

Debug Tool for z/OS



User's Guide

Version 13.1

Debug Tool for z/OS



User's Guide

Version 13.1

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 569.

Fifth Edition (October 2015)

This edition applies to Debug Tool for z/OS, Version 13.1 (Program Number 5655-Q10 with the PTF for APAR PI37275), which supports the following compilers:

- AD/Cycle C/370 Version 1 Release 2 (Program Number 5688-216)
- C/C++ for MVS/ESA Version 3 (Program Number 5655-121)
- C/C++ feature of OS/390 (Program Number 5647-A01)
- C/C++ feature of z/OS Version 1 (Program Number 5694-A01)
- C/C++ feature of z/OS Version 2 (Program Number 5650-ZOS)
- OS/VS COBOL, Version 1 Release 2.4 (5740-CB1) - with limitations
- VS COBOL II Version 1 Release 3 and Version 1 Release 4 (Program Numbers 5668-958, 5688-023) - with limitations
- COBOL/370 Version 1 Release 1 (Program Number 5688-197)
- COBOL for MVS & VM Version 1 Release 2 (Program Number 5688-197)
- COBOL for OS/390 & VM Version 2 (Program Number 5648-A25)
- Enterprise COBOL for z/OS and OS/390 Version 3 (Program Number 5655-G53)
- Enterprise COBOL for z/OS Version 4 (Program Number 5655-S71)
- Enterprise COBOL for z/OS Version 5 Release 1 (Program Number 5655-W32)
- High Level Assembler for MVS & VM & VSE Version 1 Release 4, Version 1 Release 5, Version 1 Release 6 (Program Number 5696-234)
- OS PL/I Version 2 Release 1, Version 2 Release 2, Version 2 Release 3 (Program Numbers 5668-909, 5668-910) - with limitations
- PL/I for MVS & VM Version 1 Release 1 (Program Number 5688-235)
- VisualAge PL/I for OS/390 Version 2 Release 2 (Program Number 5655-B22)
- Enterprise PL/I for z/OS and OS/390 Version 3 (Program Number 5655-H31)
- Enterprise PL/I for z/OS Version 4.4 and earlier (Program Number 5655-W67)

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

You can access publications online at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss

You can find out more about Debug Tool by visiting the IBM Web site for Debug Tool at: www.ibm.com/software/products/us/en/debugtool

© Copyright IBM Corporation 1992, 2015.

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

Contents

About this document	xiii
Who might use this document	xiii
Accessing z/OS licensed documents on the Internet	xiii
How this document is organized	xiv
Terms used in this document	xvi
How to read syntax diagrams	xviii
Symbols	xviii
Syntax items	xix
Syntax examples	xix
How to send your comments	xx

Summary of changes	xxi
-------------------------------------	------------

Part 1. Getting started with Debug Tool 1

Chapter 1. Debug Tool: overview 3

Debug Tool interfaces	4
Batch mode	5
Full-screen mode	5
Full-screen mode using the Terminal Interface Manager	5
Remote debug mode	6
Debug Tool Utilities	6
Debug Tool Utilities: Job Card	6
Debug Tool Utilities: Program Preparation	6
Debug Tool Utilities: Debug Tool Setup File	7
Debug Tool Utilities: Code Coverage	7
Debug Tool Utilities: IMS TM Functions	7
Debug Tool Utilities: Load Module Analyzer	8
Debug Tool Utilities: Debug Tool User Exit Data Set	8
Debug Tool Utilities: Other IBM Problem Determination Tools	8
Debug Tool Utilities: JCL for Batch Debugging	8
Debug Tool Utilities: IMS BTS Debugging	8
Debug Tool Utilities: JCL to Setup File Conversion	8
Debug Tool Utilities: Delay Debug Profile	8
Debug Tool Utilities: IMS Transaction and User ID Cross Reference Table	9
Debug Tool Utilities: Non-CICS Debug Session Start and Stop Message Viewer	9
Debug Tool Utilities: Debug Tool Code Coverage	9
Debug Tool Utilities: Debug Tool Deferred Breakpoints	9
Debug Tool Utilities: Debug Tool JCL Wizard	9
Starting Debug Tool 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 Debug Tool	12
The Debug Tool full screen interface	13

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	17
Changing the value of a variable	17
Skipping a breakpoint	18
Clearing a breakpoint	18
Recording and replaying statements	18
Stopping Debug Tool	19

Part 2. Preparing your program for debugging 21

Chapter 3. Planning your debug session 23

Choosing compiler options for debugging	24
Choosing TEST or NOTEST compiler suboptions for COBOL programs	27
Choosing TEST or NOTEST compiler suboptions for PL/I programs	33
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	52
Choosing a method or methods for starting Debug Tool	54
Choosing how to debug old COBOL programs	58
Creating deferred breakpoints for COBOL and PL/I programs	59

Chapter 4. Updating your processes so you can debug programs with Debug Tool 61

Update your compilation, assembly, and linking process	61
Compiling your program without using Debug Tool Utilities	61
Compiling your program by using Debug Tool Utilities	63
Compiling a Enterprise PL/I program on an HFS file system	64
Compiling your C program with c89 or c++	64
Compiling a C program on an HFS file system	65
Compiling a C++ program on an HFS file system	66
Update your library and promotion process	66

Make the modifications necessary to implement your preferred method of starting Debug Tool. .. 67

Chapter 5. Preparing a LangX COBOL program 71

Compiling your OS/VS COBOL program 71

Compiling your VS COBOL II program 72

Compiling your Enterprise COBOL program 72

Creating the EQALANGX file for LangX COBOL programs 72

Link-editing your program 74

Chapter 6. Preparing an assembler program 75

Before you assemble your program 75

Assembling your program 75

Creating the EQALANGX file for an assembler program 76

Assembling your program and creating EQALANGX 77

Link-editing your program 77

Restrictions for link-editing your assembler program 78

Chapter 7. Preparing a DB2 program 79

Processing SQL statements 79

Linking DB2 programs for debugging 81

Binding DB2 programs for debugging 82

Chapter 8. Preparing a DB2 stored procedures program 83

Chapter 9. Preparing a CICS program 87

Link-editing EQADCCXT into your program 87

Creating and storing a DTCN profile 88

Displaying a list of active DTCN profiles and managing DTCN profiles 91

Description of fields on the DTCN Primary Menu screen 92

Description of fields on the DTCN Menu 2 screen 97

Description of fields on the DTCN Advanced Options screen 98

Creating and storing debugging profiles with CADP 99

Starting Debug Tool for non-Language Environment programs under CICS 99

Passing runtime parameters to Debug Tool for non-Language Environment programs under CICS 100

Chapter 10. Preparing an IMS program 101

Starting Debug Tool under IMS by using CEEUOPT or CEEROPT 101

Managing runtime options for IMSplex users by using Debug Tool Utilities 102

Setting up the DFSBXITA user exit routine 102

Chapter 11. Specifying the TEST runtime options through the Language Environment user exit 105

Editing the source code of CEEBXITA 106

Modifying the naming pattern. 107

Modifying the message display level 108

Modifying the call back routine registration .. 108

Activate the cross reference function and modifying the cross reference table data set name 108

Comparing the two methods of linking CEEBXITA 109

Linking the CEEBXITA user exit into your application program 109

Linking the CEEBXITA user exit into a private copy of a Language Environment runtime module . . 110

Creating and managing the TEST runtime options data set 111

Creating and managing the TEST runtime options data set by using Terminal Interface Manager (TIM) 111

Creating and managing the TEST runtime options data set by using Debug Tool Utilities . 113

Part 3. Starting Debug Tool 115

Chapter 12. Writing the TEST run-time option string. 117

Special considerations while using the TEST run-time option. 117

Defining TEST suboptions in your program .. 117

Suboptions and NOTEST 117

Implicit breakpoints 118

Primary commands file and USE file 118

Running in batch mode 118

Starting Debug Tool at different points 118

Session log 119

Precedence of Language Environment runtime options 119

Example: TEST run-time options 120

Specifying additional run-time options with VS COBOL II and PL/I programs. 121

Specifying the STORAGE run-time option 121

Specifying the TRAP(ON) run-time option .. 121

Specifying TEST run-time option with #pragma runopts in C and C++ 122

Chapter 13. Starting Debug Tool from the Debug Tool Utilities 123

Creating the setup file 123

Editing an existing setup file 123

Copying information into a setup file from an existing JCL 124

Entering file allocation statements, runtime options, and program parameters 124

Saving your setup file 126

Starting your program 126

Chapter 14. Starting Debug Tool from a program	127
Starting Debug Tool with CEETEST	127
Additional notes about starting Debug Tool with CEETEST	129
Example: using CEETEST to start Debug Tool from C/C++	130
Example: using CEETEST to start Debug Tool from COBOL	131
Example: using CEETEST to start Debug Tool from PL/I	132
Starting Debug Tool with PLITEST	134
Starting Debug Tool with the __ctest() function	135

Chapter 15. Starting Debug Tool in batch mode	137
Example: JCL that runs Debug Tool in batch mode	137
Modifying the example to debug in full-screen mode	138

Chapter 16. Starting Debug Tool for batch or TSO programs	139
Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal	139
Starting Debug Tool for programs that start in Language Environment	141
Example: Allocating Debug Tool load library data set	142
Example: Allocating Debug Tool files	142
Starting Debug Tool for programs that start outside of Language Environment	143
Passing parameters to EQANMDBG.	143
Example: Modifying JCL that invokes an assembler DB2 program running in a batch TSO environment.	146

Chapter 17. Starting Debug Tool under CICS	147
Comparison of methods for starting Debug Tool under CICS	147
Starting Debug Tool under CICS by using DTCN	148
Ending a CICS debugging session that was started by DTCN	149
Example: How Debug Tool chooses a CICS program for debugging	149
Starting Debug Tool for CICS programs by using CADP.	149
Starting Debug Tool under CICS by using CEEUOPT	150
Starting Debug Tool under CICS by using compiler directives.	150

Chapter 18. Starting a full-screen debug session	151
---	------------

Chapter 19. Starting Debug Tool in other environments.	153
Starting Debug Tool from DB2 stored procedures	153

Part 4. Debugging your programs in full-screen mode 155

Chapter 20. Using full-screen mode: overview	157
Debug Tool session panel	157
Session panel header	158
Source window.	160
Monitor window	161
Log window.	162
Memory window	163
Command pop-up window.	164
List pop-up window	164
Creating a preferences file	165
Displaying the source	165
Changing which file appears in the Source window	166
Entering commands on the session panel	167
Order in which Debug Tool accepts commands from the session panel	170
Using the session panel command line	170
Issuing system commands	171
Entering prefix commands on specific lines or statements	171
Entering multiple commands in the Memory window	172
Using commands that are sensitive to the cursor position	173
Using Program Function (PF) keys to enter commands	173
Initial PF key settings	173
Retrieving previous commands	174
Composing commands from lines in the Log and Source windows	174
Opening the Command pop-up window to enter long Debug Tool commands	175
Navigating through Debug Tool windows.	175
Moving the cursor between windows	176
Switching between the Memory window and Log window.	176
Scrolling through the physical windows	176
Enlarging a physical window	177
Scrolling to a particular line number.	178
Finding a string in a window	178
Displaying the line at which execution halted	180
Navigating through the Memory window	181
Creating a commands file	182
Recording your debug session in a log file	183
Creating the log file	184
Recording how many times each source line runs	185
Recording the breakpoints encountered.	186

Setting breakpoints to halt your program at a line	186
Setting breakpoints in a load module that is not loaded or in a program that is not active	186
Controlling how Debug Tool handles warnings about invalid data in comparisons	187
Stepping through or running your program	188
Recording and replaying statements	189
Saving and restoring settings, breakpoints, and monitor specifications	191
Saving and restoring automatically	193
Disabling the automatic saving and restoring of breakpoints, monitors, and settings	194
Restoring manually	194
Performance considerations in multi-enclave environments	195
Displaying and monitoring the value of a variable	196
One-time display of the value of variables	196
Adding variables to the Monitor window	197
Displaying the Working-Storage Section of a COBOL program in the Monitor window	198
Displaying the data type of a variable in the Monitor window	199
Replacing a variable in the Monitor window with another variable	199
Adding variables to the Monitor window automatically	200
How Debug Tool handles characters that cannot be displayed in their declared data type	203
Modifying characters that cannot be displayed in their declared data type	203
Formatting values in the Monitor window	204
Displaying values in hexadecimal format	204
Monitoring the value of variables in hexadecimal format	205
Modifying variables or storage by using a command	205
Modifying variables or storage by typing over an existing value	206
Opening and closing the Monitor window	206
Displaying and modifying memory through the Memory window	206
Modifying memory through the hexadecimal data area	207
Managing file allocations	207
Displaying error numbers for messages in the Log window	209
Displaying a list of compile units known to Debug Tool	209
Requesting an attention interrupt during interactive sessions	210
Ending a full-screen debug session	210
Chapter 21. Debugging a COBOL program in full-screen mode	213
Example: sample COBOL program for debugging	213
Halting when certain routines are called in COBOL	216
Identifying the statement where your COBOL program has stopped	217
Modifying the value of a COBOL variable	217
Halting on a COBOL line only if a condition is true	218

Debugging COBOL when only a few parts are compiled with TEST	218
Capturing COBOL I/O to the system console	219
Displaying raw storage in COBOL	219
Getting a COBOL routine traceback	220
Tracing the run-time path for COBOL code compiled with TEST	220
Generating a COBOL run-time paragraph trace	221
Finding unexpected storage overwrite errors in COBOL	222
Halting before calling an invalid program in COBOL	222

Chapter 22. Debugging a LangX COBOL program in full-screen mode **225**

Example: sample LangX COBOL program for debugging	225
Defining a compilation unit as LangX COBOL and loading debug information	227
Defining a compilation unit in a different load module as LangX COBOL	228
Halting when certain LangX COBOL programs are called	228
Identifying the statement where your LangX COBOL program has stopped	228
Displaying and modifying the value of LangX COBOL variables or storage	229
Halting on a line in LangX COBOL only if a condition is true	229
Debugging LangX COBOL when debug information is only available for a few parts	229
Getting a LangX COBOL program traceback	230
Finding unexpected storage overwrite errors in LangX COBOL	230

Chapter 23. Debugging a PL/I program in full-screen mode **231**

Example: sample PL/I program for debugging	231
Halting when certain PL/I functions are called	234
Identifying the statement where your PL/I program has stopped	234
Modifying the value of a PL/I variable	235
Halting on a PL/I line only if a condition is true	235
Debugging PL/I when only a few parts are compiled with TEST	236
Displaying raw storage in PL/I	236
Getting a PL/I function traceback	236
Tracing the run-time path for PL/I code compiled with TEST	237
Finding unexpected storage overwrite errors in PL/I	238
Halting before calling an undefined program in PL/I	238

Chapter 24. Debugging a C program in full-screen mode **241**

Example: sample C program for debugging	241
Halting when certain functions are called in C	244
Modifying the value of a C variable	245
Halting on a line in C only if a condition is true	245

Debugging C when only a few parts are compiled with TEST	246
Capturing C output to stdout	246
Capturing C input to stdin	247
Calling a C function from Debug Tool	247
Displaying raw storage in C	247
Debugging a C DLL	248
Getting a function traceback in C.	248
Tracing the run-time path for C code compiled with TEST	248
Finding unexpected storage overwrite errors in C	249
Finding uninitialized storage errors in C	249
Halting before calling a NULL C function	250

Chapter 25. Debugging a C++ program in full-screen mode 251

Example: sample C++ program for debugging	251
Halting when certain functions are called in C++	255
Modifying the value of a C++ variable	256
Halting on a line in C++ only if a condition is true	257
Viewing and modifying data members of the this pointer in C++	257
Debugging C++ when only a few parts are compiled with TEST	257
Capturing C++ output to stdout	258
Capturing C++ input to stdin	258
Calling a C++ function from Debug Tool	259
Displaying raw storage in C++	259
Debugging a C++ DLL	259
Getting a function traceback in C++.	260
Tracing the run-time path for C++ code compiled with TEST	260
Finding unexpected storage overwrite errors in C++	261
Finding uninitialized storage errors in C++	261
Halting before calling a NULL C++ function	262

Chapter 26. Debugging an assembler program in full-screen mode 263

Example: sample assembler program for debugging	263
Defining a compilation unit as assembler and loading debug data	266
Deferred LDDs	267
Re-appearance of an assembler CU	267
Multiple compilation units in a single assembly	267
Loading debug data from multiple CSECTs in a single assembly using one LDD command.	268
Loading debug data from multiple CSECTs in a single assembly using separate LDD commands.	268
Debugging multiple CSECTs in a single assembly after the debug data is loaded	268
Halting when certain assembler routines are called	269
Identifying the statement where your assembler program has stopped.	269
Displaying and modifying the value of assembler variables or storage	269
Converting a hexadecimal address to a symbolic address	270
Halting on a line in assembler only if a condition is true	270

Getting an assembler routine traceback	270
Finding unexpected storage overwrite errors in assembler	271

Chapter 27. Customizing your full-screen session 273

Defining PF keys	273
Defining a symbol for commands or other strings	273
Customizing the layout of physical windows on the session panel	274
Opening and closing physical windows	275
Resizing physical windows.	275
Zooming a window to occupy the whole screen	276
Customizing session panel colors.	276
Customizing profile settings	277
Saving customized settings in a preferences file	279
Saving and restoring customizations between Debug Tool sessions	279

Part 5. Debugging your programs by using Debug Tool commands . 281

Chapter 28. Entering Debug Tool commands 283

Using uppercase, lowercase, and DBCS in Debug Tool commands	283
DBCS	283
Character case and DBCS in C and C++	284
Character case in COBOL and PL/I	284
Abbreviating Debug Tool keywords	284
Entering multiline commands in full-screen	285
Entering multiline commands in a commands file	285
Entering multiline commands without continuation	286
Using blanks in Debug Tool commands	286
Entering comments in Debug Tool commands	287
Using constants in Debug Tool commands.	287
Getting online help for Debug Tool command syntax	288

Chapter 29. Debugging COBOL programs 289

Debug Tool commands that resemble COBOL statements	289
COBOL command format	289
COBOL compiler options in effect for Debug Tool commands	290
COBOL reserved keywords.	290
Using COBOL variables with Debug Tool	291
Accessing COBOL variables	291
Assigning values to COBOL variables	291
Example: assigning values to COBOL variables	291
Displaying values of COBOL variables	292
Using DBCS characters in COBOL	293
%PATHCODE values for COBOL.	293
Declaring session variables in COBOL	295
Debug Tool evaluation of COBOL expressions	295
Displaying the results of COBOL expression evaluation	296
Using constants in COBOL expressions.	296

Using Debug Tool functions with COBOL	297
Using %HEX with COBOL	297
Using the %STORAGE function with COBOL	297
Qualifying variables and changing the point of view in COBOL	297
Qualifying variables in COBOL	297
Changing the point of view in COBOL	299
Considerations when debugging a COBOL class	299
Debugging VS COBOL II programs	300
Finding the listing of a VS COBOL II program	300

**Chapter 30. Debugging a LangX
COBOL program 303**

Loading a LangX COBOL program's debug information	303
Debug Tool session panel while debugging a LangX COBOL program	304
Restrictions for debugging a LangX COBOL program	304
%PATHCODE values for LangX COBOL programs	306
Restrictions for debugging non-Language Environment programs	306

Chapter 31. Debugging PL/I programs 307

Debug Tool subset of PL/I commands	307
PL/I language statements	307
%PATHCODE values for PL/I.	308
PL/I conditions and condition handling	309
Entering commands in PL/I DBCS freeform format	310
Initializing Debug Tool for PL/I programs when TEST(ERROR, ...) run-time option is in effect . ..	310
Debug Tool enhancements to LIST STORAGE PL/I command	310
PL/I support for Debug Tool session variables ..	310
Accessing PL/I program variables	311
Accessing PL/I structures	311
Debug Tool evaluation of PL/I expressions . . .	313
Supported PL/I built-in functions	314
Using SET WARNING PL/I command with built-in functions	316
Unsupported PL/I language elements	316
Debugging OS PL/I programs.	316
Restrictions while debugging Enterprise PL/I programs.	317

**Chapter 32. Debugging C and C++
programs 319**

Debug Tool commands that resemble C and C++ commands	319
Using C and C++ variables with Debug Tool . . .	320
Accessing C and C++ program variables . . .	320
Displaying values of C and C++ variables or expressions	320
Assigning values to C and C++ variables . . .	321
%PATHCODE values for C and C++	322
Declaring session variables with C and C++ . . .	322
C and C++ expressions	323
Calling C and C++ functions from Debug Tool ..	324
C reserved keywords	325
C operators and operands	326

Language Environment conditions and their C and C++ equivalents	326
Debug Tool evaluation of C and C++ expressions	327
Intercepting files when debugging C and C++ programs.	328
Scope of objects in C and C++.	330
Storage classes in C and C++	331
Blocks and block identifiers for C	332
Blocks and block identifiers for C++.	332
Example: referencing variables and setting breakpoints in C and C++ blocks.	333
Scope and visibility of objects in C and C++ programs.	333
Blocks and block identifiers in C and C++ programs.	334
Displaying environmental information for C and C++ programs	334
Qualifying variables and changing the point of view in C and C++	335
Qualifying variables in C and C++	335
Changing the point of view in C and C++. ..	336
Example: using qualification in C.	336
Stepping through C++ programs	338
Setting breakpoints in C++.	338
Setting breakpoints in C++ using AT ENTRY/EXIT	338
Setting breakpoints in C++ using AT CALL ..	339
Examining C++ objects	339
Example: displaying attributes of C++ objects	339
Monitoring storage in C++.	340
Example: monitoring and modifying registers and storage in C	341

**Chapter 33. Debugging an assembler
program 343**

The SET ASSEMBLER and SET DISASSEMBLY commands	343
Loading an assembler program's debug information	343
Debug Tool session panel while debugging an assembler program	344
%PATHCODE values for assembler programs ..	345
Using the STANDARD and NOMACGEN view	347
Debugging non-reentrant assembler	347
Manipulating breakpoints in non-reentrant assembler load modules.	348
Manipulating local variables in non-reentrant assembler load modules.	348
Restrictions for debugging an assembler program	348
Restrictions for debugging a Language Environment assembler MAIN program	350
Restrictions on setting breakpoints in the prologue of Language Environment assembler programs.	350
Restrictions for debugging non-Language Environment programs	350
Restrictions for debugging assembler code that uses instructions as data.	351
Restrictions for debugging self-modifying assembler code	351

Chapter 34. Debugging a disassembled program	355
The SET ASSEMBLER and SET DISASSEMBLY commands	355
Capabilities of the disassembly view	355
Starting the disassembly view	356
The disassembly view	356
Performing single-step operations in the disassembly view	357
Setting breakpoints in the disassembly view	357
Restrictions for debugging self-modifying code	357
Displaying and modifying registers in the disassembly view	358
Displaying and modifying storage in the disassembly view	358
Changing the program displayed in the disassembly view	358
Restrictions for the disassembly view	358

Part 6. Debugging in different environments 361

Chapter 35. Debugging DB2 programs 363	
Debugging DB2 programs in batch mode	363
Debugging DB2 programs in full-screen mode	364

Chapter 36. Debugging DB2 stored procedures 367	
Resolving some common problems while debugging DB2 stored procedures	367

Chapter 37. Debugging IMS programs 369	
Using IMS Transaction Isolation to create a private message-processing region and select transactions to debug	369
Debugging IMS batch programs interactively by running BTS in TSO foreground	372
Debugging IMS batch programs in batch mode	372
Debugging non-Language Environment IMS MPPs	372
Verifying configuration and starting a region for non-Language Environment IMS MPPs.	373
Choosing an interface and gathering information for non-Language Environment IMS MPPs	373
Running the EQASET transaction for non-Language Environment IMS MPPs.	373
Debugging Language Environment IMS MPPs without issuing /SIGN ON.	375
Syntax of the EQASET transaction for Language Environment MPPs	375
Creating setup file for your IMS program by using Debug Tool Utilities	376
Using IMS message region templates to dynamically swap transaction class and debug in a private message region	377
Placing breakpoints in IMS applications to avoid the appearance of Debug Tool becoming unresponsive	379

Chapter 38. Debugging CICS programs 381	
Displaying the contents of channels and containers	381
Controlling pattern-match breakpoints with the DISABLE and ENABLE commands	383
Preventing Debug Tool from stopping at EXEC CICS RETURN	385
Early detection of CICS storage violations	385
Saving settings while debugging a pseudo-conversational CICS program	386
Saving and restoring breakpoints and monitor specifications for CICS programs	386
Restrictions when debugging under CICS	386
Accessing CICS resources during a debugging session	387
Accessing CICS storage before or after a debugging session	388

Chapter 39. Debugging ISPF applications 389

Chapter 40. Debugging programs in a production environment. 393	
Fine-tuning your programs for Debug Tool	393
Removing hooks	393
Removing statement and symbol tables.	394
Debugging without hooks, statement tables, and symbol tables	395
Debugging optimized COBOL programs	396

Chapter 41. Debugging UNIX System Services programs 399	
Debugging MVS POSIX programs	399

Chapter 42. Debugging non-Language Environment programs 401	
Debugging exclusively non-Language Environment programs.	401
Debugging MVS batch or TSO non-Language Environment initial programs	401
Debugging CICS non-Language Environment assembler or non-Language Environment COBOL initial programs	402

Part 7. Debugging complex applications 403

Chapter 43. Debugging multilanguage applications 405	
Debug Tool evaluation of HLL expressions	405
Debug Tool interpretation of HLL variables and constants	406
HLL variables	406
HLL constants	406
Debug Tool commands that resemble HLL commands	406
Qualifying variables and changing the point of view	407

