
z/OS Debugger Setup Utility 115
z/OS Debugger Utilities
creating private message region for IMS program 345
creating setup file for IMS program 344
Deferred Breakpoints 9
how to use, to link-edit 73
instructions for modifying and using a setup file 419
instructions for running a program in batch 419
JCL Wizard 9
Non-CICS Debug Session Start and Stop Message Viewer 9
specifying TEST runtime options for IMS program 95
starting your program 117
using to assemble and create 72
ZOOM command, how and where to use 166
zooming a window, z/OS Debugger 253

Product Number: 5724-T07