Qualifying variables	407
Changing the point of view	409
Handling conditions and exceptions in Debug Tool	409
Handling conditions in Debug Tool	410
Handling exceptions within expressions (C and C++ and PL/I only)	411
Debugging multilanguage applications	411
Debugging an application fully supported by Language Environment	412
Using session variables across different programming languages	412
Creating a commands file that can be used across different programming languages	414
Coexistence with other debuggers	414
Coexistence with unsupported HLL modules	414
Chapter 44. Debugging multithreading programs	415
Restrictions when debugging multithreading applications	415
Chapter 45. Debugging across multiple processes and enclaves	417
Starting Debug Tool within an enclave	417
Viewing Debug Tool windows across multiple enclaves	417
Ending a Debug Tool session within multiple enclaves	418
Using Debug Tool commands within multiple enclaves	418
Chapter 46. Debugging a multiple-enclave interlanguage communication (ILC) application	423
Chapter 47. Debugging programs called by Java native methods	425
Chapter 48. Solving problems in complex applications	427
Debugging programs loaded from library lookaside (LLA)	427
Debugging user programs that use system prefixed names	427
Displaying system prefixes	428
Debugging programs with names similar to system components	428
Debugging programs containing data-only modules	428
Optimizing the debugging of large applications	429
Using explicit debug mode to load debug data for only specific modules	429
Excluding specific load modules and compile units	430
Displaying current NAMES settings	431
Using the EQAOPTS NAMES command to include or exclude the initial load module	431
Using delay debug mode to delay starting of a debug session	431

Usage notes	432
Debugging subtasks created by the ATTACH assembler macro	433
Debugging tasks running under a generic user ID by using Terminal Interface Manager	434

Part 8. Appendixes 437

Appendix A. Data sets used by Debug Tool 439

Appendix B. How does Debug Tool locate source, listing, or separate debug files? 445

How does Debug Tool locate source and listing files?	447
How does Debug Tool locate COBOL and PL/I separate debug file files?	447
How does Debug Tool locate EQALANGX files	448
How does Debug Tool locate the C/C++ source file and the .dbg file?	448
How does Debug Tool locate the C/C++ .mdbg file?	449

Appendix C. Examples: Preparing programs and modifying setup files with Debug Tool Utilities 451

Creating personal data sets	451
Starting Debug Tool Utilities	452
Compiling or assembling your program by using Debug Tool Utilities	452
Modifying and using a setup file	455
Run the program in foreground	455
Run the program in batch	456

Appendix D. Debug Tool JCL Wizard 457

Debug Tool JCL Wizard introduction	457
Debug Tool JCL Wizard use cases	458
Help information	458
Debug a Language Environment program by using the Terminal Interface Manager	460
Debug a Language Environment program with the Remote GUI by using the A line command with a Procedure Step Override	465
Debug a non-Language Environment program by using the Terminal Interface Manager	468
Debug a Language Environment DB2 program by using the Remote GUI	472
Debug a non-Language Environment DB2 program by using the Remote GUI	474
Start Code Coverage without an interactive Debug Tool session	477
Start Code Coverage with an interactive Debug Tool session using the Terminal Interface Manager	479
Debug a Language Environment VS COBOL II program compiled with the NOTEST option by using the Terminal Interface Manager	481

	Debug a non-Language Environment program when the debug member does not match the program name	484
--	--	-----

Appendix E. Debug Tool Code Coverage 491

Overview of Debug Tool Code Coverage	491
Introduction to Debug Tool Code Coverage ..	491
Collecting code coverage observations with Debug Tool	492
Code coverage selection and extraction process	493
Code coverage reporting process	493
Code coverage Viewer	494
Code coverage by using Debug Tool.	495
Setup	495
Generating code coverage extracted observations.	499
Debug Tool Utilities Option E	502
Annotated listing format	510
Batch facilities	515
Generating code coverage for CICS transactions	516
XML tags for code coverage	516
XML tags definition for the Observation file ..	516
XML tag hierarchy for the Observation file ..	519
XML Tags used in the Options file	520
XML tags used in the Selection file	520

Appendix F. Notes on debugging in batch mode 523

Appendix G. Using IMS message region templates to dynamically swap transaction class and debug in a private message region 525

Appendix H. Displaying and modifying CICS storage with DTST. 527

Starting DTST	527
Examples of starting DTST	527
Modifying storage through the DTST storage window	529
Navigating through the DTST storage window ..	529
DTST storage window	530
Navigation keys for help screens	531
Syntax of the DTST transaction	532
Examples.	533

Appendix I. Debug Tool Load Module Analyzer 535

Choosing a method to start Load Module Analyzer	535
Starting the Load Module Analyzer by using JCL	535
Starting the Load Module Analyzer by using Debug Tool Utilities	535
Description of the JCL statements to use with Load Module Analyzer	535
Description of DD names used by Load Module Analyzer	536

Description of parameters used by Load Module Analyzer	537
Description of EQASYSPF file format	539
Description of EQAPGMNM file format	540
Description of program output created by Load Module Analyzer	540
Description of output contents created by Load Module Analyzer	541
Example: Output created by Load Module Analyzer for an OS/VS COBOL load module ..	541
Example: Compiler options output created by Load Module Analyzer	541

Appendix J. Running NEWCOPY on programs by using DTNP transaction . 543

Appendix K. Installing the IBM Debug Tool plug-ins 545

Instrument JCL for Debug Tool Debugging Plug-in	548
Debug Tool Code Coverage Plug-in	551
Debug Tool Load Module Analyzer Plug-in	553
Locating the trace file of the DTCN Profile, the DTSP Profile, Instrument JCL for Debug Tool Debugging, Code Coverage, and Load Module Analyzer view	555
Example: .debugtool.dtcn.trace file	555
Examples: .debugtool.dtsp.trace files.	556
Examples: .debugtool.bjfd.trace files	556

Appendix L. Support resources and problem solving information 559

Searching knowledge bases.	559
Searching the information center	559
Searching product support documents	559
Getting fixes.	561
Subscribing to support updates	561
RSS feeds and social media subscriptions	561
My Notifications	561
Contacting IBM Support.	562
Define the problem and determine the severity of the problem	563
Gather diagnostic information	563
Submit the problem to IBM Support.	564

Appendix M. Accessibility 567

Using assistive technologies	567
Keyboard navigation of the user interface	567
Accessibility of this document	567

Notices 569

Copyright license	570
Programming interface information	570
Trademarks and service marks	570

Glossary 571

Bibliography. 579

Debug tool publications	579
High level language publications	579

Related publications 581
Softcopy publications. 582

Index 583

About this document

Debug Tool 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. Debug Tool 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.

This document contains a summary of commands, built-in functions, and EQAOPTS commands provided by Debug Tool. Each topic contains the name of the command, built-in function, or EQAOPTS command and then a syntax diagram. For more information about each command or built-in function, refer to the *Debug Tool Reference and Messages*. For more information about each EQAOPTS command, see the *Debug Tool Customization Guide* or *Debug Tool Reference and Messages*.

Who might use this document

This document is intended for programmers using Debug Tool 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.

Debug Tool 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.

How this document is organized

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 Debug Tool. The following list describes each chapter:
 - Chapter 1 introduces Debug Tool and describes some of its features.
 - Chapter 2 describes a simple scenario of how to use Debug Tool 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 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 4 describes how to implement the choices you made in chapter 3.
 - Chapter 5 describes how to prepare a LangX COBOL program.
 - Chapter 6 describes how to prepare an assembler program.
 - Chapter 7 describes how to prepare a DB2 program.
 - Chapter 8 describes how to prepare a DB2 stored procedures program.
 - Chapter 9 describes how to prepare a CICS program.
 - Chapter 10 describes how to prepare an IMS program.
 - Chapter 11 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 Debug Tool. The following list describes each chapter:
 - Chapter 12 describes how to write the TEST runtime option to indicate how and when you want to start Debug Tool.
 - Chapter 13 describes how to start Debug Tool from Debug Tool Utilities.
 - Chapter 14 describes how to start Debug Tool from a program.
 - Chapter 15 describes how to start Debug Tool in batch mode.
 - Chapter 16 describes how to start Debug Tool for your batch or TSO programs.
 - Chapter 17 describes how to start Debug Tool from CICS programs.
 - Chapter 18 describes how to start Debug Tool in full-screen mode.

- Chapter 19 describes how to start Debug Tool in full-screen mode using the Terminal Interface Manager. This chapter also describes some tips to starting Debug Tool 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 20 provides overview information about full-screen mode.
 - Chapter 21 provides a sample COBOL program to describe how to debug it in full-screen mode.
 - Chapter 22 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 23 provides a sample PL/I program to describe how to debug it in full-screen mode.
 - Chapter 24 provides a sample C 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 assembler program to describe how to debug it in full-screen mode.
 - Chapter 27 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 Debug Tool commands.
 - Chapter 28 provides information about entering mixed case commands, using DBCS characters, abbreviating commands, entering multiline commands, and entering comments.
 - Chapter 29 describes how to use Debug Tool commands to debug COBOL programs.
 - Chapter 30 describes how to use Debug Tool commands to debug LangX COBOL programs.
 - Chapter 31 describes how to use Debug Tool commands to debug PL/I programs.
 - Chapter 32 describes how to use Debug Tool commands to debug C or C++ programs.
 - Chapter 33 describes how to use Debug Tool commands to debug assembler programs.
 - Chapter 34 describes how to use Debug Tool 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 35 describes how to debug a DB2 program.
 - Chapter 36 describes how to debug a DB2 stored procedure.
 - Chapter 37 describes how to debug an IMS program.
 - Chapter 38 describes how to debug a CICS program.
 - Chapter 39 describes how to debug an ISPF program.
 - Chapter 40 describes how to debug a production-level program.
 - Chapter 41 describes how to debug a program running in the UNIX System Services shell.

- Chapter 42 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 43 describes how to debug a program written in multiple languages.
 - Chapter 44 describes the restrictions when you debug a multithreaded program.
 - Chapter 45 describes how to debug a program that runs across multiple processes and enclaves.
 - Chapter 46 describes how to debug a multiple-enclave interlanguage communication (ILC) application.
 - Chapter 47 describes how to debug programs that are called by Java™ native methods.
 - Chapter 48 describes how to instruct Debug Tool to handle problems created by program names or the large size of programs.
- Part 8 groups together appendixes. The following list describes each appendix:
 - Appendix A describes the data sets that Debug Tool uses to retrieve and store information.
 - Appendix B describes the process Debug Tool 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 Debug Tool Utilities.
 - Appendix D describes the Debug Tool JCL Wizard.
 - Appendix E describes how to use Debug Tool 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 Debug Tool.
 - Appendix J describes how you can use the DTNP transaction, supplied by Debug Tool, 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, Debug Tool Code Coverage, and Load Module Analyzer plug-ins.
 - Appendix L describes the resources that are available to help you solve any problems you might encounter with Debug Tool.
 - Appendix M describes the features and tools available to people with physical disabilities that help them use Debug Tool and Debug Tool 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 Debug Tool, 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.

Debug Tool term	C and C++ equivalent	COBOL or LangX COBOL equivalent	PL/I equivalent	assembler
Compile unit	C and C++ source file	Program	<ul style="list-style-type: none"> • Program • PL/I source file for Enterprise PL/I • A package statement or the name of the main procedure for Enterprise PL/I¹ 	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, and run with the following versions of Language Environment:
 - Language Environment Version 1.9, or later
 - Language Environment Version 1.6, Version 1.7, or Version 1.8, with the PTF for APAR PK33738 applied
 - Compiled with Enterprise PL/I for z/OS, Version 3.5, with the PTFs for APARs PK35230 and PK35489 applied and run with the following versions of Language Environment:
 - Language Environment Version 1.9, or later
 - Language Environment Version 1.6, Version 1.7, or Version 1.8, with the PTF for APAR PK33738 applied

Debug Tool provides facilities that apply only to programs compiled with specific levels of compilers. Because of this, *Debug Tool User's Guide* 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 Debug Tool 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 Debug Tool 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 Debug Tool 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.	▶▶—KEYWORD—required_item————▶▶
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	▶▶—KEYWORD— <div style="display: inline-block; vertical-align: middle; margin-left: 10px;"> required_choice1 required_choice2 </div> ————▶▶
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.	▶▶—KEYWORD— <div style="display: inline-block; vertical-align: middle; margin-left: 10px;"> optional_item </div> ————▶▶
Optional items appear below the main path of the horizontal line.	
Optional choice.	▶▶—KEYWORD— <div style="display: inline-block; vertical-align: middle; margin-left: 10px;"> optional_choice1 optional_choice2 </div> ————▶▶
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.	

Table 1. Syntax examples (continued)

Item	Syntax example
Default.	
<p>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.</p>	
Variable.	
<p>Variables appear in lowercase italics. They represent names or values.</p>	
Repeatable item.	
<p>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</p>	
<p>A character within the arrow means you must separate repeated items with that character.</p>	
<p>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.</p>	
Fragment.	
<p>The \dashv fragment \vdash 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.</p>	

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this document or any other Debug Tool documentation, contact us in one of these ways:

- Use the Online Readers' Comment Form at www.ibm.com/software/awdtools/rcf/. Be sure to include the name of the document, the publication number of the document, the version of Debug Tool, and, if applicable, the specific location (for example, page number) of the text that you are commenting on.
- Send your comments by email to comments@us.ibm.com. Be sure to include the name of the book, the part number of the book, the version of Debug Tool, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

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

Changes introduced with the PTF for APAR PI37275

- The IMS Transaction Isolation Facility is added to enhance the isolation of Debug Tool users in IMS message regions. Debug Tool users can register to debug certain IMS transactions in a given IMS system. Users can also start a private IMS region, which is cloned from the currently-running environment for the selected transaction.

For more information, see “Using IMS Transaction Isolation to create a private message-processing region and select transactions to debug” on page 369.

- Support is added for deferred breakpoints for COBOL and PL/I programs. The new Debug Tool Deferred Breakpoints option in Debug Tool Utilities allows users to create and view a list of breakpoints prior to starting the debug session. It reduces the time spent on the debugging session and also the system resource usage.

For more information, see “Creating deferred breakpoints for COBOL and PL/I programs” on page 59.

- Support is added for the Debug Tool JCL Wizard. By using this new ISPF edit macro, **EQAJCL**, you can modify a JCL or procedure member and create statements to invoke Debug Tool in various environments. You can also create statements to invoke Debug Tool for the Terminal Interface Manager, Dedicated Terminal, or Remote GUI available with RD/z and PD Tools Studio.

For more information, see Appendix D, “Debug Tool JCL Wizard,” on page 457.

- Support is added to allow STEP and GO in the RD/z Debug Configuration Startup commands.

- Support is added for the code coverage and load module analyzer plug-ins.

For more information, see “Debug Tool Code Coverage Plug-in” on page 551 and “Debug Tool Load Module Analyzer Plug-in” on page 553.

- New EQAOPTS commands DOPTACBDSN, IGNOREODOLIMIT, and MAXTRANUSER.

Changes introduced with the PTF for APAR PI29800

- Support is added for debugging subtasks initiated by the ATTACH assembler macro. Multiple subtasks might be debugged concurrently, as well as the parent task.
- Delay debug mode is enhanced to support non-LE programs in certain circumstances.

Changes introduced with the PTF for APAR PI24559

- The environment specific Language Environment user exits EQADBCXT, EQADDCXT, and EQADICXT are now in sunset mode and will be removed in the next Debug Tool version. Users should move to the consolidated environment specific Language Environment user exit EQAD3CXT.

Debug Tool Utilities option 'B Delay Debug Profile' and the DTSP Profile Manager plug-in now create an enhanced debug profile that contains a new <NM2> tag (rather than the old <PGM> tag) and will only work with the consolidated user exit.

For more information about the Language Environment user exits, see Chapter 11, “Specifying the TEST runtime options through the Language

Environment user exit," on page 105 and "Using delay debug mode to delay starting of a debug session" on page 431.

- Delay debug mode is enhanced to include C functions. In addition, a load module name can now be paired with a program name or C function in the delay debug data set.
- Terminal Interface Manager now allows the same user to log on multiple times on separate terminals. This allows multiple tasks to be debugged by the same user ID simultaneously.
- Support is added for Automonitor for C/C++.
- M/L prefix commands are enhanced to support these prefix commands for C/C++.
- The CICS DTST transaction that is used to display, scan and modify CICS storage is enhanced to support 64 bit addresses.
- Code Coverage support is added for Enterprise COBOL for z/OS Version 5.
- CALL %FA command is enhanced to support remote debug mode.
- AT GLOBAL LABEL, AT LABEL, CLEAR AT GLOBAL LABEL, CLEAR AT LABEL, LIST AT GLOBAL LABEL, LIST AT LABEL and LIST NAMES LABEL commands are enhanced to support remote debug mode.

Changes introduced with the PTF for APAR PI16543

- Support is added for CICS TS 5.2.
- The DTCN Remote Plug-in is enhanced to support the use of SYSID (the SYSIDNT of a region in a CICSplex) to select CICS tasks to debug.
- The DTCN Remote Plug-in is enhanced to support additional CICS "Application" resource types that are introduced in CICS TS 5.2.
You can use these new resource types to select a program to debug:
 - Platform
 - Application
 - Operation
 - Version
- Code Coverage support for interactive remote.
- Code Coverage support for z/OS XL C.
- SET LIST BY SUBSCRIPT ON support for COBOL for MFI. This includes QUERY LIST BY SUBSCRIPT support.
- The CLIENTID option is added to the EQAOPTS DLAYDBGXRF command. This option allows a remote Debug Tool user using the enhanced DTSP Plug-in to identify a specific DB/2 client ID, and to only trap DB/2 stored procedures executed using that client ID.

Changes introduced with the PTF for APAR PI06312

- The REPOSITORY option is added to the EQAOPTS DLAYDBGXRF command. This option instructs Debug Tool to communicate with the Terminal Interface Manager to determine whether a user requests to debug an IMS transaction or DB/2 stored procedure associated with a generic user ID. This is an alternative to using the cross reference table data set.
- The Swap IMS Transaction Class and Run Transaction utility is enhanced to allow the user to manipulate the TEST option that is used for the debug message region. This allows the user to supply commands or preference files, and to direct the Debug Tool session to the remote user interface or to an alternate Terminal Interface Manager user ID.

- For CICS, provide protection of storage that was GETMAINED in the current task by a program that is not the active program. This support is enabled via INITPARM=(EQA0CPLT='STG') in the CICS startup parameters and is only available during a remote debug session.
- <PROGRAMDSCOMPILEDATE> and <PROGRAMDSCOMPILETIME> tags are added to the XML tags for code coverage. These two tags specify the compile data and time of the program source that is contained in the program data set.
- Support is added for generating a client certificate to the remote debugger if it does not exist.
- A note is added to the Coverage Utility User's Guide and Messages about the accuracy of execution counts (frequency counts) for single statement Program Areas (PAs).

Changes introduced with Debug Tool for z/OS Version 13.1

- A method for gathering code coverage is added for the generation, viewing, and reporting of code coverage by using the Debug Tool mainframe interface (MFI) as the engine. This support is provided for applications written in Enterprise COBOL and Enterprise PL/I that are compiled with the TEST compiler option and its suboption SEPARATE. This is enabled via the new EQAOPTS CCPROGSELECTDSN, CCOUPTUTDSN, and CCOUPTUTDSNALLOC commands.
- Debug Tool is enhanced to provide the automatic start of IMS message processing program (MPP) regions and dynamic routing of transactions. This allows a developer to dynamically start an MPP region, route a transaction to that MPP region, and at the end of the transaction shutdown the MPP region created for the developer thus reducing system resources.
- To help with the ease of use of the MFI mode of Debug Tool for some users, an option is added that enables breakpoints, the current line, and the line with found text to be identified by a character indicator. This feature is enabled via a new EQAOPTS ALTDISP command.
- Debug Tool is enhanced to support the following languages and platforms:
 - Enterprise COBOL for z/OS V5.1
 - Enterprise PL/I for z/OS V4.4
 - CICS TS V5.1
 - DB2 V11
 - IMS V13
 - z/OS, V2.1
 - C/C++ V2.1
- Debug Tool is enhanced to support JCL for Batch Debugging in the DTSP plug-in. This facility is used to instrument JCL to initiate a debug session from the DTSP plug-in.
- Support is added for an IMS transaction that is associated with a generic ID. A new feature is added to the consolidated Language Environment user exit (EQAD3CXT) to search a new cross-reference table for the user ID of a user who wants to debug a IMS transaction that is started from the web and is associated with a generic ID. This enables Debug Tool to debug these transactions that use a generic ID. The user ID from the cross-reference table is used to find the user's Debug Tool user exit data set (userid.DBGTOOL.EQUAOPTS), which specifies the TEST runtime parameters and the display device address. An option is added to the Debug Tool Utilities ISPF panel, "C IMS Transaction and User ID Cross Reference Table", to allow each user to update the new cross reference table.

- Support is added for tracing load modules loaded by an application. Commands TRACE LOAD and LIST TRACE LOAD are added for Debug Tool's MFI mode. This set of commands allows the user to get a trace of load modules loaded by the application. Start the trace by issuing TRACE LOAD START. Use LIST TRACE LOAD to display the trace. The trace includes load modules known to Debug Tool at the time the TRACE LOAD START command is entered and all that are loaded while the trace is active. End the trace by issuing TRACE LOAD END. Note that when the trace is ended, all trace information is deleted.
- Support is added for terminating an idle Debug Tool session that uses the Terminal Interface Manager. Debug Tool supports a timeout option via the EQAOPTS SESSIONTIMEOUT command. This command allows the system programmer to establish a maximum wait time for debug sessions that use a dedicated terminal or the Terminal Interface Manager. If the debug session exceeds the specified time limit without any user interaction, the session will be terminated with either a QUIT or QUIT DEBUG.
- Debug Tool Coverage Utility "Create HTML Targeted Coverage Report" is enhanced to allow the user to select from a list of COBOL Program-IDs, ignore changes to non-executable code, and produce a summary of the targeted lines with selectable HTML links.
- Adds IMS information to start and stop messages generated by the EQAOPTS STARTSTOPMSG command.
- Adds EQAOPTS STARTSTOPMSGDSN command and a Debug Tool Utilities option "Non-CICS Debug Session Start and Stop Message Viewer" to collect and view Debug Tool debugger session start and stop information.
- Delay debug mode is enhanced with an EQAOPTS DLAYDBGEND command to control CONDITION trapping. In addition, an EQAOPTS DLAYDBGXRF command is added so that delay debug mode can use the "IMS Transaction and User ID Cross Reference Table". Further, NOTEST is now handled in delay debug mode.
- A confirmation message is added to Debug Tool Utilities Option 6 "Debug Tool User Exit Data Set" to indicate that the updates have been saved into the EQAUOPTS data set.
- The ON and AT OCCURRENCE commands are enhanced for Enterprise PL/I to support qualifying data.
- LIST LDD and CLEAR LDD commands are added to display and remove LDD commands known to Debug Tool. LIST CC, CC START, and CC STOP commands are added to gather and display code coverage data.
- Two EQAOPTS commands are added for remote debug mode. The EQAOPTS HOSTPORTS command specifies the specific host port number or range of host port numbers on the host for a TCP/IP connection from the host to a workstation. The EQAOPTS TCPIPDATA command provides the data set name for TCPIP.DATA via the SYSTCPD DD NAME when no GLOBALTCPIPDATA statement is configured.
- A timestamp is added to the EQAY999* messages that the Terminal Interface Manager issues if the +T trace flag is on.
- Debug Tool is enhanced to allow for using GOTO or JUMPTO command for programs that are compiled with OPT and NOEJPD suboptions of the Enterprise COBOL TEST compile option when SET WARNING setting is OFF.
- An updated DTST transaction is included to write messages to the operator log when a user changes storage. These messages are intended to provide an audit trail of DTST storage changes.
- Support is added for remote Playback through the Playback Toolbar in the Debug View.

- The EQALANGP and EQALANGX modules are moved from Debug Tool's EQAW.SEQAMOD library to Common Component's IPV.SIPVMODA library. They are now aliases of IPVLANGP and IPVLANGX respectively. This removes duplication between the two tools.
- Appendix "Quick start guide for compiling and assembling programs for use with IBM Problem Determination Tools products" in the *Debug Tool User's Guide* is removed because this has been placed in the *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* instead.

Part 1. Getting started with Debug Tool

Chapter 1. Debug Tool: overview

Debug Tool helps you test programs and examine, monitor, and control the execution of programs that are written in assembler, C, C++, COBOL, or PL/I on a z/OS system. Your applications can include other languages; Debug Tool 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 2 and Table 3 on page 4 map out the combinations of compilers and subsystems that Debug Tool supports.

You can use Debug Tool to debug your programs in batch mode, interactively in full-screen mode, or in remote debug mode.

Table 2 maps out the Debug Tool interfaces and compilers or assemblers each interface supports.

Table 2. Debug Tool interface type by compiler or assembler

Compiler or assembler	Batch mode	Full-screen mode	Remote debug mode
OS/VS COBOL, Version 1 Release 2.4 (with limitations) ¹	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
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.) 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.) ¹	X	X	
AD/Cycle COBOL/370 Version 1 Release 1	X	X	
COBOL for MVS™ & VM	X	X	X
COBOL for OS/390 & VM	X	X	X
Enterprise COBOL for z/OS and OS/390 compiled with the TEST compiler option	X	X	X
Enterprise COBOL for z/OS and OS/390 compiled with the NOTEST compiler option ¹	X	X	
Enterprise COBOL for z/OS compiled with the TEST compiler option	X	X	X
Enterprise COBOL for z/OS V3 and V4 compiled with the NOTEST compiler option ¹	X	X	X
OS PL/I Version 2 Release 1, Version 2 Release 2, and Version 2 Release 3 (with limitations)	X	X	
PL/I for MVS & VM	X	X	
Enterprise PL/I for z/OS and OS/390 compiled with the TEST compiler option	X	X	X
Enterprise PL/I for z/OS compiled with the NOTEST compiler option	X	X	
AD/Cycle C/370™ Version 1 Release 2	X	X	
C/C++ for MVS/ESA Version 3 Release 2	X	X	
C/C++ feature of OS/390 Version 1 Release 3 and earlier	X	X	
C/C++ feature of OS/390 Version 2 Release 10 and later	X	X	X
C/C++ feature of z/OS	X	X	X

Table 2. Debug Tool interface type by compiler or assembler (continued)

Compiler or assembler	Batch mode	Full-screen mode	Remote debug mode
IBM High Level Assembler (HLASM), Version 1 Release 4, Version 1 Release 5, and Version 1 Release 6	X	X	X

1. See Chapter 5, “Preparing a LangX COBOL program,” on page 71 for information about how to prepare a program of this type.

Table 3 maps out the Debug Tool interfaces and subsystems each interface supports.

Table 3. Debug Tool 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

“Debug Tool interfaces”

Related tasks

Chapter 3, “Planning your debug session,” on page 23

Chapter 20, “Using full-screen mode: overview,” on page 157

Related references

Debug Tool Reference and Messages

Debug Tool interfaces

The terms *full-screen mode*, *batch mode*, and *remote debug mode* identify the types of debugging interfaces that Debug Tool provides.

Batch mode

You can use a Debug Tool commands file to predefine a series of Debug Tool 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 Debug Tool 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 Debug Tool, including online applications running under CICS, IMS/TM, or TSO.

Full-screen mode

Debug Tool 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 Debug Tool and your program
- A Monitor window in which to monitor changes in your program

You can debug all languages supported by Debug Tool in full-screen mode.

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

Related tasks

Debug Tool Customization Guide

Full-screen mode using the Terminal Interface Manager

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 Debug Tool 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 Debug Tool, which communicates through a TCP/IP connection to a remote debugger on your Windows workstation.

Debug Tool can work with a remote debugger to provide you with the ability to debug host programs, including batch programs, through a graphical user interface (GUI) on the workstation. Debug Tool works with the following remote debuggers:

- IBM Problem Determination Tools Studio
To learn more about IBM Problem Determination Tools Studio, see “IBM Problem Determination Tools Studio” (www.ibm.com/support/docview.wss?uid=swg24033230).
- IBM Problem Determination Tools Plug-ins V13 Combined Packages
To learn more about and download IBM Problem Determination Tools Plug-ins V13 Combined Packages, see “IBM Problem Determination Tools Plug-ins V13 Combined Packages” (www.ibm.com/support/docview.wss?uid=swg24033351).
- Compiled Language Debugger component of Rational Developer for System z®
To learn how to use the Compiled Language Debugger component of Rational Developer for System z, see “Compiled language debugger” in the online help for Rational® Developer for System z (<http://publib.boulder.ibm.com/infocenter/ratdevz/v8r5/topic/com.ibm.debug.pdt.doc/topics/cbpovrvw.html>).

You can enter some Debug Tool commands through the remote debugger's Debug Console. For a list of Debug Tool commands that you can enter, see “Debug Tool commands supported in remote debug mode” in the *Debug Tool Reference and Messages*.

The IBM Debug Tool DTCN and DTSP Profile Manager plug-in can make it more convenient to work in remote debug mode. The plug-in adds 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.
- The **DTSP Profile** view, which helps you create and manage the TEST runtime options data set (EQUAUOPTS).

To learn how to install and configure these plug-ins, see Appendix K, “Installing the IBM Debug Tool plug-ins,” on page 545.

Debug Tool Utilities

Debug Tool 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.

Debug Tool 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), **Debug Tool Setup File** (option 2), and **JCL for Batch Debugging** (option 8).

Debug Tool 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.

Debug Tool Utilities: Debug Tool 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 Debug Tool 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 Debug Tool Setup File (option 2).

Debug Tool Utilities: Code Coverage

Determining code coverage can help you improve your test cases so they test your program more thoroughly. Debug Tool Utilities provides you with Debug Tool Coverage Utility, a tool to report which code statements have been run by your test cases. Using the report, you can enhance your test cases so they run code statements that were not run previously. Select Code Coverage (option 3) to use this tool.

Debug Tool Utilities: IMS TM Functions

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 Functions.

Debug Tool Utilities: Load Module Analyzer

The Debug Tool 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.

Debug Tool Utilities: Debug Tool User Exit Data Set

This function assists you in preparing a TEST runtime option data set that is used by the Debug Tool Language Environment user exit. The Debug Tool 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 Debug Tool User Exit Data Set, in Debug Tool Utilities.

Debug Tool Utilities: Other IBM Problem Determination Tools

This function provides an interface to the IBM File Manager ISPF functions. You can find these tools under option 7, called Other IBM Problem Determination Tools, in Debug Tool Utilities.

Debug Tool Utilities: JCL for Batch Debugging

Modify the JCL for a batch job so that Debug Tool 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 Debug Tool Utilities.

Debug Tool 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. Debug Tool 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.

Debug Tool Utilities: JCL to Setup File Conversion

The JCL to Setup File Conversion option is an alternative to the Debug Tool 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.

Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool Utilities.

Debug Tool 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. Debug Tool 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.

Debug Tool 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.

Debug Tool Utilities: Debug Tool Code Coverage

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

Debug Tool Utilities: Debug Tool Deferred Breakpoints

The Debug Tool 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.

Debug Tool Utilities: Debug Tool JCL Wizard

The Debug Tool JCL Wizard, an ISPF edit macro named EQAJCL, can be used to modify a JCL or procedure member and create statements to invoke Debug Tool in various environments. You can also create statements to invoke Debug Tool for the Terminal Interface Manager, Dedicated Terminal, or Remote GUI available with RD/z and PD Tools Studio.

Starting Debug Tool Utilities

Debug Tool Utilities can be started in one of the following ways:

- If an option was installed to access the Debug Tool Utilities primary options ISPF panel from an existing panel, then select that option by using instructions from the installer.
- If the Debug Tool 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 common_parameters
```

- If Debug Tool 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)' 'common_parameters'
```

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

The *common_parameters* are optional and specify any of the parameters described in Appendix E of *Debug Tool Coverage Utility User's Guide and Messages*. Multiple options are separated by blanks. Note that if you specify any of these *common_parameters*, your settings are remembered by EQASTART and become the default on subsequent starts of EQASTART when you do not specify parameters.

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

Full-screen mode is the interface that Debug Tool 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"
2. "Starting Debug Tool" on page 12
3. After you start Debug Tool, you will see the full-screen mode interface. "The Debug Tool full screen interface" on page 13 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 18
 - "Clearing a breakpoint" on page 18
 - "Displaying the value of a variable" on page 15
 - "Displaying memory through the Memory window" on page 17
 - "Changing the value of a variable" on page 17
 - "Recording and replaying statements" on page 18
4. "Stopping Debug Tool" 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.

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 3, “Planning your debug session,” on page 23 for instructions on how to choose the correct combination of compiler options and suboptions to use for your situation.

Starting Debug Tool

There are several methods to start Debug Tool in full-screen mode. Each method is designed to help you start Debug Tool for programs that are compiled with an assortment of compiler options and that run in a variety of runtime environments. Part 3, “Starting Debug Tool,” on page 115 describes each of these methods.

In this topic, we describe the simplest and most direct method to start Debug Tool 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 Debug Tool:

- Chapter 13, “Starting Debug Tool from the Debug Tool Utilities,” on page 123
- Chapter 12, “Writing the TEST run-time option string,” on page 117
- “Starting Debug Tool with CEETEST” on page 127
- “Starting Debug Tool with PLITEST” on page 134
- “Starting Debug Tool with the __ctest() function” on page 135
- “Starting Debug Tool for programs that start in Language Environment” on page 141
- Chapter 15, “Starting Debug Tool in batch mode,” on page 137
- “Starting Debug Tool for programs that start outside of Language Environment” on page 143
- “Starting Debug Tool under CICS by using DTCN” on page 148
- “Starting Debug Tool for CICS programs by using CADP” on page 149
- “Starting Debug Tool under CICS by using CEEUOPT” on page 150
- “Starting Debug Tool under CICS by using compiler directives” on page 150

- “Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 139
- “Starting Debug Tool from DB2 stored procedures” on page 153

The Debug Tool full screen interface

After you start Debug Tool, the Debug Tool 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 Debug Tool Version 13 Release 1 Mod 0
10/23/2013 04:11:41 PM
5655-Q10: Copyright IBM Corp. 1992, 2013
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 Debug Tool 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 Debug Tool finds or you can specify where to find it.

Log window

The record of your interactions with Debug Tool 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 167

“Navigating through Debug Tool windows” on page 175

“Customizing the layout of physical windows on the session panel” on page 274

Related references

“Debug Tool session panel” on page 157

MEMORY command in *Debug Tool Reference and Messages*

MONITOR command in *Debug Tool Reference and Messages*

SET AUTOMONITOR command in *Debug Tool Reference and Messages*

WINDOW SWAP command in *Debug Tool 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 188

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 G0 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 G0 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

Debug Tool Reference and Messages

Setting a breakpoint

In Debug Tool, 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 Debug Tool to display the contents of the variable `myvar` when Debug Tool 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 Debug Tool to go to label `newPlace` when it reaches line 100:

```
AT 100 GOTO newPlace ;
```

A breakpoint can contain a condition, which means that Debug Tool 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 Debug Tool 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

Debug Tool 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;
```

Debug Tool 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;
```

Debug Tool 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 *Debug Tool 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 320

“Displaying values of COBOL variables” on page 292

“Displaying and monitoring the value of a variable” on page 196

Related references

“Monitor window” on page 161

Description of the MONITOR COMMAND in *Debug Tool Reference and Messages*

Description of the SET AUTOMONITOR COMMAND in *Debug Tool 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. Debug Tool 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 27, “Customizing your full-screen session,” on page 273

“Displaying the Memory window” on page 181

“Displaying and modifying memory through the Memory window” on page 206

“Scrolling through the physical windows” on page 176

Related references

“Debug Tool session panel” on page 157

WINDOW SWAP command in *Debug Tool 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 Debug Tool, 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 321

“Assigning values to COBOL variables” on page 291

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 *Debug Tool Reference and Messages*

Description of the `ENABLE` command in *Debug Tool 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 *Debug Tool 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 4 describes the sequence in which statements are replayed when you replay them in a forward direction or a backward direction.

Table 4. 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. Debug Tool returns you to the point at which you entered the PLAYBACK START command. You can resume normal debugging. Debug Tool 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
```

Debug Tool 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 *Debug Tool Reference and Messages*

Stopping Debug Tool

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 Debug Tool screen closes.

Refer to *Debug Tool 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 *Debug Tool Reference and Messages*

Description of the QQUIT command in *Debug Tool Reference and Messages*

Part 2. Preparing your program for debugging

Chapter 3. 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 Debug Tool.
- The method or methods you can use to start Debug Tool.
- 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 5 on page 25 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 24 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 Debug Tool functionality.
 - If you are using other Problem Determination Tools, information on how to choose compiler options so that you create output that can be used by the other Problem Determination Tools.
2. Use Table 3 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 11 on page 55 to record the methods you will use to specify TEST runtime options. See “Choosing a method or methods for starting Debug Tool” on page 54 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 58 for more information.

After you have completed these tasks, use the information you collected to follow the instructions in Chapter 4, “Updating your processes so you can debug programs with Debug Tool,” on page 61.

Choosing compiler options for debugging

Compiler options affects the size of your load module and the amount of Debug Tool functionality available to you. Debug Tool 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 Debug Tool debug your program, read the following topics:

- “Understanding how hooks work and why you need them” on page 48
- “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 Debug Tool 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 33
- “Choosing TEST or DEBUG compiler suboptions for C programs” on page 39
- “Choosing DEBUG compiler suboptions for C programs” on page 39
- “Choosing TEST or NOTEST compiler suboptions for C programs” on page 41
- “Choosing DEBUG compiler suboptions for C++ programs” on page 44
- “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 5. Record the compiler options you need to use in this table. The options you use work with Debug Tool for z/OS, Version 13.1 or later.

Compiler or assembler	Compiler options you will use
OS/VS COBOL, Version 1 Release 2.4 ¹	NOTEST, SOURCE, DMAP, PMAP, VERB, XREF, NOLST, NOBATCH, NOSYDMP, NOCOUNT <i>or</i>
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 <i>or</i>
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) ¹	NOTEST, NOOPTIMIZE, SOURCE, MAP, XREF, LIST(or OFFSET) <i>or</i>
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) ¹	NOTEST, NOOPTIMIZE, SOURCE, MAP, XREF, LIST(or OFFSET) <i>or</i>
AD/Cycle COBOL/370 Version 1 Release 1	TEST (ALL, SYM) <i>or</i>
COBOL for MVS & VM	TEST (ALL, SYM) <i>or</i>
COBOL for OS/390 & VM	TEST (NONE, SYM, SEPARATE) <i>or</i>
Enterprise COBOL for z/OS and OS/390, Version 3	TEST (NONE, SYM, SEPARATE) <i>or</i>
Enterprise COBOL for z/OS Version 3 or Version 4 compiled with the NOTEST compiler option ¹	NOTEST, NOOPTIMIZE, SOURCE, MAP, XREF, LIST(or OFFSET) <i>or</i>
Enterprise COBOL for z/OS Version 4 compiled with the TEST compiler option	TEST (NOHOOK, SEPARATE, EJPD) <i>or</i>
Enterprise COBOL for z/OS Version 5 compiled with the TEST compiler option	TEST (EJPD, SOURCE) <i>or</i>
OS PL/I Version 2 Release 1, Version 2 Release 2, and Version 2 Release 3	TEST (ALL, SYM) <i>or</i>
PL/I for MVS & VM	TEST (ALL, SYM) <i>or</i>
Enterprise PL/I, Version 3.1 through Version 3.3	TEST (ALL, SYM) <i>or</i>
Enterprise PL/I, Version 3.4	TEST (ALL, NOHOOK, SYM) <i>or</i>

Table 5. Record the compiler options you need to use in this table. The options you use work with Debug Tool for z/OS, Version 13.1 or later. (continued)

Compiler or assembler	Compiler options you will use
Enterprise PL/I, Version 3.5 or later	TEST (ALL, NOHOOK, SYM, SEPARATE) <i>or</i>
Enterprise PL/I, Version 3.7	TEST (ALL, NOHOOK, SYM, SEPARATE, SOURCE) <i>or</i>
Enterprise PL/I, Version 3.8 or later	TEST (ALL, NOHOOK, SYM, SEPARATE) and LISTVIEW <i>or</i>
Enterprise PL/I, Version 4 or later	TEST (ALL, NOHOOK, SYM, SEPARATE) and LISTVIEW and GONUMBER (SEPARATE) <i>or</i>
<ul style="list-style-type: none"> • 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 <i>or</i>
<ul style="list-style-type: none"> • C feature of OS/390 Version 2 Release 6 or later • C feature of z/OS, Version 1.5 or earlier 	TEST (HOOK) <i>or</i>
C/C++ feature of z/OS, Version 1.6 or later	DEBUG (FORMAT (DWARF)) <i>or</i>
IBM High Level Assembler (HLASM), Version 1 Release 4, Version 1 Release 5, Version 1 Release 6	ADATA

1. See Chapter 5, "Preparing a LangX COBOL program," on page 71 for information on how to prepare a program of this type.

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 wanted 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 Debug Tool 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 Problem Determination Tools, review the information in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination Tools.

- Scenario B: If you are compiling with any of the following compilers, you can get the most Debug Tool 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 Debug Tool 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 Problem Determination Tools, review the information in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination Tools.

- Scenario C: To get all Debug Tool 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 Problem Determination Tools, review the information in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure that you specify all the compiler options that you need to create the files needed by all the Problem Determination Tools.

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

If you are using other Problem Determination Tools, review the topic in *IBM Problem Determination Tools for z/OS Common Component: 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 Problem Determination 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 Debug Tool's functionality by compiling with the N0TEST compiler option and generating an EQALANGX file. This requires that you debug your program in LangX COBOL mode.
- Scenario F: You can get some Debug Tool's functionality by compiling with the N0TEST compiler option. This requires that you debug your program in disassembly mode.

If you are using other Problem Determination Tools, review the topic in *IBM Problem Determination Tools for z/OS Common Component: 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 Problem Determination 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, you can get the most Debug Tool 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.
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 Debug Tool, 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 Debug Tool 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 3, “Planning your debug session,” on page 23 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 Debug Tool behavior or the availability of features, which are not described in *Enterprise COBOL for z/OS Programming Guide*:

Table 6. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on Debug Tool.

Name of compiler option or suboption	Description of the effect
NOTEST	<ul style="list-style-type: none"> • 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 Debug Tool 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. <p>However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 34, “Debugging a disassembled program,” on page 355.</p>

Table 6. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on Debug Tool. (continued)

Name of compiler option or suboption	Description of the effect
NONE and NOHOOK	<ul style="list-style-type: none"> • If you use one of the following compilers, you can use the GOTO or the JUMPTO commands when you debug a non-optimized program: <ul style="list-style-type: none"> – 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 <p>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.</p> <p>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.</p> <ul style="list-style-type: none"> • A call to CEETEST can be used at any point to start Debug Tool. • 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	<p>You can modify variables in an optimized program that was compiled with one the following compilers:</p> <ul style="list-style-type: none"> • 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 <p>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.</p> <p>LOUD</p> <p>The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.</p>

Table 6. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on Debug Tool. (continued)

Name of compiler option or suboption	Description of the effect
NOSYM	<ul style="list-style-type: none"> • 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). <p>If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler ignores SYM or NOSYM and always creates a symbol table.</p> <p>This option is not available with Enterprise COBOL for z/OS Version 5.</p>
STMT	<ul style="list-style-type: none"> • 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. • Debug Tool cannot gain control at path points unless they are also at statement boundaries. • Branching to all statements and labels using the Debug Tool command GOTO is allowed. <p>If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the STMT 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.</p> <p>This option is not available with Enterprise COBOL for z/OS Version 5.</p>
PATH	<ul style="list-style-type: none"> • Debug Tool can gain control only at path points and block entry and exit points. If you attempt to step through your program, Debug Tool 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 Debug Tool. • The Debug Tool command GOTO is valid for all statements and labels coinciding with path points. <p>If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the PATH 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.</p> <p>This option is not available with Enterprise COBOL for z/OS Version 5.</p>

Table 6. Description of the effects that the COBOL NOTEST compiler option and some of the TEST compiler suboptions have on Debug Tool. (continued)

Name of compiler option or suboption	Description of the effect
BLOCK	<ul style="list-style-type: none"> • Debug Tool gains control at entry and exit of your program, methods, and nested programs. • Debug Tool 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. • GOTO can be used to branch to statements that coincide with block entry and exit points. <p>If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the BLOCK 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.</p> <p>This option is not available with Enterprise COBOL for z/OS Version 5.</p>
ALL	<ul style="list-style-type: none"> • You can set breakpoints at all statements and path points, and step through your program. • Debug Tool can gain control of the program at all statements, path points, data processing statements, labels, and block entry and exit points, allowing you to enter Debug Tool commands. • Branching to statements and labels using the Debug Tool command GOTO is allowed. <p>If you are compiling with Enterprise COBOL for z/OS, Version 4, the compiler treats the ALL 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.</p> <p>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.</p>

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.

Debug Tool 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 Debug Tool 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 must install the PTF for APAR PM19445 on Language Environment, Version 1.10 to Version 1.12.
- You can specify any of the LISTVIEW sub-options (SOURCE, AFTERALL, AFTERCICS, AFTERMARCO, 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(AFTERMARCO) compiler option and, if you do not specify the MARCO compiler option, specify the PP(MARCO(INCONLY)) compiler option.
- If you are debugging in remote debug mode and you want to automonitor variables in INCLUDE files, specify the LISTVIEW(AFTERMARCO) compiler option and, if you do not specify the MARCO compiler option, specify the PP(MARCO(INCONLY)) compiler option.

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

- Scenario B: If you are using Enterprise PL/I for z/OS, Version 3.7, and you want to get the most Debug Tool 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, AFTERMARCO, 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,AFTERMARCO) compiler options and, if you do not specify the MARCO compiler option, specify the PP(MARCO(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,AFTERMARCO) compiler options and, if you do not specify the MARCO compiler option, specify the PP(MARCO(INCONLY)) compiler option.

If you are using other Problem Determination Tools, see topic *Enterprise PL/I Version 3.5 and Version 3.6 programs* in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination 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 Debug Tool 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 Problem Determination Tools, see topic *Enterprise PL/I Version 3.5 and Version 3.6 programs* in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination 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 Problem Determination Tools, see topic *Enterprise PL/I Version 3.4 and earlier programs* in *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination Tools.

- Scenario E: If you are using Enterprise PL/I for z/OS, Version 3.3 or earlier, and you want to get all Debug Tool functionality, use TEST(ALL,SYM).

If you are using other Problem Determination 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 Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination Tools.

- Scenario F: You can get some Debug Tool functionality by compiling with the NOTEST compiler option. This requires that you debug your program in disassembly mode.

If you are using other Problem Determination Tools, review the topic in *IBM Problem Determination Tools for z/OS Common Component: 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 Problem Determination 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 Debug Tool, 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 Debug Tool 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, Debug Tool 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 Debug Tool cannot find the listing at this location, see “Changing which file appears in the Source window” on page 166.

After you have chosen the compiler options and suboptions, see Chapter 3, “Planning your debug session,” on page 23 to determine the next task you must complete.

Table 7. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on Debug Tool.

Name of compiler option or suboption	Description of the effect
NOTEST	<p>Some behaviors or features change when you debug a PL/I program compiled with the NOTEST compiler option. The following list describes these changes:</p> <ul style="list-style-type: none"> • 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 Debug Tool 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. <p>However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 34, “Debugging a disassembled program,” on page 355.</p>

Table 7. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on Debug Tool. (continued)

Name of compiler option or suboption	Description of the effect
NOHOOK	<p>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:</p> <ul style="list-style-type: none"> • For Debug Tool 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 Debug Tool stops.
NONE	<p>When you compile a PL/I program with the NONE suboption of the TEST compiler option, you can start Debug Tool at any point in your program by writing a call to PLITEST or CEETEST in your program.</p>
SYM	<p>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:</p> <ul style="list-style-type: none"> • 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.
NOSYM	<p>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:</p> <ul style="list-style-type: none"> • 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).

Table 7. Description of the effects that the PL/I NOTEST compiler option and the TEST compiler suboptions have on Debug Tool. (continued)

Name of compiler option or suboption	Description of the effect
BLOCK	<p>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:</p> <ul style="list-style-type: none"> • Enables Debug Tool 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 Debug Tool 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	<p>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:</p> <ul style="list-style-type: none"> • You can set breakpoints at all statements and step through your program. • Debug Tool 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 Debug Tool command GOTO is allowed.
ALL	<p>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:</p> <ul style="list-style-type: none"> • You can set breakpoints at all statements and path points, and STEP through your program. • Debug Tool can gain control of the program at all statements, path points, labels, and block entry and exit points, allowing you to enter Debug Tool commands. • Enables branching to statements and labels using the Debug Tool 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. Debug Tool 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. Debug Tool 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. Debug Tool needs only the .mdbg 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.

Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool functionality by compiling with the NODEBUG compiler option. This requires that you debug your program in disassembly mode.

For all scenarios, if you are using other Problem Determination Tools, see topic *z/OS XL C and C++ programs in IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination 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 Debug Tool to search for a .dbg file in a .mdbg file)¹.
 - 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.
 - If you use only .dbg files, verify that the .dbg files are non-temporary files and specify NO for the EQAOPTS MDBG command².
 - For scenario C, do the following steps:
 - a. If you are running on z/OS Version 1.6 or Version 1.7, verify that Language Environment PTF for APAR PK12833 is installed.
 - b. 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.
 - c. After you start Debug Tool, 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 Debug Tool requires access to a permanent data set for the source of the program you are debugging.
 - Debug Tool 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.

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.

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.

- 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.

Debug Tool 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 Debug Tool functionality but have a larger program size (compared to using DEBUG(FORMAT(DWARF))), use TEST(ALL,HOOK,SYMBOL).
 - Scenario B: You can get some Debug Tool 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 Problem Determination Tools, see topic *z/OS XL C and C++ programs in IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination Tools.

2. For scenario B, do the following steps:
 - a. If you are running on z/OS Version 1.6 or Version 1.7, verify that Language Environment PTF for APAR PK12833 is installed.
 - b. 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.
 - c. After you start Debug Tool, 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 Debug Tool 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, Debug Tool 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 8. Description of the effects that the C NOTEST compiler option and the TEST compiler suboptions have on Debug Tool.

Name of compiler option or suboption	Description of the effect
NOTEST	<p>The following list explains the effect the NOTEST compiler option will have on how Debug Tool behaves or the availability of features, which are not described in <i>z/OS XL C/C++ User's Guide</i>:</p> <ul style="list-style-type: none"> • 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 Debug Tool command G0T0. <p>However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 34, "Debugging a disassembled program," on page 355.</p>
TEST	<p>The following list explains the effect some of the suboptions of the TEST compiler option will have on how Debug Tool behaves or the availability of features, which are not described in <i>z/OS XL C/C++ User's Guide</i>:</p> <ul style="list-style-type: none"> • 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	<p>The following list explains the effect the NOSYM suboption of the TEST compiler option will have on how Debug Tool behaves or the availability of features, which are not described in <i>z/OS XL C/C++ User's Guide</i>.</p> <ul style="list-style-type: none"> • 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 G0T0 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.
- The z/OS 2.1 Language Environment with the PTF for APAR PI12415 applied must be available on the target system where the C programs are executed.
- 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. Debug Tool 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. Debug Tool 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. Debug Tool needs only the .mdbg 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.

Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool functionality by compiling with the NODEBUG compiler option. This requires that you debug your program in disassembly mode.

For all scenarios, if you are using other Problem Determination Tools, see *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination 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 Debug Tool to search for a .dbg file in a .mdbg file)³.
 - 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.
 - If you use only .dbg files, verify that the .dbg files are non-temporary files and specify NO for the EQAOPTS MDBG command⁴.
 - For scenario C, do the following steps:
 - a. If you are running on z/OS Version 1.6 or Version 1.7, verify that Language Environment PTF for APAR PK12833 is installed.
 - b. 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
 - c. After you start Debug Tool, 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.
 -

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.

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.

When you compile a program, do not associate SYSIN with an in-stream data set (for example //SYSIN DD *) because Debug Tool requires access to a permanent data set for the source of the program you are debugging.

- Debug Tool 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 can not call user defined functions from the command line.

After you have chosen the compiler options and suboptions, see Chapter 3, “Planning your debug session,” on page 23 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.

Debug Tool 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 Debug Tool functionality but have a larger program size (compared to using DEBUG(FORMAT(DWARF))), use TEST.
 - Scenario B: You can get some Debug Tool 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 Problem Determination Tools, see *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the compiler options you need to create the files needed by all the Problem Determination 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 Debug Tool 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, Debug Tool 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 3, “Planning your debug session,” on page 23 to determine the next task you must complete.

Table 9. Description of the effects that the C++ NOTEST and TEST compiler option have on Debug Tool.

Name of compiler option or suboption	Description of the effect
NOTEST	<p>The following list explains the effect of the NOTEST compiler has on Debug Tool behavior, which are not described in <i>z/OS XL C/C++ User's Guide</i>:</p> <ul style="list-style-type: none"> • 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 Debug Tool command G0T0. <p>However, you can still debug your program using the disassembly view. To learn how to use the disassembly view, see Chapter 34, “Debugging a disassembled program,” on page 355.</p>
TEST	<p>The following list explains the effect the TEST compiler has on Debug Tool behavior, which are not described in <i>z/OS XL C/C++ User's Guide</i>:</p> <ul style="list-style-type: none"> • 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 Debug Tool while debugging programs compiled with any of the following compilers:

- any COBOL compiler supported by Debug Tool
- any PL/I compiler supported by Debug Tool

- any C/C++ compiler supported by Debug Tool

When you compile with one of 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 Debug Tool
- any PL/I compiler supported by Debug Tool
- any C/C++ compiler supported by Debug Tool

When you start Debug Tool, 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` Debug Tool command. Note that the `SET DYNDEBUG ON` Debug Tool command must be issued before you enter the `STEP` or `GO` command. If the Dynamic Debug facility is not active, Debug Tool 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. Debug Tool 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 assemblers:

- 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. Debug Tool 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 have all of their debug information (including the symbol table) stored in a `NLOAD` 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 Debug Tool on a 3270 device, choose full-screen mode and one of the following terminal modes:

- Single terminal mode: The application program and Debug Tool 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: Debug Tool 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 Debug Tool 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 Debug Tool. 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 Debug Tool.
- Separate terminal mode (formerly called *Dual Terminal Mode*): Debug Tool 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 Debug Tool.

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 *Debug Tool 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. Debug Tool 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 *Debug Tool 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, Debug Tool 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 (Debug Tool 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, Debug Tool 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 *Debug Tool Customization Guide*.

Table 10. How different combinations of RACF access and the EQAOPTS BROWSE command activate or deactivate browse mode.

Status of RACF access	Setting of the EQAOPTS BROWSE command		
	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 Debug Tool	Cannot use Debug Tool	Cannot use Debug Tool
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 Debug Tool

Table 11 on page 55 indicates that there are several different methods to start Debug Tool 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 11. Methods for specifying the TEST runtime options and the subsystems that support these methods.

	TSO	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
Use the EQADBCXT user exit routine		X							X	X
Use the EQADICXT user exit routine								X		X
Use the EQADDCXT user exit routine						X	X ⁵			
Use the DFSBXITA user exit								X	X	X
Use the CADP transaction				X						
Use the DTCN transaction				X						
Use the DB2 catalog						X ²	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 CEEEROPT	X	X	X	X ¹	X ¹	X ^{1,2}		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 ³										
Through the EQANMDBG program ⁴	X ⁴	X ⁴						X ³	X ⁴	X ⁴
Use the EQAD3CXT user exit routine		X				X	X	X	X	X

Note:

1. You cannot use CEEEROPT to specify TEST runtime options.
2. The DB2 catalog method always takes precedence over CEEUOPT.
3. This method is only for non-Language Environment assembler programs.
4. This method is only for non-Language Environment programs.
5. This method is only for DB2 stored procedures invoked with the call_sub function.
6. EQAD3CXT now also supports DB2 stored procedures (PROGRAM TYPE=SUB) if you set the RRTIN_SW flag as x'01'.

For each subsystem, Table 11 on page 55 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

1. 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.
2. 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.
3. 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

1. 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.
2. 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.
3. 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

1. Specify the TEST runtime options by setting the _CEE_RUNOPTS environment variable.
2. 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

1. Specify the TEST runtime options using either the DTCN or CADP transaction to create and store a profile that contains the TEST runtime options.
2. 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

1. 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.
2. Specify the TEST runtime options using the parameters on the RUN statement option if you have a small number of runtime options.
3. 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

1. Specify the TEST runtime options using the Language Environment EQADDCXT or 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.
2. 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 invoked with the call_sub function, specify the TEST runtime options using the Language Environment EQADDCXT or 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 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

1. Specify the TEST runtime options using the Language Environment EQADICXT or EQAD3CXT user exit routine.
2. If your program is a non-Language Environment program, issue the EQASET transaction to setup your debugging preference.
3. If the EQADICXT or EQAD3CXT user exit routine is not available at your site, specify the TEST runtime options using the DFSBXITA user exit routine.
4. If the EQADICXT, EQAD3CXT, or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
5. If none of the previous options are 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

1. 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.
2. Specify the TEST runtime options using the EQADBCXT or EQAD3CXT user exit routine.
3. If your program is a non-Language Environment program, use the EQANMDBG program to start your debugging session.

4. If the EQADBCXT or 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.
5. If the EQADBCXT, EQAD3CXT, or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
6. If none of the previous options are 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

1. 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.
2. Specify the TEST runtime options using the EQADICXT or EQAD3CXT user exit routine.
3. If your program is a non-Language Environment program, use the EQANMDBG program to start your debugging session.
4. If the EQADICXT or EQAD3CXT user exit routine is not available at your site, specify the TEST runtime options using the EQADBCXT user exit routine.
5. If the EQADBCXT user exit routine is not available at your site, specify the TEST runtime options using the DFSBXITA user exit routine.
6. If the EQADICXT, EQAD3CXT, EQADBCXT, or DFSBXITA user exit routines are not available at your site, specify the TEST runtime options using CEEUOPT or CEEROPT.
7. If none of the previous options are 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 Debug Tool, see Chapter 3, “Planning your debug session,” on page 23 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, “Debug Tool Load Module Analyzer,” on page 535.

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 Debug Tool Utilities.
4. Debug the object module by using Debug Tool.

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 Debug Tool 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 Debug Tool 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 Debug Tool command format using the View breakpoints option in the Debug Tool 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 12. 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 4. Updating your processes so you can debug programs with Debug Tool

After you have completed the tasks in Chapter 3, “Planning your debug session,” on page 23, 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 Debug Tool needs to debug your programs are available.
- Your libraries or security systems so that you have access to the files that Debug Tool needs to debug your programs. For example, if you have RACF security measures, you might need to update them so that Debug Tool 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”
- “Update your library and promotion process” on page 66
- “Make the modifications necessary to implement your preferred method of starting Debug Tool” on page 67

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 3, “Planning your debug session,” on page 23. If you are familiar with managing JCL and with your site's compilation or assembly process, see “Compiling your program without using Debug Tool Utilities” for instructions on the specific changes you need to make. If your site uses Debug Tool Utilities to manage these processes, see “Compiling your program by using Debug Tool Utilities” on page 63 for instructions on how to use the **Program Preparation** option to update these processes.

Compiling your program without using Debug Tool 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 5 on page 25.
For each compiler, there might be additional updates you might need to make so that Debug Tool starts. The following list describes these updates:
 - If you are compiling an Enterprise PL/I program on an HFS file system, see “Compiling a Enterprise PL/I program on an HFS file system” on page 64.
 - If you are compiling a C program on an HFS file system, see “Compiling a C program on an HFS file system” on page 65.
 - If you are compiling a C program with c89 or c++, see “Compiling your C program with c89 or c++” on page 64.

- If you are compiling a C++ program on an HFS file system, see “Compiling a C++ program on an HFS file system” on page 66.
- Specify the statements to save the files that Debug Tool needs. Table 13 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.
- If you are using other Problem Determination Tools, review the topics in the chapter *Quick start guide for compiling and assembling programs for use with IBM Problem Determination Tools products of IBM Problem Determination Tools for z/OS Common Component: 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 Debug Tool to search for a .dbg file in a .mdbg file)⁵, 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 dbgl 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 72.
- 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 76.
- For DB2 programs, specify the correct DB2 preprocessor and coprocessor, as described in “Processing SQL statements” on page 79.

Table 13. 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 ⁶
	NOTEST	listing ⁶
LangX COBOL		
	“Compiling your OS/VS COBOL program” on page 71 “Compiling your VS COBOL II program” on page 72 “Compiling your Enterprise COBOL program” on page 72	EQALANGX
	any other	listing file containing pseudo-assembler code
PL/I		
	SEPARATE	separate debug file

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.

6. It is except for Enterprise COBOL for z/OS Version 5.

Table 13. 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
	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
	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 66.

Compiling your program by using Debug Tool Utilities

Debug Tool Utilities provides several utilities that can help you compile your programs and start Debug Tool. 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 Debug Tool Utilities to prepare your program and start Debug Tool, read Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 451, which describes how to prepare a sample program and start Debug Tool by using Debug Tool Utilities. After you read the sample and understand how to use Debug Tool Utilities, do the following steps:

1. Start Debug Tool 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 5 on page 25, 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 Debug Tool Utilities panel.

After you complete this task, see "Update your library and promotion process" on page 66.

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

If you are compiling and launching Enterprise PL/I programs on an HFS 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, Debug Tool 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, Debug Tool 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/'
```

Debug Tool 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 Debug Tool Utilities" on page 61 or "Compiling your program by using Debug Tool Utilities" on page 63.
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. Debug Tool 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*. Debug Tool needs access to the .mdbg file to debug your program.

Compiling a C program on an HFS file system

If you are compiling and launching programs on an HFS 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, Debug Tool 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, Debug Tool 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/'

Debug Tool 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*. Debug Tool needs access to the .mdbg file to debug your program.

Compiling a C++ program on an HFS file system

If you are compiling and launching programs on an HFS 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, Debug Tool 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, Debug Tool 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/'
```

Debug Tool 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*. Debug Tool 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 Debug Tool 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 Debug Tool.

If you are using other Problem Determination Tools, review the topics in the chapter *Quick start guide for compiling and assembling programs for use with IBM Problem Determination Tools products of IBM Problem Determination Tools for z/OS Common Component: 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 Problem Determination 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 *Debug Tool Reference and Messages* or the *Debug Tool Customization Guide*.

Make the modifications necessary to implement your preferred method of starting Debug Tool

In this topic, you will use the information you gathered after completing 2 in Chapter 3, “Planning your debug session,” on page 23 and “Choosing a method or methods for starting Debug Tool” on page 54 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 14 to record the string you want to use for each type of program you are debugging.

Table 14. Record the TEST runtime options strings you need for your site

	Test runtime options string (for example, TEST(ALL,,MFI %SYSTEM01.TRMLU001:))
TSO	
JES batch	
UNIX System Services	
CICS	
DB2	
DB2 stored procedures (PROGRAM TYPE=MAIN)	
DB2 stored procedures (PROGRAM TYPE=SUB)	
IMS TM	
IMS batch	

Table 14. Record the TEST runtime options strings you need for your site (continued)

	Test runtime options string (for example, TEST(ALL,,MFI %SYSTEM01.TRMLU001:))
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 *Debug Tool Reference and Messages*
- Chapter 12, “Writing the TEST run-time option string,” on page 117

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 11 on page 55, review the following list, which directs you to the instructions on where and how to save the TEST runtime options strings:

Through the EQADBCXT, EQADICXT, EQADDCXT, or EQAD3CXT user exit routines

See Chapter 11, “Specifying the TEST runtime options through the Language Environment user exit,” on page 105.

Through the DFSBXITA user exit routine

See “Setting up the DFSBXITA user exit routine” on page 102.

Using the CADP transaction

See “Creating and storing debugging profiles with CADP” on page 99.

Using the DTCN transaction

See “Creating and storing a DTCN profile” on page 88.

Using the DB2 catalog

See Chapter 8, “Preparing a DB2 stored procedures program,” on page 83.

By coding a call to CEETEST, __ctest(), or PLITEST

See one of the following topics:

- “Starting Debug Tool with CEETEST” on page 127
- “Starting Debug Tool with the __ctest() function” on page 135
- “Starting Debug Tool with PLITEST” on page 134

Through CEEUOPT or CEEROPT

See one of the following topics:

- “Starting Debug Tool under CICS by using CEEUOPT” on page 150
- “Linking DB2 programs for debugging” on page 81
- “Starting Debug Tool under IMS by using CEEUOPT or CEEROPT” on page 101

Using the CEEOPTS DD statement in JCL or CEEOPTS allocation in TSO

Use the **JCL for Batch Debugging** option in Debug Tool 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 Debug Tool” on page 12.

Through the EQASET transaction

See “Running the EQASET transaction for non-Language Environment IMS MPPs” on page 373.

Through the EQANDBG program

See “Starting Debug Tool for programs that start outside of Language Environment” on page 143.

Chapter 5. Preparing a LangX COBOL program

This chapter describes how to prepare a LangX COBOL program that you can debug with Debug Tool.

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 Debug Tool 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 Problem Determination Tools (for example, Application Performance Analyzer), you might need to specify additional compiler options. To understand how the Problem Determination Tools work together, see chapter *Quick start guide for compiling and assembling programs for use with IBM Problem Determination Tools products of IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide*. To learn which additional compiler options you might need to specify, see *IBM Problem Determination Tools for z/OS Common Component: 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 Problem Determination Tools (for example, Application Performance Analyzer), you might need to specify additional compiler options. To understand how the Problem Determination Tools work together, see chapter *Quick start guide for compiling and assembling programs for use with IBM Problem Determination Tools products of IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide*. To learn which additional compiler options you might need to specify, see *IBM Problem Determination Tools for z/OS Common Component: 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

Use the EQALANGX program to create the EQALANGX file. The EQALANGX program is an alias of IPVLANGX, which is shipped as part of the Common Component. 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 *Problem Determination Tools Common Component Customization Guide and User's 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 *Problem Determination Tools Common Component Customization Guide and User's 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 Common Component load modules. If the Common Component 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).

Debug Tool 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 Debug Tool.

Chapter 6. Preparing an assembler program

This chapter describes how to prepare an assembler program that you can debug with Debug Tool's full capabilities. To prepare an assembler program, you must do the following steps:

1. Assemble your program with the proper options.
2. Create the EQALANGX file.
3. Link-edit your program.

If you use Debug Tool 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 Debug Tool commands. There are three differences between debugging an assembler program and debugging programs written in other programming languages supported by Debug Tool:

- After you assemble your program, you must create a debug information file, also called the EQALANGX file. Debug Tool uses this file to obtain information about your assembler program.
- Debug Tool assumes all compile units are written in some high-level language (HLL). You must inform Debug Tool that a compile unit is an assembler compile unit and instruct Debug Tool 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. Debug Tool provides assembler-like language elements you can use to write expressions for Debug Tool commands that require an expression. See *Debug Tool 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."
2. "Creating the EQALANGX file for an assembler program" on page 76.

"Assembling your program and creating EQALANGX" on page 77 describes how to prepare an assembler program by using Debug Tool Utilities.

Assembling your program

If you assemble your program without using Debug Tool 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 Debug Tool.

If you are using other Problem Determination Tools, see *IBM Problem Determination Tools for z/OS Common Component: Customization Guide and User Guide* to make sure you specify all the assembler options you need to create the files needed by all the Problem Determination Tools.

Creating the EQALANGX file for an assembler program

Use the EQALANGX program to create the EQALANGX file. The EQALANGX program is an alias of IPVLANGX, which is shipped as part of the Common Component. 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 *Problem Determination Tools Common Component Customization Guide and User's Guide*.

To create the EQALANGX files without using Debug Tool 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 *Problem Determination Tools Common Component Customization Guide and User's 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 Common Component load modules. If the Common Component 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).

Debug Tool 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 Debug Tool to the member containing the EQALANGX data.

Debug Tool does not support debugging of Private Code (unnamed CSECT). The EQALANGX will issue error messages if an unnamed CSECT is detected in your assembler program.

Assembling your program and creating EQALANGX

You can assemble your program and create the EQALANGX file at the same time by using Debug Tool Utilities. Do the following:

1. Start Debug Tool Utilities. The Debug Tool Utilities panel is displayed.
2. Select option 1, "Program Preparation" . The Debug Tool Program Preparation panel is displayed.
3. Select option 5, "Assemble". The Debug Tool 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 exist, 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 Debug Tool Utilities by doing the following:

1. From the Debug Tool Program Preparation panel, select option L, "Link Edit". The Debug Tool 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 Debug Tool.

Restrictions for link-editing your assembler program

Debug Tool 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 7. Preparing a DB2 program

You do not need to use any special coding techniques to debug DB2 programs with Debug Tool.

The following sections describe the tasks you need to do to prepare a DB2 program for debugging:

1. "Processing SQL statements."
2. "Linking DB2 programs for debugging" on page 81.
3. "Binding DB2 programs for debugging" on page 82.

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 Debug Tool.

- 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:
<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • Use the DB2 precompiler. Save the program source files generated by the DB2 precompiler, which Debug Tool uses to debug your program. Then compile your program as described in the appropriate section for your programming language. • Use the PP(SQL:(<i>option,...</i>)) 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:
 - 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 33, 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 Debug Tool 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 Debug Tool 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 Debug Tool.

To enhance the performance of Debug Tool, 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 Debug Tool. 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,INSPREF),
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 Debug Tool.

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
CEEUOPT 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 (DSNxxxx).

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

Related tasks

Chapter 14, "Starting Debug Tool from a program," on page 127

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 Debug Tool.

Chapter 8. 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 Debug Tool. Debug Tool 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. Debug Tool 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. Debug Tool 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 *Debug Tool 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. For stored procedures of program type SUB that are *not* invoked by the call_sub function, verify that when your system programmer or DB2 system administrator defines the WLM address space, the value for NUMTCB is set to 1. NUMTCB specifies the maximum number of Task Control Blocks (TCBs) that can run concurrently in a WLM address space. If the stored procedure might run in a TCB other than the one it was started in, you will not be able to debug that stored procedure. Setting the value of NUMTCB to 1 ensures that the stored procedure is not run in a different TCB.
4. 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 that are *not* invoked by the call_sub function, determine if other users might run the stored procedure while you are debugging it. If other users might run the stored procedure, you can not debug it.

- For stored procedures of program type SUB that *are* invoked by the call_sub function, review the following options:
 - If you plan to specify the TEST runtime options through the Language Environment EQADDCXT 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.
- 5. 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.
- 6. 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 EQADDCXT or EQAD3CXT exit routine, or through the DB2 catalog. If you use both methods, the Language Environment EQADDCXT or EQAD3CXT exit routine take precedence over the DB2 catalog.
 - For stored procedures of program type SUB that *are* invoked by the call_sub function, you can specify the TEST runtime option either through the Language Environment EQADDCXT or EQAD3CXT exit routine or through the DB2 catalog. If you choose to use the Language Environment EQADDCXT or EQAD3CXT exit routine, you must specify the NOTEST runtime option for the RUN OPTIONS parameter when you define the stored procedure.
 - For stored procedures of program type SUB that are *not* invoked by the call_sub function, you can specify the TEST runtime option through the DB2 catalog or from within a program by coding a call to CEETEST, __ctest(), or PLITEST.
- 7. To specify the TEST runtime options through the Language Environment EQADDCXT or EQAD3CXT exit routine, prepare a copy of the EQADDCXT or EQAD3CXT user exit as described in Chapter 11, “Specifying the TEST runtime options through the Language Environment user exit,” on page 105.

Remember that if you want to debug an *existing* stored procedure of program type SUB that is invoked by the call_sub function, 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';
```

- 8. 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(,,TCP&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 9. 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 33
 - “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 6, “Preparing an assembler program,” on page 75
 - Chapter 5, “Preparing a LangX COBOL program,” on page 71
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.”
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 88.
 - 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 99 for more information about using CADP.

Link-editing EQADCCXT into your program

Debug Tool 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
 - Running with Language Environment Version 1 Release 6 with the PTF for APAR PK03093 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 Debug Tool 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.”

Creating and storing a DTCN profile

You can create and store DTCN profiles in the following manner:

- By using the DTCN transaction. The rest of the information in these topics describe how to do this.
- By using a plug-in for remote users. See Appendix K, “Installing the IBM Debug Tool plug-ins,” on page 545.

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.

Debug Tool determines which storage method is used based on the presence of a debugging profile VSAM file. If Debug Tool 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 *Debug Tool 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 **Debug Tool 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, Debug Tool CICS Control - Primary Menu, shown below.

```

DTCN                      Debug Tool 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
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 Debug Tool, 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 Debug Tool, 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 90. 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 92.
 - 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 92.

- Place the cursor next to the field and press PF1 to display the online help.
- 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                Debug Tool CICS Control - Menu 2                S07CICPD

Select Debug Tool 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 Debug Tool, 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. If you want to change the settings on this panel, continue with this step.

- Press PF3 to return to the main DTCN panel.
- 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                Debug Tool 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 92.
 - 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 Debug Tool.

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”

Related references

“Description of fields on the DTCN Primary Menu screen” on page 92
Description of the DTCD transaction in *Debug Tool 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, Debug Tool CICS Control - Primary Menu.
2. Press PF7. The Debug Tool CICS Control - All Sessions screen displays, shown below.

```

DTCN                Debug Tool 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 Debug Tool 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 CEEOPT 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 88

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 *Debug Tool Customization Guide* or *Debug Tool Reference and Messages*.

Transaction Id

Specify the CICS transaction to debug. If you specify a transaction ID without any other resource, Debug Tool 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 Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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 Debug Tool 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 nested CU or a subprogram within a composite load module, invoked by a static or dynamic CALL.
- Any Enterprise PL/I for z/OS Version 3 Release 4 CU (with the PTF for APAR PK03264 applied), or later, running with Language Environment Version 1 Release 6 (with the PTF for APAR PK03093 applied), or later, even if it is a nested CU or a subprogram within a composite load module, invoked as a static or 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 "Debug Tool CICS Control - Menu 2" to PROMPT, Debug Tool initializes for the first program invoked.
- If you migrate from a version of Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool stops each time you enter LoadMod::>CU.
- If Debug Tool is already running and it cannot find the separate debug file, then Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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. Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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 Debug Tool initializes on a 3270 type of terminal.
- TCP** Indicates that you want to interact with Debug Tool from your workstation using TCP/IP and a remote debugger.

Port Number

Specifies the TCP/IP port number that is listening for debug sessions on your workstation. By default, the following products use port 8001:

- IBM Problem Determination Tools Studio
- IBM Problem Determination Tools Plug-ins V13 Combined Packages
- Compiled Language Debugger component of Rational Developer for System z

Display Id

Identifies the target destination for Debug Tool.

If you entered **TCP** 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 Debug Tool on a 3270 terminal:

- Single terminal mode: Debug Tool 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: Debug Tool 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: Debug Tool 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 Debug Tool assumes control during the initialization of your application.

Test Level

ALL/ERROR/NONE specifies what conditions need to be met for Debug Tool 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, Debug Tool does the following tasks to find a commands file:

1. Debug Tool constructs the name of a data set from the naming pattern specified in the command.
2. Debug Tool 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 Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool Reference and Messages*.

Prompt Level

Specifies whether Debug Tool 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, Debug Tool does the following tasks to find a preferences file:

1. Debug Tool constructs the name of a data set from the naming pattern specified in the command.
2. Debug Tool 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 Debug Tool 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 *Debug Tool Customization Guide* or *Debug Tool 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

Specifies 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 Debug Tool to include the debugging of URMs as part of the debug session. Choose from the following options:

YES Debug Tool debugs URMs which match normal Debug Tool debugging criteria.

NO Debug Tool excludes URMs from 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 9, "Preparing a CICS program," on page 87.
- Verified that your site uses CADP and that all the tasks required to customize Debug Tool so that it can debug CICS programs described in *Debug Tool Customization Guide* are completed. In particular, verify that the DEBUGTOOL system initialization parameter is set to YES so that Debug Tool 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 Debug Tool for non-Language Environment programs under CICS

You can start Debug Tool 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 Debug Tool 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

Debug Tool global user exits before you start the programs. Debug Tool provides the following global user exits to help you debug non-Language Environment applications: XPCFTCH, XEIIIN, XEIOUT, XPCTA, and XPCHAIR. The exits can be started by using either the DTCX transaction (provided by Debug Tool), or using a PLTPI program that runs during CICS region startup. DTCXXO activates the non-Language Environment Exits for Debug Tool in CICS. DTCXXF inactivates the non-Language Environment Exits for Debug Tool in CICS.

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

Related tasks

Debug Tool Customization Guide

Passing runtime parameters to Debug Tool 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 Debug Tool
 - COUNTRY: to specify a Country Code for Debug Tool
 - TRAP: to specify whether Debug Tool is to intercept Abends

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

Related references

Debug Tool Reference and Messages

Chapter 10. Preparing an IMS program

To prepare an IMS program, do the following tasks:

1. Verify that Chapter 3, "Planning your debug session," on page 23 and Chapter 4, "Updating your processes so you can debug programs with Debug Tool," on page 61 have been completed.
2. Contact your system programmer to find out the preferred method for starting Debug Tool 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 11, "Specifying the TEST runtime options through the Language Environment user exit," on page 105 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 Debug Tool under IMS by using CEEUOPT or CEEROPT" 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 373 for instructions.
 - "Managing runtime options for IMSplex users by using Debug Tool Utilities" on page 102.

Starting Debug Tool 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, Debug Tool 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 Debug Tool Utilities

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:

1. From the main Debug Tool 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. Debug Tool 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 11. Specifying the TEST runtime options through the Language Environment user exit

Debug Tool provides a customized version of the Language Environment user exit (CEE BXITA). The user exit returns a TEST runtime option when called by the Language Environment initialization logic. Debug Tool provides user exits for three different environments. This topic is also described in *Debug Tool 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

Debug Tool 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

Debug Tool 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.

Debug Tool 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 CEE BXITA. Use this source code if you want naming patterns or message display levels that are different than the default values.

Debug Tool provides 4 customized versions of the user exit: EQADBCXT, EQADDCXT, EQADICXT, and EQAD3CXT. Table 3 shows the environments in which these user exits can be used. EQAD3CXT determines the runtime environment internally, and can be used in multiple environments. You can use the environment-specific user exit or EQAD3CXT.

Note: The environment specific Language Environment user exits (EQADBCXT, EQADDCXT and EQADICXT) are now in sunset mode and will be removed in the next Debug Tool version. Users should move to the consolidated Language Environment user exit (EQAD3CXT).

Debug Tool Utilities option 'B Delay Debug Profile' and the DTSP Profile Manager plug-in now create an enhanced debug profile that contains a new <NM2> tag (rather than the old <PGM> tag) and will only work with the consolidated user exit.

Table 15. 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: <ul style="list-style-type: none"> • type MAIN¹ • type SUB, invoked by the call_sub function⁴ 	EQADDCXT and EQAD3CXT ⁵
IMS TM ² and BTS ³	EQADICXT and EQAD3CXT
Batch	EQADBCXT and EQAD3CXT

Note:

1. EQADDCXT or EQAD3CXT is supported for DB2 version 7 or later. If DB2 RUNOPTS is specified, EQADDCXT or EQAD3CXT takes precedence over DB2 RUNOPTS.
2. For IMS TM, if you do not sign on to the IMS terminal, you might need to run the EQASET transaction with the TS01D option. For instructions on how to run the EQASET transaction, see “Debugging Language Environment IMS MPPs without issuing /SIGN ON” on page 375.
3. 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.
4. Link the user exit into a private copy of the Language Environment module CEEPIPI, not to your application program.

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 (CEEINIT, 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 109.

To prepare a program to use the Language Environment user exit, do the following tasks:

1. “Editing the source code of CEEBXITA.”
2. “Linking the CEEBXITA user exit into your application program” on page 109 or “Linking the CEEBXITA user exit into a private copy of a Language Environment runtime module” on page 110.
3. “Creating and managing the TEST runtime options data set” on page 111.

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 an SMP/E USERMOD to update the copy of the exit in the *hlq*.SEQAMOD data set. The system programmer usually implements USERMODs. Use the following sample USERMODs in *hlq*.SEQASAMP for this task:

User exit name	USERMOD name
EQADDCXT	EQAUMODC
EQADICXT	EQAUMODD
EQADBCXT	EQAUMODB
EQAD3CXT	EQAUMODK

- 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 Debug Tool substitutes with the user ID of the current user. You can also add a &PGMNAME token, which Debug Tool 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 Debug Tool substitutes the token with a value.

Table 16. 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
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 Debug Tool is started for the application program. You might also not need to make any changes to your JCL to start Debug Tool.

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 (CEEINIT, 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(EQADICXT)
//        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 17. 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: <ul style="list-style-type: none"> • type MAIN • type SUB, invoked by the call_sub function¹ 	EQADDCXT or EQAD3CXT ²	CEEPIPI
IMS TM and BTS	EQADICXT or EQAD3CXT	CEEBINIT
Batch	EQADBCXT or EQAD3CXT	CEEBINIT

Note:

1. This requires that you install the PTF for APAR PM15192 for Language Environment Version 1.10 to Version 1.12.

Edit and run sample *hlq.SEQASAMP*(EQAWLCEE or EQAWLCE3) to create these updated Language Environment runtime modules. This is typically done by the system programmer installing Debug Tool. 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 Debug Tool Language Environment user exits (EQADDCXT, EQADICXT, EQADBCXT, and EQAD3CXT) construct the name of this data set based on a naming pattern described in "Modifying the naming pattern" in the *Debug Tool 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)."
- By using Debug Tool Utilities option 6, "Debug Tool User Exit Data Set", as described in "Creating and managing the TEST runtime options data set by using Debug Tool Utilities" on page 113.
- By using the **DTSP Profile** view. To learn more about this view, see Appendix K, "Installing the IBM Debug Tool plug-ins," on page 545.

Creating and managing the TEST runtime options data set by using Terminal Interface Manager (TIM)

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 **DEBUG TOOL 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 Debug Tool 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 Debug Tool Reference and Messages manual.

EQAOPTS File

If you want Debug Tool 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 Debug Tool 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 Debug Tool Terminal Interface Manager**.
 - TCP/IP&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 Debug Tool Reference and Messages manual.
- For instructions on creating a commands file or preferences file, see the topics “Creating a commands file” on page 182 or “Creating a preferences file” on page 165.

- For instructions on creating an EQAOPTS file, see the topic “Providing EQAOPTS commands at run time” in the Debug Tool Reference and Messages manual or Debug Tool 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 Debug Tool Terminal Interface Manager** field, see “Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 139.
- For more information about the values to specify for the **Remote debug mode** field, see the online help for the Compiled Language Debugger component of Rational Developer for System z or the IBM Problem Determination Tools Plug-ins V13 Combined Packages or the IBM Problem Determination Tools Studio.

Creating and managing the TEST runtime options data set by using Debug Tool Utilities

To create the TEST runtime options data set by using Debug Tool Utilities, do the following steps:

1. Start Debug Tool Utilities and select option 6, "Debug Tool 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 Debug Tool

Chapter 12. Writing the TEST run-time 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 Debug Tool for programs that start outside of Language Environment” on page 143.

This topic describes some of the factors you should consider when you use the TEST runtime option, provides examples, and describes other runtime options you might need to specify. The syntax of the TEST runtime option is described in the topic “TEST run-time option” in *Debug Tool Reference and Messages*.

To specify how Debug Tool gains control of your application and begins a debug session, you use the TEST run-time option. The simplest form of the TEST option is TEST with no suboptions specified; however, suboptions provide you with more flexibility. There are four types of suboptions available, summarized below.

test_level

Determines what high-level language conditions raised by your program cause Debug Tool 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 Debug Tool 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.

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 Debug Tool taking control.

To enable the suboptions you specified with `NOTEST`, start Debug Tool 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 Debug Tool 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 `G0`), 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 `G0` 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 Debug Tool at different points

If Debug Tool 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 Debug Tool 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 Debug Tool, even if they were run prior to Debug Tool'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 Debug Tool, only `cu1` and `cu2` are known to Debug Tool. Debug Tool does not recognize `cux`.

The initial command string is run only once, when Debug Tool is first initialized in the process.

Commands in the preferences file are run only once, when Debug Tool 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 87

Related references

Debug Tool 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, Debug Tool 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 Debug Tool 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 run-time options

The following examples of using the TEST run-time option are provided to illustrate run-time options available for your programs. They do not illustrate complete commands. The complete syntax of the TEST run-time option can be found in the *Debug Tool Reference and Messages*.

NOTEST Debug Tool is not started at program initialization. Note that a call to CEETEST, PLITEST, or `__ctest()` causes Debug Tool to be started during the program's execution.

NOTEST(ALL,MYCMDS,*,*)

Debug Tool is not started at program initialization. Note that a call to CEETEST, PLITEST, or `__ctest()` causes Debug Tool to be started during the program's execution. After Debug Tool 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 Debug Tool 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 are (ALL, *, PROMPT, INSPREF).

TEST(ALL,*,*,*)

Debug Tool is not started initially; however, any condition or an attention in your program causes Debug Tool to be started, as does a call to CEETEST, PLITEST, or `__ctest()`. Neither a primary commands file nor preferences file is used.

TEST(NONE,,*,*)

Debug Tool is not started initially and begins by running in a "production mode", that is, with minimal effect on the processing of the program. However, Debug Tool can be started using CEETEST, PLITEST, or `__ctest()`.

TEST(ALL,test.scenario,PROMPT,prefer)

Debug Tool 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 Debug Tool 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.

TEST(ALL,,,MFI%F000:)

When running under CICS, Debug Tool displays its screens on terminal ID F000.

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 Debug Tool is started.

TEST(ALL,,,MFI%SYSTEM01.TRMLU001:)

For use in the following situation:

- 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 Debug Tool is started.

TEST(ALL,,,VTAM%USERABCD:)

For use with full-screen mode using the Terminal Interface Manager. The user accessed the Debug Tool Terminal Interface Manager with user id USERABCD.

Remote debug mode

If you are working in remote debug mode, that is, you are debugging your host application from your workstation, the following examples apply:

```
TEST(,,,TCPIP&machine.somewhere.something.com%8001:*)
TEST(,,,TCPIP&9.2404.79%8001:*)
NOTEST(,,,TCPIP&9.2411.55%8001:*)
```

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 Debug Tool. Along with the TEST option, it **must** be used if you want the Debug Tool 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 Debug Tool causes unpredictable results to occur, including the operating system cancelling your application and Debug Tool 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 13. Starting Debug Tool from the Debug Tool Utilities

The Debug Tool Setup File option (also called Debug Tool Setup Utilities or DTSU) in Debug Tool 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 Debug Tool 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”

“Editing an existing setup file”

“Saving your setup file” on page 126

“Starting your program” on page 126

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 **Debug Tool Utilities** panel, select the **Debug Tool Setup File** option.
2. In the Debug Tool 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 124

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 Debug Tool Utilities panel, select the **Debug Tool Setup File** option.
2. In the Debug Tool 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.
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”

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 Debug Tool 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 Debug Tool 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.

Debug Tool 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”

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”

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 14. Starting Debug Tool 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 Debug Tool for programs that start outside of Language Environment” on page 143.

Debug Tool 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 Debug Tool.

However, you cannot use these methods in DB2 stored procedures with the PROGRAM TYPE of SUB.

If you use these methods to start Debug Tool, you can debug non-Language Environment programs and detect non-Language Environment events only in the enclave in which Debug Tool 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 Debug Tool using these alternatives, you still need to be aware of the TEST suboptions specified using N0TEST, CEEUOPT, or other "indirect" settings.

“Example: using CEETEST to start Debug Tool from C/C++” on page 130

“Example: using CEETEST to start Debug Tool from COBOL” on page 131

“Example: using CEETEST to start Debug Tool from PL/I” on page 132

Related tasks

“Starting Debug Tool with CEETEST”

“Starting Debug Tool with PLITEST” on page 134

“Starting Debug Tool with the `__ctest()` function” on page 135

“Starting Debug Tool under CICS by using CEEUOPT” on page 150

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 117

Starting Debug Tool with CEETEST

Using CEETEST, you can start Debug Tool from within your program and send it a string of commands. If no command string is specified, or the command string is insufficient, Debug Tool 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 Debug Tool 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 Debug Tool 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++

```
▶▶ void CEETEST ( [string_of_commands] , [fc] ) ;
```

For COBOL

```
▶▶ CALL "CEETEST" USING string_of_commands , fc ;
```

For PL/I

```
▶▶ CALL CEETEST ( * [string_of_commands] , * [fc] ) ;
```

string_of_commands (input)

Halfword-length prefixed string containing a Debug Tool command list. The command string *string_of_commands* is optional.

If Debug Tool is available, the commands in the list are passed to the debugger and carried out.

If *string_of_commands* is omitted, Debug Tool prompts for commands in interactive mode.

For Debug Tool, 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool in remote debug mode, specify the TCPIP suboption of the TEST runtime option with the IP address and port number that the remote debugger is listening to:

For example, you can code the following call in your PL/I program:

```
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 Debug Tool 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 Debug Tool was started through CALL CEETEST, the GOTO command is only allowed after Debug Tool has returned control to your program via STEP or GO.

Additional notes about starting Debug Tool 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 Debug Tool from a batch process or a CICS nonterminal task; otherwise, results are unpredictable.

“Example: using CEETEST to start Debug Tool from C/C++”

“Example: using CEETEST to start Debug Tool from COBOL” on page 131

“Example: using CEETEST to start Debug Tool from PL/I” on page 132

Related tasks

“Entering multiline commands in full-screen” on page 285

Related references

z/OS Language Environment Programming Guide

Debug Tool Reference and Messages

Example: using CEETEST to start Debug Tool from C/C++

The following examples show how to use the Language Environment callable service CEETEST to start Debug Tool from C or C++ programs.

Example 1

In this example, an empty command string is passed to Debug Tool 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 Debug Tool with all defaults in effect. After it gains control, Debug Tool 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 Debug Tool commands is passed to Debug Tool and a pointer to Language Environment feedback code is returned. The call to CEETEST starts Debug Tool 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, Debug Tool 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 G0 is executed.

Note: If you include a STEP or G0 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 Debug Tool commands is passed to Debug Tool 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, Debug Tool 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, Debug Tool 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 Debug Tool from COBOL

The following examples show how to use the Language Environment callable service CEETEST to start Debug Tool from COBOL programs.

Example 1

A command string is passed to Debug Tool at its invocation and the feedback code is returned. After it gains control, Debug Tool 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 Parm.
05 AA PIC S9(4) BINARY Value 14.
05 BB PIC x(14) Value 'SET SCREEN ON;'.

CALL Debugger USING Parm. FC.

```

Example 2

A string of commands is passed to Debug Tool when it is started. After it gains control, Debug Tool 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 Debug Tool from PL/I

The following examples show how to use the Language Environment callable service CEETEST to start Debug Tool from PL/I programs.

Example 1

No command string is passed to Debug Tool at its invocation and no feedback code is returned. After it gains control, Debug Tool becomes active and prompts you for commands or reads them from a commands file.


```
CALL CEESTEST(*,*) ; /* omit arguments */
```

Example 2

A command string is passed to Debug Tool at its invocation and the feedback code is returned. After it gains control, Debug Tool becomes active and executes the command string. Barring any further interruptions, the program runs to completion, where Debug Tool prompts for further commands.

```
DCL ch char(50)
    init('AT STATEMENT 10 D0; 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 CEESTEST 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 CEESTEST(ch, fb);
```

Example 3

This example assumes that you use predefined function prototypes and macros by including CEEIBMVA, and predefined feedback code constants and macros by including CEEIBMCT.

A command string is passed to Debug Tool that sets a breakpoint on every tenth executed statement. Once a breakpoint is reached, Debug Tool displays the current location information and continues the execution. After the CEESTEST 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 CEEIBMVA;
%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 Debug Tool with PLITEST

For PL/I programs, the preferred method of Starting Debug Tool 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:

```

▶▶ CALL PLITEST [ (—character_string_expression—) ] ; ▶▶

```

character_string_expression

Specifies a list of Debug Tool commands. If necessary, this is converted to a fixed-length string.

Note:

1. If Debug Tool 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 Debug Tool for PL/I.

Example 1

No argument is passed to Debug Tool when it is started. After gaining control, Debug Tool prompts you for commands.

```
CALL PLITEST;
```

Example 2

A string of commands is passed to Debug Tool when it is started. After gaining control, Debug Tool 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 Debug Tool when it is started. After it runs the commands, Debug Tool prompts you for more commands.

```
DCL ch Char(45) Init('At Statement 23 Do; List x; End;');
```

```
CALL PLITEST(ch);
```

Starting Debug Tool with the `__ctest()` function

You can also use the C and C++ library routine `__ctest()` or `ctest()` to start Debug Tool. 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()`, Debug Tool runs the commands in that list. If you specify a null argument, Debug Tool 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. Debug Tool 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 Debug Tool 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 Debug Tool is already initialized results in a reentry that is similar to a breakpoint.

The syntax for this option is:

```
(1)  
▶▶ int __ctest (—char—*char_str_exp—);
```

Notes:

1 The syntax for `ctest()` and `__ctest()` is the same.

char_str_exp

Specifies a list of Debug Tool commands.

The following examples show how to use the `__ctest()` function for C and C++.

Example 1

A null argument is passed to Debug Tool when it is started. After it gains control, Debug Tool 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 Debug Tool when it is started. At statement 23, Debug Tool 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 `G0`.

```
__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 Debug Tool when it is started. After it runs the string of commands, Debug Tool prompts you for more commands.

```
char *ch = "at line 23 list x;";  
:  
__ctest(ch);
```

Example 4

A string of commands is passed to Debug Tool when it is started. After Debug Tool gains control, you are not prompted for commands. Debug Tool runs the commands in the command string and returns control to the program by way of the G0 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 15. Starting Debug Tool in batch mode

Choose one of the following options to start Debug Tool in batch mode:

- Follow the instructions outlined in this section. This includes modifying your JCL to include the appropriate Debug Tool data sets and TEST runtime options.
- Use the Debug Tool Setup Utility (DTSU). DTSU can generate JCL that includes the appropriate Debug Tool data sets and TEST runtime options, and can submit your batch job. For instructions on how to use DTSU, refer to Chapter 13, “Starting Debug Tool from the Debug Tool Utilities,” on page 123.

To start Debug Tool 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 Debug Tool 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 139.
- In remote debug mode. Follow the instructions in the topic “Preparing to debug” of the online help for the Compiled Language Debugger component of Rational Developer for System z or the IBM Problem Determination Tools Studio.

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 523

Chapter 28, “Entering Debug Tool commands,” on page 283

Example: JCL that runs Debug Tool 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 Debug Tool session
//*   Program EMPLRUN was previously compiled with the COBOL
//*   compiler TEST option
//* *****
//STEP1 EXEC PGM=EMPLRUN,
//        PARM='/TEST(,INSPIN,,)'           1
//*
//* Include the Debug Tool 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;

/*
/**
/** 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.INSPLUG
/**
//INSPLUG    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 Debug Tool in batch mode” on page 137 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 16. Starting Debug Tool for batch or TSO programs

This section describes how to start Debug Tool 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

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 Debug Tool Terminal Interface Manager. The Debug Tool 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 TSO user ID and password to the Terminal Interface Manager and press Enter.

Note: When you provide your user ID and password to the Terminal Interface Manager, you are not logging on TSO. You are only indicating that your user ID is to be associated with this terminal LU.

A panel similar to the following panel is then displayed on the second terminal emulator session:

```
DEBUG TOOL TERMINAL INTERFACE MANAGER

EQAY001I Terminal TRMLU001 connected for user USER1
EQAY001I Ready for Debug Tool

PF3=EXIT PF10=Edit LE options data set PF12=LOGOFF
```

The terminal is now ready to receive a Debug Tool 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 Debug Tool, the second terminal emulator session displays the panel and messages you saw in step 3. This indicates that Debug Tool can use this session again. (this will happen each time you exit from Debug Tool).
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 Debug Tool 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 Debug Tool, 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 Debug Tool for programs that start in Language Environment

Choose one of the following options to start Debug Tool 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 Debug Tool 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 13, “Starting Debug Tool from the Debug Tool Utilities,” on page 123.

To start Debug Tool 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 Debug Tool SEQAMOD library is in the load module search path.

Note: High-level qualifiers and load library names are specific to your installation. Ask the person who installed Debug Tool 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 Debug Tool 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 12, “Writing the TEST run-time option string,” on page 117
“Starting a debugging session in full-screen mode using the Terminal Interface Manager or a dedicated terminal” on page 139
“Recording your debug session in a log file” on page 183
Chapter 14, “Starting Debug Tool from a program,” on page 127

Related references

Debug Tool Reference and Messages
z/OS Language Environment Programming Guide

Example: Allocating Debug Tool load library data set

The following example CLIST fragments show how you might allocate the Debug Tool 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 Debug Tool data set is allocated.

Example: Allocating Debug Tool files

The following example illustrate how you can use the command line to allocate the preferences and log files, then start the COBOL program tstscript with the TEST run-time option:

```
ALLOCATE FILE(insppref) data set(setup.pref) REUSE
ALLOCATE FILE(inspllog) data set(session.log) REUSE
CALL 'USERID1.MYLIB(TSTSCRIPT)' '/TEST'
```

The example illustrates that the default Debug Tool 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(inspllog) DA(debug.log) REUSE
ALLOC FI(insppref) DA(debug.preferen) REUSE

CALL 'MYID.MYQUAL.LOAD(PROG1)' +
' TRAP(ON) TEST(,*,;,insppref)/'
```

All the data sets must exist before starting this CLIST.

Starting Debug Tool 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 Debug Tool program EQANMDBG to start Debug Tool.

If you need to debug a non-Language Environment program where EQANMDBG is used to start Debug Tool, and your program frees SUBPOOL 1 (which Debug Tool 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 Debug Tool 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 Debug Tool for programs that start in Language Environment” on page 141 to debug all the programs.

To start Debug Tool by using EQANMDBG, do one of the following options:

- By using the Debug Tool Setup Utility (DTSU) option 3 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 Debug Tool run-time parameters. See “Passing parameters to EQANMDBG by using only the PARM string” on page 144 for instructions.
 - Add a EQANMDBG DD statement that provides the name of the program to be debugged and any required Debug Tool run-time parameters. See “Passing parameters to EQANMDBG using only the EQANMDBG DD statement” on page 145 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 Debug Tool run-time parameters. See “Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement” on page 145 for instructions.
 - Verify that the Debug Tool SEQAMOD and SEQABMOD libraries are in the load module search path.

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

Related tasks

Chapter 13, “Starting Debug Tool from the Debug Tool Utilities,” on page 123

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 Debug Tool
 - NATLANG to specify the national language used to communicate with Debug Tool
 - NQNLESP to specify the SUBPOOL for Debug Tool to use for its storage
 - TEST to specify Debug Tool options. For example, you can use suboptions of the TEST run-time option to specify the data sets that contain Debug Tool 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 Debug Tool is to intercept abends.

You can specify these parameters in one of following ways:

- “Passing parameters to EQANMDBG by using only the PARM string”
- “Passing parameters to EQANMDBG using only the EQANMDBG DD statement” on page 145
- “Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement” on page 145

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

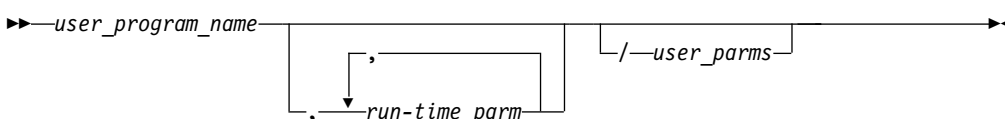
Related references

Debug Tool run-time options (*Debug Tool 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 Debug Tool 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
<pre>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12) ' ... //</pre>	<pre>//STEP1 EXEC PGM=EQANMDBG, // PARM='MYPROG,NATLANG(UEN)/ABC,X(12) ' ... //</pre>

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 Debug Tool parameters to the PARM string, you can leave the PARM string unchanged and pass all required parameters to Debug Tool 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
<pre>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)' ... //</pre>	<pre>//STEP1 EXEC PGM=EQANMDBG, // PARM='ABC,X(12)' //EQANMDBG DD * MYPROG, TEST(ALL,INSPIN,,MFI:*), NATLANG(ENU) /* ... //</pre>

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 Debug Tool 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
<pre>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)' ... //</pre>	<pre>//STEP1 EXEC PGM=EQANMDBG, // PARM='MYPROG/ABC,X(12)' //EQANMDBG DD * *,TEST(ALL,INSPIN,,MFI:*),NATLANG(ENU) /* ... //</pre>

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 Debug Tool.

Original sample JCL	Modified sample JCL
<pre>//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 ...</pre>	<pre>//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 ...</pre>

Chapter 17. Starting Debug Tool under CICS

This topic compares the different methods you can use to start Debug Tool 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 Debug Tool 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 3, “Planning your debug session,” on page 23
 - Chapter 4, “Updating your processes so you can debug programs with Debug Tool,” on page 61
 - Chapter 9, “Preparing a CICS program,” on page 87

Comparison of methods for starting Debug Tool under CICS

There are several different mechanisms available to start Debug Tool under CICS. Each mechanism has a different advantage and are listed below:

- DTCN is a full-screen CICS transaction that Debug Tool 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 Debug Tool which are defined in your CICS installation except the STACK option.

DTCN has the following advantages and differences compared to CADP:

- Provides a plug-in for remote users. See Appendix K, “Installing the IBM Debug Tool plug-ins,” on page 545.
- 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 *Debug Tool 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
- Provides a **EQAOPTS File** field. You can use this field to specify a file that contains a set of Debug Tool EQA0PTS commands for the debug session.

To learn how to set up profiles by using DTCN, see Chapter 9, “Preparing a CICS program,” on page 87.

- 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 Debug Tool every time the application is run.

This mechanism can be useful during initial testing of new code when you will want to run Debug Tool 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 Debug Tool until it encounters the directive, at which time Debug Tool is started at the precise point that you specify. With `CALL CEETEST`, you can even make the invocation of Debug Tool 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 Debug Tool under CICS by using DTCN

If a DTCN profile exists, when a CICS program starts, Debug Tool analyzes the program's resources to see if they match a profile. If Debug Tool finds a match, Debug Tool starts a debugging session for that program. If multiple profiles exist, Debug Tool selects the profile with the greatest number of resources that match the program. If two programs have an equal number of matching resources, Debug Tool selects the older profile.

Before you begin, verify that you prepared your CICS program as instructed in Chapter 9, “Preparing a CICS program,” on page 87.

To start Debug Tool 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 Debug Tool identifies a task that matches a DTCN profile, Debug Tool starts. If you chose screen control mode, press Enter on the terminal running the DTSC transaction to connect to Debug Tool.

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 Debug Tool.

Example: How Debug Tool 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 Debug Tool for CICS programs by using CADP

Before you begin, verify that you prepared your CICS program as instructed in Chapter 9, “Preparing a CICS program,” on page 87.

To start Debug Tool 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 Debug Tool identifies a task that matches a CADP profile, Debug Tool starts. If you chose screen control mode, press Enter on the terminal running the DTSC transaction to connect to Debug Tool.

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 99

Related references

CICS Supplied Transactions

Starting Debug Tool under CICS by using CEEUOPT

To request that Language Environment start Debug Tool 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 Debug Tool will be started.

Debug Tool 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 Debug Tool, 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 12, "Writing the TEST run-time option string," on page 117

Starting Debug Tool under CICS by using compiler directives

When compile-directives are processed by your program, Debug Tool 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 Debug Tool with CEETEST" on page 127

Chapter 18. Starting a full-screen debug session

You can start Debug Tool by using the Language Environment TEST run-time option in one of the following ways:

- Using the Debug Tool 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.
- For TSO programs that start in Language Environment, start your program with the TEST run-time option as described in “Starting Debug Tool for programs that start in Language Environment” on page 141.
- 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 15, “Starting Debug Tool in batch mode,” on page 137.
- For MVS batch programs that do not start in Language Environment, start the non-Language Environment Debug Tool (EQANMDBG), and pass your program name and the TEST runtime option. Specify the appropriate suboptions, as described in “Starting Debug Tool for programs that start outside of Language Environment” on page 143.
- For CICS, make sure Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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 13, “Starting Debug Tool from the Debug Tool Utilities,” on page 123
- “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 33
- “Ending a full-screen debug session” on page 210
- “Entering commands on the session panel” on page 167
- “Passing parameters to EQANMDBG” on page 143

Related references

“Debug Tool session panel” on page 157

Chapter 19. Starting Debug Tool in other environments

You can start Debug Tool to debug batch programs from DB2 stored procedures.

Starting Debug Tool from DB2 stored procedures

Before you run the stored procedure, verify that you have completed all the instructions in Chapter 8, “Preparing a DB2 stored procedures program,” on page 83.

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 Debug Tool 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 EQADDCXT or 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 3, “Planning your debug session,” on page 23

Part 4. Debugging your programs in full-screen mode

Chapter 20. Using full-screen mode: overview

The topics below describe the Debug Tool 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 Debug Tool, even if you plan to use batch or line modes later.

The following list describes the maximum screen size supported by Debug Tool 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, Debug Tool 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 18, "Starting a full-screen debug session," on page 151

"Ending a full-screen debug session" on page 210

"Entering commands on the session panel" on page 167

"Navigating through Debug Tool windows" on page 175

"Recording your debug session in a log file" on page 183

"Setting breakpoints to halt your program at a line" on page 186

"Setting breakpoints in a load module that is not loaded or in a program that is not active" on page 186

"Stepping through or running your program" on page 188

"Displaying and monitoring the value of a variable" on page 196

"Displaying error numbers for messages in the Log window" on page 209

"Displaying a list of compile units known to Debug Tool" on page 209

"Requesting an attention interrupt during interactive sessions" on page 210

Chapter 24, "Debugging a C program in full-screen mode," on page 241

Chapter 25, "Debugging a C++ program in full-screen mode," on page 251

Chapter 21, "Debugging a COBOL program in full-screen mode," on page 213

Chapter 23, "Debugging a PL/I program in full-screen mode," on page 231

Debug Tool session panel

The Debug Tool 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 Debug Tool finds or you can specify where to find it.

Log window

The record of your interactions with Debug Tool and the results of those interactions.

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 Debug Tool 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 *-----1-----2-----3-----4----- .
109 ADD 1 TO AA-NUM1 2 .
110 .
111 *-----1-----2-----3-----4----- .
112 * SINGLE DATAITEM IN A STRUCTURE - QUALIFIED .
LOG 0-----1----+----2----+----3----+----4----+----5----+----6- LINE: 40 OF 43
0040 MONITOR
0041 LIST NUM4 ;
0042 MONITOR
0043 LIST WK-LONG-FIELD-2 ; 3
```

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 274

Related references

“Session panel header”

“Monitor window” on page 161

“Source window” on page 160

“Log window” on page 162

“Memory window” on page 163

Session panel header

The first few lines of the Debug Tool session panel contain a command line and header fields that display information about the program that you are debugging.

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 18 lists all the scrolling commands.

Table 18. Scrolling commands

Command	Description
<i>n</i>	Scroll by <i>n</i> 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 <i>x</i>	Scroll to the left by <i>x</i> number of characters.
RIGHT <i>x</i>	Scroll to the right by <i>x</i> number of characters.
CURSOR	Position of the cursor.
TO <i>x</i>	Scroll to line <i>x</i> , where <i>x</i> 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 Debug Tool 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 171

“Customizing profile settings” on page 277

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-----+-----3-----+-----4-----
0001  1 NUM1                                0000000005
0002  2 NUM4                                '1111'
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
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 Debug Tool displays the values of the variables. Debug Tool extends the display to the right up to the full width of the displayable area of the Monitor window.
- 3 Monitor name area, where Debug Tool displays the names of the variables.
- 4 Monitor reference number area, where Debug Tool displays the reference number it assigned to a variable.

When you enter the MONITOR LIST, MONITOR QUERY, MONITOR DESCRIBE, and SET AUTOMONITOR commands, Debug Tool displays the output in the Monitor window. If this window is not open, Debug Tool 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 197

“Replacing a variable in the Monitor window with another variable” on page 199

“Adding variables to the Monitor window automatically” on page 200

“Scrolling through the physical windows” on page 176

Related references

“SET MONITOR command” in *Debug Tool Reference and Messages*

“QUERY command” in *Debug Tool 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 Debug Tool.

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 181

Command pop-up window

Debug Tool displays the Command pop-up window as a pop-up window over the Source, Log, and Monitor windows so that you can more easily enter long or multiline commands. Debug Tool 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, Debug Tool 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, Debug Tool 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, Debug Tool 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. Debug Tool closes the window and runs the command.

Creating a preferences file

If you have a preference as to the appearance or behavior of Debug Tool, 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 279 describes what you can specify in a preferences file and how to make Debug Tool 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 Debug Tool 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, Debug Tool 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, Debug Tool 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, Debug Tool reads that member as a commands file.

Because of the order in which Debug Tool 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 *Debug Tool Reference and Messages* or *Debug Tool 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 Debug Tool,” on page 439.

Displaying the source

Debug Tool 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 Debug Tool, if your source is not displayed, see “Changing which file appears in the Source window” on page 166 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 7, “Preparing a DB2 program,” on page 79 or Chapter 9, “Preparing a CICS program,” on page 87 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 5, “Preparing a LangX COBOL program,” on page 71

“Choosing TEST or NOTEST compiler suboptions for PL/I programs” on page 33

“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 6, “Preparing an assembler program,” on page 75

Chapter 7, “Preparing a DB2 program,” on page 79

Chapter 8, “Preparing a DB2 stored procedures program,” on page 83

Chapter 9, “Preparing a CICS program,” on page 87

Chapter 10, “Preparing an IMS program,” on page 101

Related references

Appendix B, “How does Debug Tool locate source, listing, or separate debug files?,” on page 445

Debug Tool 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 Debug Tool” on page 209 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 Debug Tool locates source, listing, and separate debug files by reading Appendix B, “How does Debug Tool locate source, listing, or separate debug files?,” on page 445.

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 Debug Tool.
- Use the Source Identification panel to direct Debug Tool 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 Debug Tool.

- 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 Debug Tool 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 Debug Tool 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 Debug Tool to search for the .dbg and the source file in a .mdbg file)⁷, 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 168 and Figure 2 on page 169.

7. 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          LOCATION: MYID.SOURCE(ICFSSCU1) :> 89
Command ==> 1                               Scroll ==> PAGE 2
MONITOR  --+---1---+---2---+---3---+---4---+---5---+---6 LINE: 1 OF 2
***** TOP OF MONITOR *****
-----+---1---+---2---+---3---+---4---
0001  1  VARBL1                10
0002  2  VARBL2                20
***** BOTTOM OF MONITOR *****
SOURCE: ICFSSCU1 - 3 --+---2---+---3---+---4---+---5---+ LINE: 81 OF 96
  81  main()                    .
  82  {                          .
  83     int VARBL1 = 10;         .
4  84     int VARBL2 = 20;         .
  85     int R = 1;              .
  86                               .
  87     printf("— IBFSSCC1 : BEGIN\n"); 5 .
  88     do {                    .
  89         VARBL1++;            .
  90         printf("INSIDE PERFORM\n"); .
  91         VARBL2 = VARBL2 - 2; .
  92         R++;                 .
LOG 6 --+---1---+---2---+---3---+---4---+---5---+---6 LINE: 7 OF 15
0007  STEP ;
0008  AT 87 ;
0009  MONITOR
0010     LIST VARBL1 ;
0011  MONITOR
0012     LIST VARBL2 ;
0013  GO ; 7
0014  STEP ;
0015  STEP ;

```

Figure 1. Debug Tool session panel displaying the Log window.

```

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. Debug Tool session panel displaying the Memory window.

Note: Figure 2 shows PF keys that were redefined. If you want to redefine your PF keys, see “Defining PF keys” on page 273.

- 1 Command line**
You can enter any valid Debug Tool 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 Debug Tool 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 Debug Tool 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"

"Issuing system commands" on page 171

"Entering prefix commands on specific lines or statements" on page 171

"Entering multiple commands in the Memory window" on page 172

"Using commands that are sensitive to the cursor position" on page 173

"Using Program Function (PF) keys to enter commands" on page 173

"Retrieving previous commands" on page 174

"Composing commands from lines in the Log and Source windows" on page 174

Related references

"Order in which Debug Tool accepts commands from the session panel"

"Initial PF key settings" on page 173

Order in which Debug Tool 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 Debug Tool command in the command field. You can also enter any TSO command by prefixing them with SYSTEM or TSO. Commands can be up to 48 SBCS characters or 23 DBCS characters in length.

If you need to enter a lengthy command, Debug Tool 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.

Debug Tool 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, Debug Tool 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 28, “Entering Debug Tool commands,” on page 283

Issuing system commands

During your Debug Tool 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. Debug Tool 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 Debug Tool 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. Debug Tool 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. Debug Tool 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 *Debug Tool Reference and Messages*

Prefix commands in *Debug Tool Reference and Messages*

Entering multiple commands in the Memory window

You can enter multiple commands and changes into the Memory window. Debug Tool 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 Debug Tool 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 273

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 273

"Using commands that are sensitive to the cursor position"

Related references

"Initial PF key settings"

Initial PF key settings

The table below shows the initial PF key settings.

PF key	Label	Definition	Use
PF1	?	?	"Getting online help for Debug Tool command syntax" on page 288
PF2	STEP	STEP	"Stepping through or running your program" on page 188

PF key	Label	Definition	Use
PF3	QUIT	QUIT	"Ending a full-screen debug session" on page 210
PF4	LIST	LIST	"Displaying a list of compile units known to Debug Tool" on page 209
PF4	LIST	LIST <i>variable_name</i>	"Displaying and monitoring the value of a variable" on page 196
PF5	FIND	IMMEDIATE FIND	"Finding a string in a window" on page 178
PF6	AT/CLEAR	AT TOGGLE CURSOR	"Setting breakpoints to halt your program at a line" on page 186
PF7	UP	IMMEDIATE UP	"Scrolling through the physical windows" on page 176
PF8	DOWN	IMMEDIATE DOWN	"Scrolling through the physical windows" on page 176
PF9	GO	GO	"Stepping through or running your program" on page 188
PF10	ZOOM	IMMEDIATE ZOOM	"Zooming a window to occupy the whole screen" on page 276
PF11	ZOOM LOG	IMMEDIATE ZOOM LOG	"Zooming a window to occupy the whole screen" on page 276
PF12	RETRIEVE	IMMEDIATE RETRIEVE	"Retrieving previous commands"

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

Related tasks

"Defining PF keys" on page 273

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"

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. Debug Tool displays the input line or lines on the command line. If the line or lines do not fit on the command line, Debug Tool 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 174

Chapter 28, “Entering Debug Tool commands,” on page 283

Related references

“COBOL command format” on page 289

“Debug Tool subset of PL/I commands” on page 307

“PL/I language statements” on page 307

“Debug Tool commands that resemble C and C++ commands” on page 319

Opening the Command pop-up window to enter long Debug Tool 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 Debug Tool command.

Debug Tool 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 Debug Tool 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 176

“Scrolling through the physical windows” on page 176

“Scrolling to a particular line number” on page 178
“Finding a string in a window” on page 178
“Changing which file appears in the Source window” on page 166
“Displaying the line at which execution halted” on page 180
“Customizing profile settings” on page 277

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 273

Switching between the Memory window and Log window

Debug Tool 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 178
“Scrolling through the physical windows”

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, Debug Tool 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 274

“Scrolling to a particular line number” on page 178

“Customizing profile settings” on page 277

“Enlarging a physical window”

“Navigating through the Memory window using the history area” on page 181

Related references

`QUERY` command in *Debug Tool Reference and Messages*

`SCROLL` command in *Debug Tool Reference and Messages*

`SET DEFAULT WINDOW` command in *Debug Tool 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 274

Related references

WINDOW command in *Debug Tool 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. Debug Tool 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 179.
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 Debug Tool search for strings?”

Related references

“Syntax of a search string” on page 179

How does Debug Tool search for strings?

The Debug Tool FIND command uses many of the same rules for beginning a search that the ISPF FIND command uses to begin its searches. Debug Tool begins a search in the first position after the cursor location.

If you reach the end, Debug Tool 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, Debug Tool 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 contain 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. Debug Tool 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 Debug Tool 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, Debug Tool searches for the string you specified only within those boundaries. For example, to specify that you want Debug Tool 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, Debug Tool 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"

FIND command in *Debug Tool Reference and Messages*

SET FIND BOUNDS command in *Debug Tool Reference and Messages*

QUERY command in *Debug Tool 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;
```

Debug Tool 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;
```

Debug Tool 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. Debug Tool 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. Debug Tool 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 176

“Switching between the Memory window and Log window” on page 176

“Displaying memory through the Memory window” on page 17

“Customizing the layout of physical windows on the session panel” on page 274

Related references

“Memory window” on page 163

“Order in which Debug Tool accepts commands from the session panel” on page 170

MEMORY command in *Debug Tool 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, Debug Tool 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. Debug Tool 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 176

“Specifying a new base address”

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, Debug Tool tries to set the base address to X'4040329E'. If you enter the G command over the second row, second column, Debug Tool 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 Debug Tool," on page 439 describes the requirements for this file and when Debug Tool 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 Debug Tool 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 *Debug Tool 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, Debug Tool does not interpret the data in columns 73 through 80 as a command. Later, if you want Debug Tool 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, Debug Tool assigns block names in a sequential manner. If your program is compiled with the DWARF compiler option, Debug Tool 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, Debug Tool reads that member as a commands file.

To learn how to specify EQAOPTS commands, see the topic "EQAOPTS commands" in the *Debug Tool Reference and Messages* or *Debug Tool Customization Guide*. To learn about what format to use for the commands file, see Appendix A, "Data sets used by Debug Tool," on page 439.

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

Related tasks

"Entering comments in Debug Tool commands" on page 287

Related references

BEGIN command in *Debug Tool Reference and Messages*

%IF command in *Debug Tool Reference and Messages*

Recording your debug session in a log file

Debug Tool 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
- Debug Tool 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”

“Saving and restoring settings, breakpoints, and monitor specifications” on page 191

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 Debug Tool users automatically create a log file. To learn how to specify EQAOPTS commands, see the topic “EQAOPTS commands” in the *Debug Tool Reference and Messages* or *Debug Tool Customization Guide*.

If you are an existing user that saves settings in a SAVESETS data set, Debug Tool does not create a new log file for you because the SAVESETS data set contains a SET LOG command. Debug Tool 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. Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool session.

Debug Tool 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 184.
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 184

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, Debug Tool 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 245

“Halting on a line in C++ only if a condition is true” on page 257

“Halting on a COBOL line only if a condition is true” on page 218

“Halting on a PL/I line only if a condition is true” on page 235

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 Debug Tool handles warnings about invalid data in comparisons

When Debug Tool processes (evaluates) a conditional expression and the data in one of the operands is invalid, the conditional expression becomes invalid. In this situation, Debug Tool stops and prompts you for a command. You have to enter the GO command to continue running your program. If you want to prevent Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool 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. Debug Tool stops and prompts you to enter a command.
3. You decide you want Debug Tool 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, Debug Tool evaluates the conditional expression WK-TEST-NUM = 10 as false and runs the G0 command. Debug Tool 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 Debug Tool starts, none of your program has run yet (including C++ constructors and static object initialization).

Debug Tool 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 Debug Tool 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 (G0).

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 3, “Planning your debug session,” on page 23

Chapter 12, “Writing the TEST run-time option string,” on page 117

Recording and replaying statements

Debug Tool 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

Debug Tool 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 Debug Tool 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 Debug Tool can record. After Debug Tool has filled all the available storage, Debug Tool 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 4⁸
- 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 191

Preparing to replay the statements that you recorded

The PLAYBACK START command notifies Debug Tool that you want to replay the statements that you recorded. This command suspends normal debugging; all breakpoints are suspended and you cannot use many Debug Tool commands. *Debug Tool 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 ENABLE command. If you are replaying in the forward direction, you reach the point where you entered the PLAYBACK START 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.

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.

8. With Enterprise COBOL for z/OS, Version 4, and the TEST compiler option the symbol tables are always generated.

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. Debug Tool 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 Debug Tool commands are unavailable. *Debug Tool 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 Debug Tool commands.

When you are replaying statements, many Debug Tool 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.

Debug Tool 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, Debug Tool 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 Debug Tool 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.DBGT00L.SAVESETS` (a sequential data set) is used to save the settings.
- `userid.DBGT00L.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 Debug Tool to use to save and restore the settings and an `INSPBPM` DD statement to indicate the data set that you want Debug Tool 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. Debug Tool uses a separate member to store the breakpoints, LDD data, and monitor specifications for each enclave. Debug Tool 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, Debug Tool saves information about your debugging session. The next time you start a debugging session, Debug Tool 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 *Debug Tool Reference and Messages* or *Debug Tool 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 Debug Tool.

- 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 Debug Tool. Your settings are saved in the corresponding data set.

The next time you start Debug Tool, 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 Debug Tool.

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 Debug Tool.
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 Debug Tool.

The next time you start Debug Tool in one of these environments, you must use the following commands, in the sequence shown, at the beginning of your Debug Tool 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 Debug Tool is activated and once when Debug Tool terminates. Therefore, unless Debug Tool 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

Debug Tool 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 *Debug Tool Reference and Messages* for more information about this command.

If Debug Tool cannot display the value of a variable in its declared data type, see “How Debug Tool handles characters that cannot be displayed in their declared data type” on page 203.

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:

$$\text{rightSide} = (\text{leftSide} * \text{leftSide}) + (\text{bottomSide} * \text{bottomSide});$$
 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. Debug Tool 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. Debug Tool 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 Debug Tool receives control or every time you enter a Debug Tool command that can affect the display, Debug Tool 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:

$$\text{rightSide} = (\text{leftSide} * \text{leftSide}) + (\text{bottomSide} * \text{bottomSide});$$
 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 Debug Tool receives control or every time you enter a Debug Tool command that can affect the display, Debug Tool 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 200

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. Debug Tool 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 Debug Tool receives control or you enter a Debug Tool command that can effect the display, Debug Tool updates the value of each variable in the Monitor window so that Debug Tool 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 206

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*. Debug Tool adds the variable to the Monitor window and displays the current value and data type of the variable.

-

```
SET AUTOMONITOR ON;
```

Debug Tool 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;
```

Debug Tool 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, Debug Tool 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. Debug Tool 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. Debug Tool 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 197) before you run each statement can be time consuming. Debug Tool 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 Debug Tool automatically adds variables to the Monitor window” on page 201.

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 Debug Tool just ran, after the statement was run.
 - SET AUTOMONITOR ON BOTH; if you want to display variables at the statement Debug Tool just ran, after the statement was run, and the current statement, before the statement is run.

As you step through your program, Debug Tool 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. Debug Tool 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 Debug Tool automatically adds variables to the Monitor window” on page 201

Related tasks

“Saving the information in the automonitor section to the log file” on page 201

Related references

Description of the SET AUTOMONITOR command in *Debug Tool Reference and Messages*.

“Example: How Debug Tool adds variables to the Monitor window automatically” on page 202

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 Debug Tool automatically adds variables to the Monitor window”

Related tasks

“Adding variables to the Monitor window automatically” on page 200

Related references

Description of the SET AUTOMONITOR command in *Debug Tool Reference and Messages*.

“Example: How Debug Tool adds variables to the Monitor window automatically” on page 202

How Debug Tool automatically adds variables to the Monitor window

When you enter the SET AUTOMONITOR ON command, Debug Tool 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, Debug Tool 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 Debug Tool runs next. The values displayed are values *before* the statement is run.

This behavior displays the value of the variables before Debug Tool runs the statement. If you want to see the value of the variables after Debug Tool runs the statement, you can enter the SET AUTOMONITOR ON PREVIOUS command. Debug Tool

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, Debug Tool 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 Debug Tool ran. The values displayed are values *after* that statement was run.

If you want to see the value of the variables before *and* after Debug Tool runs the statement, you can enter the SET AUTOMONITOR ON BOTH command. Debug Tool 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, Debug Tool displays the names and values of the variables on the statement that Debug Tool runs next. Then, Debug Tool displays the line ***** Previous Statement *load-name* ::> *cu-name* :> *statement-id* ***** . Below this line, Debug Tool displays the names and values of the variables of the statement that Debug Tool just ran. Each time you enter the STEP command or a breakpoint is encountered, Debug Tool 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 Debug Tool 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 Debug Tool 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 200

Related references

Description of the SET AUTOMONITOR command in *Debug Tool Reference and Messages*.

“Example: How Debug Tool adds variables to the Monitor window automatically”

Example: How Debug Tool 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. Debug Tool 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 Debug Tool automatically adds variables to the Monitor window” on page 201

Related tasks

“Adding variables to the Monitor window automatically” on page 200

Related references

Description of the SET AUTOMONITOR command in *Debug Tool Reference and Messages*.

How Debug Tool handles characters that cannot be displayed in their declared data type

In the Monitor window, Debug Tool 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, Debug Tool displays a dot (X'4B').
- For assembler and LangX COBOL programs, Debug Tool displays a quotation mark (").
- For C and C++ programs, Debug Tool 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”

Modifying characters that cannot be displayed in their declared data type

As described in “How Debug Tool handles characters that cannot be displayed in their declared data type,” 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. Debug Tool 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. Debug Tool 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 196

“Modifying the value of a COBOL variable” on page 217

“Displaying and modifying the value of LangX COBOL variables or storage” on page 229

“Modifying the value of a PL/I variable” on page 235

“Modifying the value of a C variable” on page 245

“Modifying the value of a C++ variable” on page 256

“Displaying and modifying the value of assembler variables or storage” on page 269

Related references

Prefix commands in *Debug Tool 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. Debug Tool 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*) ;

Debug Tool 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. Debug Tool 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 197. 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, use 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 171

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 *Debug Tool 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 292

"Displaying values of C and C++ variables or expressions" on page 320

"Accessing PL/I program variables" on page 311

"Displaying and modifying the value of assembler variables or storage" on page 269

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.

Debug Tool modifies the variable or storage. The command that Debug Tool 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, Debug Tool prevents you from typing in the fifth character.
- If Debug Tool 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, Debug Tool 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 Debug Tool variables, except value of registers %GPRn, %FPRn, %EPRn, %LPRn.
- You cannot modify the value of a Debug Tool 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, Debug Tool 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

Debug Tool 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 176

“Switching between the Memory window and Log window” on page 176

“Displaying memory through the Memory window” on page 17

“Customizing the layout of physical windows on the session panel” on page 274

Related references

“Memory window” on page 163

“Order in which Debug Tool accepts commands from the session panel” on page 170

MEMORY command in *Debug Tool 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). Debug Tool updates the memory with the value you typed in. If you modify the program instruction area of memory, Debug Tool 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 *Debug Tool 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 277

Displaying a list of compile units known to Debug Tool

This topic describes what to do if you want to know which compile units are known to Debug Tool. 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 Debug Tool, 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, Debug Tool 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. Debug Tool 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, Debug Tool displays the following lines in the Log window:

```
USERID.MFISTART.C(CALC)  
USERID.MFISTART.C(PUSHPOP)  
USERID.MFISTART.C(READTKN)
```

If you type "SET QUALIFY CU" over the last line, then press Enter, Debug Tool composes the following command into the command line: SET QUALIFY CU "USERID.MFISTART.C(READTKN)". Press Enter and Debug Tool 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 166

Requesting an attention interrupt during interactive sessions

During an interactive Debug Tool 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 Debug Tool:

- At the next hook if Debug Tool 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 139

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 Debug Tool 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 Debug Tool 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. Debug Tool ends and your program continues running.

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

Related references

Debug Tool Reference and Messages

Chapter 21. Debugging a COBOL program in full-screen mode

The descriptions of basic debugging tasks for COBOL refer to the following COBOL program.

“Example: sample COBOL program for debugging”

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

Related tasks

Chapter 29, “Debugging COBOL programs,” on page 289

“Halting when certain routines are called in COBOL” on page 216

“Modifying the value of a COBOL variable” on page 217

“Halting on a COBOL line only if a condition is true” on page 218

“Debugging COBOL when only a few parts are compiled with TEST” on page 218

“Capturing COBOL I/O to the system console” on page 219

“Displaying raw storage in COBOL” on page 219

“Getting a COBOL routine traceback” on page 220

“Tracing the run-time path for COBOL code compiled with TEST” on page 220

“Generating a COBOL run-time paragraph trace” on page 221

“Finding unexpected storage overwrite errors in COBOL” on page 222

“Halting before calling an invalid program in COBOL” on page 222

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.
  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 " "
        INTO INTEREST-IN NO-OF-PERIODS-IN.
* Convert to numeric values
        COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).
        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) ).
* 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 21, "Debugging a COBOL program in full-screen mode," on page 213

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 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 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 Debug Tool identify your program has been stopped:

```
QUERY LOCATION
```

The Debug Tool 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 213

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 Debug Tool 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 ; G0 ; on the Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool” on page 291

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 G0.

“Example: sample COBOL program for debugging” on page 213

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 Debug Tool commands at the start of COBCALC:

```
AT 67 ; G0 ;  
CLEAR AT 67 ; STEP 4 ;
```

Now set the breakpoint like this:

```
AT 44 IF INTEREST > -1 THEN G0 ; END-IF ;
```

Line 44 is the statement labeled **VALU3**. The command causes Debug Tool to stop at line 44. If the value of INTEREST is greater than -1, the program continues. The command causes Debug Tool 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 Debug Tool command:

```
MOVE '-2 5' TO INPUT-1 ;
```

Run the program by issuing the G0 command. Debug Tool 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 Debug Tool 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 213

Suppose you want to set a breakpoint at entry to COBVALU. COBVALU has been compiled with TEST but the other programs have not. Debug Tool 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 Debug Tool 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 Debug Tool initially displays the empty Source window. Debug Tool 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 216

Capturing COBOL I/O to the system console

To redirect output normally appearing on the system console to your Debug Tool terminal, enter the following command:

```
SET INTERCEPT ON CONSOLE ;
```

“Example: sample COBOL program for debugging” on page 213

For example, if you run COBCALC and issue the Debug Tool SET INTERCEPT ON CONSOLE command, followed by the STEP 3 command, you will see the following output displayed in the Debug Tool 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 Debug Tool terminal instead of the system console by using the Debug Tool INPUT command. For example, if the next COBOL statement executed is ACCEPT INPUT-DATA FROM CONSOLE, the following message appears in the Debug Tool 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 Debug Tool command:

```
INPUT some data ;
```

Note: Whenever Debug Tool 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 206

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 213

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 Debug Tool commands in a file or data set and USE them when Debug Tool initially displays your program. Assuming you have a PDS member, USERID.DT.COMMANDS(COBCALC), that contains the following Debug Tool 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 Debug Tool 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 Debug Tool commands shown in the following example can be used. You can either enter the commands from the Debug Tool command line or place the commands in a file or data set.

“Example: sample COBOL program for debugging” on page 213

Assume you have a PDS member, USERID.DT.COMMANDS(COBCALC2), that contains the following Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool 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 Debug Tool 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 22. Debugging a LangX COBOL program in full-screen mode

The descriptions of basic debugging tasks for LangX COBOL refer to the following program.

“Example: sample LangX COBOL program for debugging”

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 Debug Tool 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 5, “Preparing a LangX COBOL program,” on page 71.
2. Verify that the debug information for this program is located in the COB030 and COB03AO members of the *yourid*.EQALANGX data set.
3. Start Debug Tool as described in “Starting Debug Tool for programs that start outside of Language Environment” on page 143.
4. To load the debug information for this program, enter the following command:

```
LDD (COB030,COB03AO) ;
```

This program is a small example of an OS/VS COBOL program (COB030) that calls another OS/VS COBOL program (COB03A0).

Load module: COB030

COB030

```
*****  
* PROGRAM NAME: COB030 *  
* * *  
* COMPILED WITH IBM OS/VS COBOL COMPILER *  
*****
```

IDENTIFICATION DIVISION.
PROGRAM-ID. COB030.

```
*****  
*  
* LICENSED MATERIALS - PROPERTY OF IBM *  
*  
* 5655-P14: Debug Tool for z/OS *  
* (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.

```
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 'COB03A0' USING LOAN INTEREST-DUE.  
  
DISPLAY 'LOAN: ' LOAN  
DISPLAY 'INTEREST-DUE: ' INTEREST-DUE  
  
STOP RUN.
```

COB03A0

```
*****  
* PROGRAM NAME: COB03A0 *  
* * *  
* COMPILED WITH IBM OS/VSE COBOL COMPILER *  
*****
```

IDENTIFICATION DIVISION.
PROGRAM-ID. COB03A0.

```
*****  
*  
* *  
*****
```

```

* LICENSED MATERIALS - PROPERTY OF IBM
*
* 5655-P14: Debug Tool for z/OS
* (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 Debug Tool as a disassembly CU or for a CU that is not currently known to Debug Tool.

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 Debug Tool
- a name that does not match the name of a CU currently known to Debug Tool

When the CU name is currently known to Debug Tool, 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 Debug Tool, 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 Debug Tool 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, Debug Tool 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 227

Halting when certain LangX COBOL programs are called

“Example: sample LangX COBOL program for debugging” on page 225

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 Debug Tool 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 Debug Tool Log window displays a message similar to the following message:

```
QUERY LOCATION
```

```
You are executing commands in the ENTRY COB030 ::> COB03AO breakpoint.
```

```
The program is currently entering block COB030 ::> 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 ; G0 ; on the Debug Tool command line. Move the cursor over LOAN and press PF4 (LIST). Debug Tool 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'
```

Debug Tool 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 G0 command.

“Example: sample LangX COBOL program for debugging” on page 225

In the COB03AO program, to halt Debug Tool when the LOANAMT variable is set to 100, enter the following command:

```
AT 36 DO; IF 'LOANAMT = 100' THEN G0; END;
```

Line 36 is the line COMPUTE INTEREST = LOANAMT * INTEREST-RATE. The command causes Debug Tool to stop at line 36. If the value of LOANAMT is not 100, the program continues. The command causes Debug Tool 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 225

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, Debug Tool would display an empty Source window. To display a list of compile units known to Debug Tool, 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 Debug Tool. If COB03AO is fetched later on by the application, it might not be known to Debug Tool. 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 225

For example, if you run the example with the following commands, the Log window displays the traceback of callers:

```
LDD (COB030,COB03AO) ;
AT ENTRY COB03AO ;
GO ;
LIST CALLS ;
```

The Log window displays information similar to the following:

```
At ENTRY in LangX COBOL program COB030 ::> COB03AO.
From LINE 74 in LangX COBOL program COB030 ::> COB030.
```

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,” 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, Debug Tool stops if the value of LOAN changes.

Chapter 23. Debugging a PL/I program in full-screen mode

The descriptions of basic debugging tasks for PL/I refer to the following PL/I program.

“Example: sample PL/I program for debugging”

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

Related tasks

Chapter 31, “Debugging PL/I programs,” on page 307

“Halting when certain PL/I functions are called” on page 234

“Modifying the value of a PL/I variable” on page 235

“Halting on a PL/I line only if a condition is true” on page 235

“Debugging PL/I when only a few parts are compiled with TEST” on page 236

“Displaying raw storage in PL/I” on page 236

“Getting a PL/I function traceback” on page 236

“Tracing the run-time path for PL/I code compiled with TEST” on page 237

“Finding unexpected storage overwrite errors in PL/I” on page 238

“Halting before calling an undefined program in PL/I” on page 238

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);
```

```

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 l tok char (100) varying;
dcl l num fixed bin (31,0);
dcl l 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 23, “Debugging a PL/I program in full-screen mode,” on page 231

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 231 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 Debug Tool identify where your program has stopped:

```
QUERY LOCATION
```

The Debug Tool 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 Debug Tool 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 Debug Tool 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 231

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 D0; IF NUM ^= 0 THEN GO; END;
```

Line 31 is the statement labeled **CALLC2**. The command causes Debug Tool to stop at line 31. If the value of NUM is not 0, the program continues. The command causes Debug Tool 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 231

Suppose you want to set a breakpoint at entry to subroutine PUSH. PUSH has been compiled with TEST, but the other files have not. Debug Tool 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 Debug Tool. If PUSH is fetched later on by the application, this compile unit might not be known to Debug Tool. 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 206

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 231

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 Debug Tool 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 Debug Tool.
4. Enter the following command:
USE DT.COMMANDS(PLICALL)
5. Run your program sequence. Debug Tool 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, Debug Tool 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, Debug Tool 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 Debug Tool 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 24. Debugging a C program in full-screen mode

The descriptions of basic debugging tasks for C refer to the following C program.

“Example: sample C program for debugging”

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

Related tasks

Chapter 32, “Debugging C and C++ programs,” on page 319

“Halting when certain functions are called in C” on page 244

“Modifying the value of a C variable” on page 245

“Halting on a line in C only if a condition is true” on page 245

“Debugging C when only a few parts are compiled with TEST” on page 246

“Capturing C output to stdout” on page 246

“Calling a C function from Debug Tool” on page 247

“Displaying raw storage in C” on page 247

“Debugging a C DLL” on page 248

“Getting a function traceback in C” on page 248

“Tracing the run-time path for C code compiled with TEST” on page 248

“Finding unexpected storage overwrite errors in C” on page 249

“Finding uninitialized storage errors in C” on page 249

“Halting before calling a NULL C function” on page 250

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 READTKN.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;
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 24, “Debugging a C program in full-screen mode,” on page 241

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 241 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 241

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 Debug Tool 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 G0 command a thousand times to reach the specific condition. You can instruct Debug Tool to continue executing a program until a specific condition is present.

“Example: sample C program for debugging” on page 241

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 **CALLC2**. The command causes Debug Tool to stop at line 40. If the value of num2 is not 0, the program continues. You can enter Debug Tool 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 241

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. Debug Tool 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 Debug Tool. 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 Debug Tool. 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 Debug Tool 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

Debug Tool Reference and Messages

Calling a C function from Debug Tool

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 241

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 206

Debugging a C DLL

“Example: sample C program for debugging” on page 241

Build PUSHPOP.C as a DLL, exporting push() and pop(). Build CALC.C and READTKN.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 Debug Tool. 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 241

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(READTKN)" :> 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 Debug Tool commands in a file and USE them when Debug Tool initially displays your program. Assuming you have a data set USERID.DTUSE(TRACE) that contains the following Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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 241

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 Debug Tool 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

The descriptions of basic debugging tasks for C++ refer to the following C++ program.

“Example: sample C++ program for debugging”

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

Related tasks

Chapter 32, “Debugging C and C++ programs,” on page 319

“Halting when certain functions are called in C++” on page 255

“Modifying the value of a C++ variable” on page 256

“Halting on a line in C++ only if a condition is true” on page 257

“Viewing and modifying data members of the this pointer in C++” on page 257

“Debugging C++ when only a few parts are compiled with TEST” on page 257

“Capturing C++ output to stdout” on page 258

“Calling a C++ function from Debug Tool” on page 259

“Displaying raw storage in C++” on page 259

“Debugging a C++ DLL” on page 259

“Getting a function traceback in C++” on page 260

“Tracing the run-time path for C++ code compiled with TEST” on page 260

“Finding unexpected storage overwrite errors in C++” on page 261

“Finding uninitialized storage errors in C++” on page 261

“Halting before calling a NULL C++ function” on page 262

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 READTKN.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();
    ~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 25, "Debugging a C++ program in full-screen mode," on page 251

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 251 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
```

Debug Tool displays the names of all the blocks and variables for the currently qualified program. Debug Tool 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 251

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 Debug Tool command:

```
LIST TITLED num
```

Debug Tool 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 Debug Tool 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 Debug Tool” on page 320

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 G0.

“Example: sample C++ program for debugging” on page 251

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) G0; }
```

Line 40 is the statement labeled **CALC2**. The command causes Debug Tool to stop at line 40. If the value of num is not 0, the program will continue. Debug Tool 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 251

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. Debug Tool 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 Debug Tool.

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 Debug Tool, 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 Debug Tool 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

Debug Tool Reference and Messages

Calling a C++ function from Debug Tool

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 251

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 206

Debugging a C++ DLL

“Example: sample C++ program for debugging” on page 251

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 Debug Tool. 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(READTKN)" :> 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 Debug Tool commands, shown in the example below, in a file and USE them when Debug Tool initially displays your program. Assume you have a data set that contains USERID.DTUSE(TRACE) and contains the following Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool 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 26. Debugging an assembler program in full-screen mode

The descriptions of basic debugging tasks for assembler refer to the following assembler program.

“Example: sample assembler program for debugging”

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

Related tasks

Chapter 33, “Debugging an assembler program,” on page 343

“Defining a compilation unit as assembler and loading debug data” on page 266

“Deferred LDDs” on page 267

“Halting when certain assembler routines are called” on page 269

“Displaying and modifying the value of assembler variables or storage” on page 269

“Halting on a line in assembler only if a condition is true” on page 270

“Getting an assembler routine traceback” on page 270

“Finding unexpected storage overwrite errors in assembler” on page 271

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*.EQLANGX data set.
2. Start Debug Tool.
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
        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

```

DISPASM.ASM

```

*****
*
* NAME: DISPASM
*
* Shows an assembler subroutine that displays inbound
* parameters and returns.
*
*****
DISPASM 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   CHARP(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,CEEMOUT)
        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,CEEMOUT)
*
        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
CHARP(80) 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 Debug Tool 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 Debug Tool
- a name that does not match the name of a CU currently known to Debug Tool

When the CU name is currently known to Debug Tool, 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 Debug Tool, 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 Debug Tool 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 Debug Tool. 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 use 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) h1q.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 Debug Tool 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

Debug Tool 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, Debug Tool finds and loads the debug data for both MYPROG and MYPROGA. After the debug data is loaded, Debug Tool 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, Debug Tool 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, Debug Tool uses only the debug information for MYPROG. Then, if you enter the command LDD MYPROGA, Debug Tool does the following steps:

1. If you entered a SET SOURCE command before entering the LDD MYPROG command, Debug Tool 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 Debug Tool did not find debug information in step 1, Debug Tool searches through all previously loaded debug information. If Debug Tool finds a name and CSECT length that matches the name and CSECT length of MYPROGA, Debug Tool 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 263 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 Debug Tool 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 Debug Tool identify where your program has stopped:

```
QUERY LOCATION
```

The Debug Tool 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 ; G0 ; on the Debug Tool 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 Debug Tool 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, Debug Tool 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 263

In the DISPARM program, to stop Debug Tool 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 Debug Tool to stop at line 78. If the value of COUNTER is not 3, the program continues. The command causes Debug Tool 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 263

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, Debug Tool stops if the value in this storage changes.

Chapter 27. Customizing your full-screen session

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 20, "Using full-screen mode: overview," on page 157

Chapter 27, "Customizing your full-screen session"

"Defining PF keys"

"Defining a symbol for commands or other strings"

"Customizing the layout of physical windows on the session panel" on page 274

"Customizing session panel colors" on page 276

"Customizing profile settings" on page 277

"Saving customized settings in a preferences file" on page 279

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 173

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, Debug Tool 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.

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 Debug Tool 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.

Window Layout Selection Panel

Command ==>

<p>1</p> <pre> 1 +-----+ M +-----+ S +-----+ L +-----+</pre>	<p>2</p> <pre> 2 +-----+ - - +-----+ - +-----+</pre>	<p>3</p> <pre> 3 +-----+ - +-----+ - - +-----+</pre>	<p>Legend:</p> <p>L - Log M - Monitor S - Source E - Memory</p> <p>To reassign the Source, Monitor, Log, and Memory windows, type over the current settings or underscores with S, M, L, or E.</p>
<p>4</p> <pre> 4 +-----+ - - +-----+ - - +-----+</pre>	<p>5</p> <pre> 5 +-----+ - - +-----+ - +-----+</pre>	<p>6</p> <pre> 6 +-----+ - - +-----+ - +-----+</pre>	

Enter END/QUIT to return with current settings saved.
CANCEL to return without current settings saved.

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 275

“Resizing physical windows”

“Zooming a window to occupy the whole screen” on page 276

“Saving customized settings in a preferences file” on page 279

Related references

“Debug Tool session panel” on page 157

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 Debug Tool 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, Debug Tool

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 ==>>
Title : field headers  Color  Highlight  Intensity
      : output fields  GREEN  NONE      LOW
Monitor: contents     TURQ   REVERSE   LOW
      : line numbers   TURQ   REVERSE   LOW
Source : listing area  WHITE  REVERSE   LOW
      : prefix area    TURQ   REVERSE   LOW
      : suffix area    YELLOW REVERSE   LOW
      : current line   RED    REVERSE   HIGH
      : breakpoints   GREEN  NONE      LOW
Log    : program output TURQ   NONE      HIGH
      : test input     YELLOW NONE      LOW
      : test output    GREEN  NONE      HIGH
      : line numbers   BLUE  REVERSE   HIGH
Memory : information   GREEN  NONE      LOW
      : offset column  WHITE  NONE      LOW
      : 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 Debug Tool 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 279

Customizing profile settings

The PANEL PROFILE command displays the Profile Settings Panel, which contains profile settings that affect the way Debug Tool 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 Debug Tool 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 Debug Tool is entered) is maintained. Equivalent to SET HISTORY.

History size

Controls the size of the Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool. 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”

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. Debug Tool 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 Debug Tool sessions

All of the customizations described in Chapter 27, “Customizing your full-screen session,” on page 273 can be preserved between Debug Tool sessions by using the save and restore settings feature. See “Recording how many times each source line runs” on page 185 for instructions.

Part 5. Debugging your programs by using Debug Tool commands

Chapter 28. Entering Debug Tool commands

Debug Tool commands can be issued in three modes: full-screen, line, and batch. Some Debug Tool commands are valid only in certain modes or programming languages. Unless otherwise noted, Debug Tool 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 Debug Tool 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 167

“Abbreviating Debug Tool keywords” on page 284

“Entering multiline commands in full-screen” on page 285

“Entering multiline commands in a commands file” on page 285

“Entering multiline commands without continuation” on page 286

“Using blanks in Debug Tool commands” on page 286

“Entering comments in Debug Tool commands” on page 287

“Using constants in Debug Tool commands” on page 287

“Getting online help for Debug Tool command syntax” on page 288

Related references

Debug Tool Reference and Messages

Using uppercase, lowercase, and DBCS in Debug Tool commands

The character set and case vary with the double-byte character set (DBCS) or the current programming language setting in a Debug Tool session.

DBCS

When the DBCS setting is ON, you can specify DBCS characters in the following portions of all the Debug Tool 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++, Debug Tool sets the programming language to C. When the current programming language setting is C, the following rules apply:

- All keywords and identifiers must be the correct case. Debug Tool does 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 "I" and "-" can be used as the boolean operators for OR and NOT.

Abbreviating Debug Tool keywords

When you issue the Debug Tool 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 Debug Tool 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 Debug Tool command truncations:

If you enter the following command...	It will be interpreted as...
A 3	AT 3

If you enter the following command...	It will be interpreted as...
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, Debug Tool 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 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -
very long string");
```

If you want to end a line with a character that Debug Tool 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 Debug Tool 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 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv\  
very long string");
```

- When the current programming language setting is COBOL, columns 1-6 are ignored by Debug Tool 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 vvvvvvvvvvvvvvvvvvvvvvvvv"  
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 D0 command is for conditional looping.
- When the current programming language setting is assembler, disassembly, LangX COBOL, or COBOL, use the language-neutral D0 command.

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

BEGIN command in *Debug Tool Reference and Messages*

DO command (PL/I) in *Debug Tool Reference and Messages*

Using blanks in Debug Tool 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 Debug Tool commands

Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool.

Debug Tool 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 296

Related references

“C and C++ expressions” on page 323

Getting online help for Debug Tool command syntax

You can get help with Debug Tool command syntax by either pressing PF1 or entering a question mark (?) on the command line. This lists all Debug Tool 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 Debug Tool 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 29. 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 297

“Debug Tool evaluation of COBOL expressions” on page 295

Chapter 21, “Debugging a COBOL program in full-screen mode,” on page 213

“Using COBOL variables with Debug Tool” on page 291

“Using DBCS characters in COBOL” on page 293

“Using Debug Tool functions with COBOL” on page 297

“Debug Tool commands that resemble COBOL statements”

“%PATHCODE values for COBOL” on page 293

“Debugging VS COBOL II programs” on page 300

Debug Tool commands that resemble COBOL statements

To test COBOL programs, you can write debugging commands that resemble COBOL statements. Debug Tool 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 Debug Tool.

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

Debug Tool 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 Debug Tool 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.

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 Debug Tool copies commands to the log file, they are formatted according to the rules above so that you can use the log file during subsequent Debug Tool 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 Debug Tool commands”

“COBOL reserved keywords”

Enterprise COBOL for z/OS Language Reference

COBOL compiler options in effect for Debug Tool commands

While Debug Tool allows you to use many commands that are either similar or equivalent to COBOL commands, Debug Tool 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 Debug Tool 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 Debug Tool, 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 Debug Tool

Debug Tool can process all variable types valid in the COBOL language.

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”

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

Related tasks

“Accessing COBOL variables”

“Assigning values to COBOL variables”

“Displaying values of COBOL variables” on page 292 “Declaring session variables in COBOL” on page 295

Accessing COBOL variables

Debug Tool 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 Debug Tool 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

Debug Tool provides three COBOL-like commands to use when assigning values to variables: COMPUTE, MOVE, and SET. Debug Tool assigns values according to COBOL rules. See *Debug Tool 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 Debug Tool 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); G0;
```


Debug Tool 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, Debug Tool 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.

Using DBCS characters in COBOL

Programs you run with Debug Tool can contain variables and character strings written using the double-byte character set (DBCS). Debug Tool 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 Debug Tool, 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 Debug Tool 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 Debug Tool variable %PATHCODE when the current programming language is COBOL.

-1	Debug Tool is not in control as the result of a path or attention situation.
0	Attention function (<i>not</i> 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • [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: <ul style="list-style-type: none"> • [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 Debug Tool 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 Debug Tool 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 412

Related references

Enterprise COBOL for z/OS Language Reference

Debug Tool evaluation of COBOL expressions

Debug Tool 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 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 *Debug Tool 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”

“Using constants in COBOL expressions”

Enterprise COBOL for z/OS Programming Guide

Related references

Debug Tool 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 Debug Tool commands” on page 290

Enterprise COBOL for z/OS Language Reference

Using constants in COBOL expressions

During your Debug Tool 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 Debug Tool, 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.

Additionally, Debug Tool 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 Debug Tool functions with COBOL

Debug Tool 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'000001234'.

Using the %STORAGE function with COBOL

This Debug Tool 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 Debug Tool to process. However, you might have more than one variable named x. You must tell Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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. Debug Tool 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, Debug Tool 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.

Debug Tool 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, Debug Tool does get control of the program. Use the AT ENTRY *, AT EXIT *, AT GLOBAL ENTRY, and AT GLOBAL EXIT commands to set breakpoints that Debug Tool 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). Debug Tool tries to find the listing data set in the following location: user.id.CUName.LIST. If the listing is in a PDS, direct Debug Tool 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 30. Debugging a LangX COBOL program

You can use most of the Debug Tool commands to debug LangX COBOL programs that have debug information available. Any exceptions are noted in *Debug Tool Reference and Messages*. Before debugging a LangX COBOL program, prepare your program as described in Chapter 5, "Preparing a LangX COBOL program," on page 71.

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 Debug Tool 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 Debug Tool 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

Debug Tool locates the debug information in a data set with the following name: yourid.EQALANGX(mypgm). If Debug Tool finds this data set, you can begin to debug your LangX COBOL program. If Debug Tool does not find the data set, enter the SET SOURCE or SET DEFAULT LISTINGS command to indicate to Debug Tool 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.

Debug Tool session panel while debugging a LangX COBOL program

The Debug Tool session panel below shows the information displayed in the Source window while you debug a LangX COBOL program.

```
1 LX COBOL LOCATION: COB030 initialization
Command ==> Scroll ==> PAGE
MONITOR --+---1---+---2---+---3---+---4---+---5---+---6 LINE: 0 OF 0
***** TOP OF MONITOR *****
***** BOTTOM OF MONITOR *****
SOURCE: COB030 ---1---+---2---+---3---+---4---+---5---+ LINE: 1 OF 111
 2 1 3 *****
 2 * PROGRAM NAME: COB030 *
 3 * * *
 4 * COMPILED WITH IBM OS/VS COBOL COMPILER *
 5 *****
 7 IDENTIFICATION DIVISION.
 8 PROGRAM-ID. COB030.
 9 *****
10 *
11 * LICENSED MATERIALS - PROPERTY OF IBM
12 *
13 * 5655-P14: Debug Tool for z/OS
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 Debug Tool Version 13 Release 1 Mod 0
10/23/2013 04:11:41 PM
5655-Q10: Copyright IBM Corp. 1992, 2013
0004 *** Commands file commands follow ***
0005 SET MSGID ON ;
0006 LDD ( COB030, 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 Debug Tool 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 G0 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool sessions.
 5. After the initialization of the transaction that started third is done, all three users can run their Debug Tool sessions, independently, for the same non-Language Environment COBOL program.

%PATHCODE values for LangX COBOL programs

This table shows the possible values for the Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool was started, you can no longer debug non-Language Environment programs or detect non-Language Environment events.

Chapter 31. Debugging PL/I programs

The topics below describe how to use Debug Tool to debug your PL/I programs.

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

Related concepts

“Debug Tool evaluation of PL/I expressions” on page 313

Related tasks

Chapter 23, “Debugging a PL/I program in full-screen mode,” on page 231

Chapter 31, “Debugging PL/I programs”

“Accessing PL/I program variables” on page 311

Related references

“Debug Tool subset of PL/I commands”

“Supported PL/I built-in functions” on page 314

Debug Tool subset of PL/I commands

The table below lists the Debug Tool *interpretive subset* of PL/I commands. This subset is a list of commands recognized by Debug Tool 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	Debug Tool 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 Debug Tool *commands*. Debug Tool makes it possible to issue commands in a manner similar to each language.

The following types of Debug Tool 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 Debug Tool commands into "one" command.

Conditional

IF/THEN, SELECT/WHEN/END

These commands evaluate an expression and control the flow of execution of Debug Tool 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 Debug Tool 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 Debug Tool 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. Debug Tool 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 Debug Tool variable %PATHCODE when the current programming language is PL/I.

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.

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 <i>select-group</i> is about to be executed.
10	The logic following an OTHERWISE within a <i>select-group</i> is about to be executed.

PL/I conditions and condition handling

All PL/I conditions are recognized by Debug Tool. They are used with the AT OCCURRENCE and ON commands.

When an OCCURRENCE breakpoint is triggered, the Debug Tool %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
ZERODIVIDE	CEE349

Note: For Enterprise PL/I programs, the following condition is not supported:

- AT OCCURRENCE CONDITION conditions (name)

Note: The Debug Tool 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 Debug Tool.

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 Debug Tool 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 Debug Tool:

- The ERROR condition
- Attention recognition
- CALL PLITEST
- CALL CEETEST

Debug Tool 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 Debug Tool session variables

PL/I will support all Debug Tool 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 412

Accessing PL/I program variables

Debug Tool 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).

Debug Tool 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 Debug Tool 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 Debug Tool:

- 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 33

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);
```

Debug Tool 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);
```

Debug Tool 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;
```

Debug Tool 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;

```

Debug Tool 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)='Pate1          '
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'
```

Debug Tool 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 Debug Tool.

The Debug Tool 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.

Debug Tool 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 Debug Tool 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 316

Supported PL/I built-in functions

Debug Tool 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	POINTINTERVALUE	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 POINTINTERVALUE

Debug Tool supports the following built-in functions for Enterprise PL/I:

ACOS	HEXIMAGE	OFFSETVALUE	POINTERDIFF
ADDR	HIGH ¹	ORDINALNAME	PTRDIFF
ADDRDATA	IAND	ORDINALPRED	POINTERVUE
ALLOCATION ³	IEOR	ORDINALSUCC	PTRVALUE
ASIN	IOR	ONCODE	PLIRETV
ATAN	INDEX	ONCONDCOND	RAISE2
ATAND	INOT	ONCHAR	REPEAT ¹
ATANH	ISRL	ONGSOURCE	SAMEKEY
BIF_DIM	ISLL	ONSOURCE	SEARCH
BINARYVALUE	LBOUND	ONCONDID	SEARCHR
BINVALUE	LENGTH	ONCOUNT	SIN
COPY ¹	LINENO	ONFILE	SIND
COS	LOG	ONKEY	SINH
COSD	LOG10	ONLOC	SQRT
COSH	LOG2	PAGENO	SUBSTR ¹
COUNT	LOGGAMMA	POINTER	SYSNULL
DATAFIELD	LOW ¹	PTR	TAN
DATE ¹	LOWER2	POINTERADD	TAND
DATETIME ¹	LOWERCASE ¹	PTRADD	TANH
DIMENSION	MAXLENGTH	POINTERSUBTRACT	TALLY
ENDFILE	NULL	PTRSUBTRACT	TIME ¹
ENTRYADDR ^{1,2}	OFFSET		TRANSLATE ¹
ERF	OFFSETADD		UNSPEC ¹
ERFC	OFFSETSUBTRACT		UPPERCASE ¹
EXP	OFFSETDIFF		VERIFY
FILEOPEN			VERIFYR
GAMMA			
HBOUND			
HEX			

Note:

1. To use the built-in functions COPY, DATE, DATETIME, ENTRYADDR, HIGH, LOW, LOWERCASE, REPEAT, SUBSTR, TIME, TRANSLATE, UNSPEC, and UPPERCASE, you must apply the Language Environment runtime PTF for APAR PQ94347 if you are running on z/OS Version 1 Release 6.
2. Pseudovariables are not supported for the ENTRYADDR built-in function under Debug Tool.
3. To use the ALLOCATION built-in function, you must apply the Language Environment runtime PTF for APAR PK16316 if you are running on z/OS Version 1 Release 6 or Version 1 Release 7.

Debug Tool does not support the following built-in functions for Enterprise PL/I:

ABS	EMPTY
ALL	GRAPHIC
ANY	IMAG
BIT	MPSTR
BOOL	REAL
CHAR	STATUS
COMPLETION	STORAGE
CSTG(2)	STRING
CURRENTSTORAGE	
DEFINE STRUCTURE	

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”

Using SET WARNING PL/I command with built-in functions

Certain checks are performed when the Debug Tool 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). Debug Tool tries to find the listing data set in the following location: user.id.CUName.LIST. If the listing is in a PDS, direct Debug Tool 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 running z/OS Version 1 Release 6, apply the Language Environment PTF for APAR PQ99039.
 - 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 314 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.
- To use the DESCRIBE ENVIRONMENT command, you must apply the Language Environment runtime PTF for APAR PQ95664 if you are running z/OS Version 1 Release 6.
- To use the DESCRIBE ATTRIBUTES command, you must apply the Language Environment runtime PTF for APAR PK30522 if you are running on z/OS Version 1 Release 6 through Version 1 Release 8.
- For typed structures, the following restrictions apply:

Debug Tool 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.

Debug Tool supports the debugging of PL/I typed structures for procedures compiled with the Enterprise PL/I V4R2 or later compilers. If you are running with Language Environment V1R11, V1R12 or V1R13, apply the PTFs for Language Environment APAR PM30489. 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 32. Debugging C and C++ programs

The topics below describe how to use Debug Tool to debug your C and C++ programs.

“Example: referencing variables and setting breakpoints in C and C++ blocks” on page 333

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

Related concepts

“C and C++ expressions” on page 323

“Debug Tool evaluation of C and C++ expressions” on page 327

“Scope of objects in C and C++” on page 330

“Blocks and block identifiers for C” on page 332

“Blocks and block identifiers for C++” on page 332

“Monitoring storage in C++” on page 340

Related tasks

Chapter 24, “Debugging a C program in full-screen mode,” on page 241

Chapter 25, “Debugging a C++ program in full-screen mode,” on page 251

“Using C and C++ variables with Debug Tool” on page 320

“Declaring session variables with C and C++” on page 322

“Calling C and C++ functions from Debug Tool” on page 324

“Intercepting files when debugging C and C++ programs” on page 328

“Displaying environmental information for C and C++ programs” on page 334

“Stepping through C++ programs” on page 338

“Setting breakpoints in C++” on page 338

“Examining C++ objects” on page 339

“Qualifying variables in C and C++” on page 335

Related references

“Debug Tool commands that resemble C and C++ commands”

“%PATHCODE values for C and C++” on page 322

“C reserved keywords” on page 325

“C operators and operands” on page 326

“Language Environment conditions and their C and C++ equivalents” on page 326

Debug Tool commands that resemble C and C++ commands

Debug Tool's command language is a subset of C and C++ commands and has the same syntactical requirements. Debug Tool 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 Debug Tool.

Command	Description
block ({})	Composite command grouping
break	Termination of loops or switch commands
declarations	Declaration of session variables
do/while	Iterative looping

Command	Description
expression	Any C expression except the conditional (?) operator
for	Iterative looping
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 325

z/OS XL C/C++ Language Reference

Using C and C++ variables with Debug Tool

Debug Tool 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”

“Displaying values of C and C++ variables or expressions”

“Assigning values to C and C++ variables” on page 321

Accessing C and C++ program variables

Debug Tool 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 Debug Tool 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] ); G0;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (G0), 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] ); G0;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (G0), 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
```

Debug Tool supports all C operators except the ternary 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 Debug Tool” on page 324

%PATHCODE values for C and C++

The table below shows the possible values for the Debug Tool variable %PATHCODE when the current programming language is C and C++.

-1	Debug Tool is not in control as the result of a path or attention situation.
0	Attention function (<i>not</i> 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.
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 <code>do/while</code> , <code>for</code> , or <code>while</code> statement is about to be executed. This can be a single or <code>Null</code> statement and not a block statement.
7	The logic following an <code>if(...)</code> is about to be executed.
8	The logic following an <code>else</code> is about to be executed.
9	The logic following a case within an <code>switch</code> is about to be executed.
10	The logic following a default within a <code>switch</code> is about to be executed.
13	The logic following the end of a <code>switch</code> , <code>do</code> , <code>while</code> , <code>if(...)</code> , or <code>for</code> is about to be executed.
17	A <code>goto</code> , <code>break</code> , <code>continue</code> , or <code>return</code> 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 Debug Tool (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 412

“Qualifying variables and changing the point of view in C and C++” on page 335

C and C++ expressions

Debug Tool allows evaluation of expressions in your test program. All expressions available in C and C++ are also available within Debug Tool 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, Debug Tool variables, or session variables declared during a Debug Tool session. Language constants are specified as described in the C and C++ Language Reference publications.

The Debug Tool 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 Debug Tool. 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 Debug Tool than when evaluated by the compiler.

The following examples show you various ways Debug Tool supports the use of expressions in your programs:

- Debug Tool assigns 12 to a (the result of the `printf()` function call, as in:

```
a = (1,2/3,a++,b++,printf("hello world\n"));
```
- Debug Tool 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);
```
- Debug Tool 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

“Debug Tool evaluation of C and C++ expressions” on page 327
z/OS XL C/C++ Language Reference

Calling C and C++ functions from Debug Tool

You can perform calls to user and C library functions within Debug Tool, 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 Debug Tool 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++.

Debug Tool 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

- Debug Tool 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 Debug Tool which then passes control to the signal handler.

Debug Tool 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 Debug Tool. No discernible difference exists if the evaluation of arguments does not have side effects.
- Debug Tool 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	= += -= *= /= <<= >>= %= &= ^= =
Comma	left to right	,

Language Environment conditions and their C and C++ equivalents

Language Environment condition names (the symbolic feedback codes CEExxx) 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

Language Environment condition	Description	Equivalent C/C++ condition
CEE34D	Exponent underflow exception	SIGFPE
CEE34E	Significance exception	SIGFPE
CEE34F	Floating-point divide exception	SIGFPE

Debug Tool evaluation of C and C++ expressions

Debug Tool 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.

Debug Tool 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 Debug Tool 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 7.2370055773322622139731865630429929E+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 323
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 Debug Tool 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 Debug Tool is waiting for input for an intercepted file (for example, if you interactively enter `fwrite(..)` when Debug Tool 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 Debug Tool, 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 Debug Tool, 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 Debug Tool, 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)`.

Debug Tool 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 Debug Tool at any point in the source file, not just from the point in the source file where it is declared. Debug Tool 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, Debug Tool 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 333

Related concepts

“Storage classes in C and C++”

Storage classes in C and C++

Debug Tool supports the change and reference of all objects declared with the following storage classes:

```
auto
register
static
extern
```

Session variables declared during the Debug Tool session are also available for reference and change.

An object with `auto` storage class is available for reference or change in Debug Tool, 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 Debug Tool, provided the variable has not been optimized to a register.

An object with `static` storage class is always available for change or reference in Debug Tool. 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 Debug Tool. 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 Debug Tool 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. Debug Tool 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 Debug Tool 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:>%BLOCKzzz`

Long form

`function_name:>%BLOCKxxx :>%BLOCKyyy: ... :>%BLOCKzzz`

`%BLOCKzzz` is contained in `%BLOCKyyy`, which is contained in `%BLOCKxxx`. 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 Debug Tool 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()
{
    >>> Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool will never know about the variables `j` and `temp` because they are defined in a block that is nested in another block. Debug Tool 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
<code>sort</code>	<code>sort</code>
<code>%BLOCK2</code>	<code>%BLOCK8</code>
<code>%BLOCK3</code>	<code>%BLOCK13</code>

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 Debug Tool 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 Debug Tool. 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 336

Related tasks

“Qualifying variables in C and C++”

“Changing the point of view in C and C++” on page 336

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 Debug Tool 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 Debug Tool 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 Debug Tool variable %BLOCK.

“Example: using qualification in C”

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 332

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”

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];
            :
            >>> Debug Tool is given <<<
            >>> control here. <<<
            :
            table[i] = table[j];
```

```

        table[j] = temp;
    }
}
}

```

When Debug Tool receives control, variables `i`, `j`, `temp`, `table`, and `length` can be specified without qualifiers in a command. If variable `sn` is referenced, Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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:

```
AT ENTRY square(int,int)
```

```
AT ENTRY shapes::square(int)
```

```
AT EXIT outer::inner::func()
```

```
AT EXIT Stack<int,5>::Stack()
```

```
AT ENTRY Plus::operator++(int)
```

```
AT ENTRY ::fail()
```

'simple' method square

Method square qualified by its class shapes.

Nested classes. Outer and inner are classes. `func()` is within class inner.

Templates.

Overloaded operator.

Functions defined at file scope must be referenced by the global scope operator `::`

The following examples are invalid:

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 Debug Tool 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 174

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”

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 yyyyyyy 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 Debug Tool 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,H00K..PGM-ENTRY
*   return 2*j;
      EX    r0,H00K..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,H00K..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, Debug Tool 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 33. Debugging an assembler program

To debug programs that have been assembled with debug information, you can use most of the Debug Tool commands. Any exceptions are noted in *Debug Tool Reference and Messages*. Before debugging an assembler program, prepare your program as described in Chapter 6, "Preparing an assembler program," on page 75.

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 Debug Tool 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 Debug Tool 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
```

Debug Tool locates the debug information in a data set with the following name: *yourid.EQALANGX(mypgm)*. If Debug Tool 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 Debug Tool 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.

Debug Tool session panel while debugging an assembler program

The Debug Tool session panel below shows the information displayed in the Source window while you debug an assembler program.

```

Assemble LOCATION: PUBS :=> 34
Command ==>
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 0000078      *   EQU *
34 0000078      +   OPEN ((2),INPUT)
34              +   CNOP 0,4
34 0000078 4510 B080 +   BAL 1,**8
35 000007C      +   DC A(0)
36 0000080 5021 0000 +   ST 2,0(1,0)
37 0000084 9280 1000 +   MVI 0(1),128
38 0000088 0A13    +   SVC 19
39              5 6   CALL CEEMOUT,(STRING,DEST,0),VL
39              +   SYSSTATE TEST
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 Debug Tool Version 13 Release 1 Mod 0
10/23/2013 04:11:41 PM
5655-Q10: Copyright IBM Corp. 1992, 2013
0004 EQA1872E An error occurred while opening file: INSPREF. 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

Debug Tool Reference and Messages

%PATHCODE values for assembler programs

This table shows the possible values for the Debug Tool %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)	These instructions are considered an Exit only if this instruction is not followed by a valid instruction.
		BALR R14,R15 (05EF)	
		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)	Except BAL 1,xxx is not considered a CALL
		BAL (45)	
		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.
6	A conditional branch is about to be executed.	BC x (47x)	$x \neq 15$ & $X \neq 0$
		BCR x (07x)	$x \neq 15$ & $X \neq 0$
		BCT (46)	
		BCTR (06)	
		BCTGR (B946)	
		BXH (86)	
		BXHG (EB44)	
		BXLE (87)	
		BXLEG (EB45)	
		BRC x (A7x4)	$x \neq 15$ & $X \neq 0$
		BRCL (C0x4)	
		BRCT (A7x6)	
		BRCTG (A7x7)	
		BRXH (84)	
		BRXHG (EC44)	
BRXLE (85)			
BRXLG (EC45)			

%PATHCODE	Entry type	Instruction	Additional requirements or comments
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, 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 Debug Tool for programs that start outside of Language Environment” on page 143.
 - 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 433.

- 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, Debug Tool 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 Debug Tool 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 Debug Tool. 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 Debug Tool 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 Debug Tool 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. Debug Tool 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 Debug Tool 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, Debug Tool 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, Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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

Debug Tool 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 Debug Tool. This is because Debug Tool sometimes replaces instructions with SVCs in order to create breakpoints.

For example, Debug Tool 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

Debug Tool 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 Debug Tool 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 is shipped in the Debug Tool 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 Debug Tool. 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, Debug Tool 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 Debug Tool 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
<pre> ModInst BC 0,Target ... MVI ModInst+1,X'F0' </pre>	<pre> ModInst DC X'4700',S(Target) ... MVI ModInst+1,X'F0' </pre>

- 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
<pre> ModInst BC 0,Target ... MVI ModInst+1,X'F0' </pre>	<pre> ModInst BC 0,Target ... MVC ModInst(4),NewInst ... NewInst DC X'47F0',S(Target) </pre>

Code that modifies an instruction	Corresponding code that replaces an instruction with one defined by a DC
Code that modifies an instruction	Corresponding code that replaces an instruction marked by EQAMODIN
<pre> ModInst BC 0,Target ... MVI ModInst+1,X'F0' </pre>	<pre> ModInst BC 0,Target ... MVC ModInst(4),NewInst ... EQAMODIN NewInst NewInst BC 15,Target </pre>

Chapter 34. 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 Debug Tool 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 Debug Tool commands.

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. Debug Tool 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 Debug Tool screen (upper left corner) displays the word **Disassem**. The Debug Tool 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 Debug Tool Version 13 Release 1 Mod 0
10/23/2013 04:11:41 PM
5655-Q10: Copyright IBM Corp. 1992, 2013
0004 EQA1872E An error occurred while opening file: INSPREF. 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. Debug Tool 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. Debug Tool 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, Debug Tool displays a warning message and lets you set the appropriate breakpoints. Then you can do the step.

Debug Tool 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.

Debug Tool 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

Debug Tool 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 Debug Tool. 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, Debug Tool 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
<pre>ModInst BC 0,Target ... MVI ModInst+1,X'F0'</pre>	<pre>ModInst BC 0,Target ... MVC ModInst(4),NewInst ... NewInst BC 15,Target</pre>

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. Debug Tool 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. Debug Tool 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, Debug Tool 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 Debug Tool encounters a valid save area backchain.

Part 6. Debugging in different environments

Chapter 35. 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 Debug Tool to debug your DB2 programs.

- Chapter 7, “Preparing a DB2 program,” on page 79
- “Processing SQL statements” on page 79
- “Linking DB2 programs for debugging” on page 81
- “Binding DB2 programs for debugging” on page 82
- “Debugging DB2 programs in batch mode”
- “Debugging DB2 programs in full-screen mode” on page 364

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

Related tasks

Chapter 7, “Preparing a DB2 program,” on page 79

DB2 UDB for z/OS Application Programming and SQL Guide

Debugging DB2 programs in batch mode

In order to debug your program with Debug Tool while in batch mode, follow these steps:

1. Make sure the Debug Tool 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 7, “Preparing a DB2 program,” on page 79

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 Debug Tool Setup Utility (DTSU)

The Debug Tool Setup Utility is available through Debug Tool Utilities.

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(progrname) PLAN(planname) LIB('user.library')  
      PARM(' /TEST(*,*,*)')
```

The following example starts a non-Language Environment COBOL program:

```
RUN PROG(EQANMDBG) PLAN(planname) LIB('user.library')  
      PARM('progrname, /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 Debug Tool 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 Debug Tool commands.

Note: If your source does not come up in Debug Tool 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 7, "Preparing a DB2 program," on page 79

"Starting Debug Tool for programs that start outside of Language Environment" on page 143

Related references

DB2 UDB for z/OS Administration Guide

Chapter 36. Debugging DB2 stored procedures

A DB2 stored procedure is a compiled high-level language (HLL) program that can run SQL statements. Debug Tool 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 8, “Preparing a DB2 stored procedures program,” on page 83. 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 8, “Preparing a DB2 stored procedures program,” on page 83
“Resolving some common problems while debugging DB2 stored procedures”

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 19. 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: <code>WLM VARY APPLENV=<i>applenv</i>,RESUME</code> where <i>applenv</i> 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.

Table 19. 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	<p>This can occur for many reasons. If the stored procedure abends without calling Debug Tool, analyze the Procedure for any logic errors. If the Procedure runs successfully without Debug Tool, 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.</p>

Chapter 37. 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

Debug Tool'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 Debug Tool Utilities. For detailed information, see "Starting Debug Tool Utilities" on page 452.
2. In the Debug Tool 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.EQUA0PTS'

Filters: 3
 / Display full transaction list.
 _ Display only transactions you are registered to debug.
 _ Filter by name ==> dtmq

(E) Edit      (S) Start Region   (P) Stop Region
(R) Register  (D) De-register

Sel Transaction PSB name Reserved user   Region name   Status
- 4  ADDINV     DFSSAM04
-   ADDPART     DFSSAM04
-   APOL11      APOL1
-   APOL12      APOL1
-   APOL13      APOL1
-   APOL14      APOL1
F1=Help      F3=Exit      F7=Backward  F8=Forward   F12=Cancel

```

1 IMS System

Specify the IMS subsystem identifier where you debug.

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 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. Add data sets to the message region STEPLIB concatenation.

When Debug Tool 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, Debug Tool 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.

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.

- The following screen highlights the fields on the Edit pattern-matching parameters panel (EQAPMPED).

```

----- Edit pattern-matching parameters -----
Command ==> _____ Scroll ==> CSR

Message processing program debug settings:

Region class . . . 021           Region name . . @USRT001
Transaction . . . . ITOC05
User ID to match . . USRT001 1
Transaction Message ITOC04 2
3 Match case / 4 Data is hex _

F1=Help      F3=Exit    F4=Run      F5=Findnext  F7=Backward
F8=Forward   F10=Submit F12=Cancel

```

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.

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 `./O` or `./T` commands.

Note: If your source (C and C++) or listing (COBOL and PL/I) does not come up in Debug Tool 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 IMS batch programs in batch mode

You can use Debug Tool to debug IMS programs in batch mode. The debug commands must be predefined and included in one of the Debug Tool 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 Debug Tool 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 373 for instructions.
2. Choose a debugging interface. See "Choosing an interface and gathering information for non-Language Environment IMS MPPs" on page 373 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.”
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 *Debug Tool Customization Guide* for instructions on how to include the APPLFE=EQANIAFE parameter string in the JCL that starts a region and EQANISSET.
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.”

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.”

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.

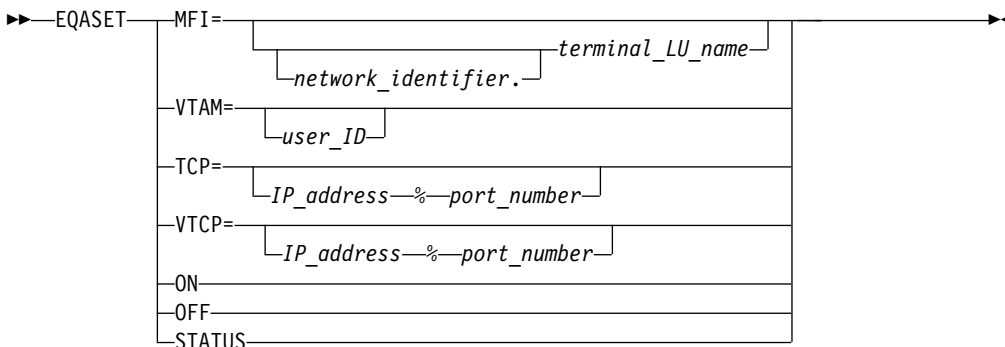
To request information about your existing preferences, enter the command `EQASET STATUS`.

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

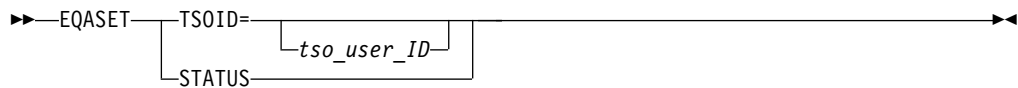
Display the current debugging preference. The `EQASET` transaction displays only the first 25 characters of the IP address.

Debugging Language Environment IMS MPPs without issuing `/SIGN ON`

The Language Environment user exit for IMS (`EQADICXT`) constructs the name of an MVS data set that contains the Language Environment runtime options, including the `TEST` runtime option. `EQADICXT` 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, `EQADICXT` cannot locate the MVS data set. To specify that `EQADICXT` assigns a *TSO* user ID as the value for the `&USERID` token, run the `EQASET` transaction specifying the `TS0ID` option. For a description of the `EQASET` transaction with the `TS0ID` option, see “Syntax of the `EQASET` transaction for Language Environment MPPs.”

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 111.

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 Debug Tool Utilities

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 Debug Tool 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 Debug Tool Utilities, do the following steps:

1. Start Debug Tool Utilities. If you do not know how to start Debug Tool Utilities, see “Starting Debug Tool Utilities” on page 9.
2. In the Debug Tool 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 Debug Tool 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

You can use predefined IMS message region templates to debug a specific transaction in a private message region by using Debug Tool 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 Debug Tool Utilities. For detailed information, see “Starting Debug Tool Utilities” on page 9.
2. In the Debug Tool 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 Debug Tool Setup Utility data set
 - Templates previously customized and stored in a private Debug Tool Setup Utility data set

If you use a member from a private Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool becoming unresponsive

When you debug an IMS application program, the way IMS manages resources might occasionally make Debug Tool 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 Debug Tool 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 PLCT⁹, 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 Debug Tool 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

9. 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 38. 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 Debug Tool 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 3, “Planning your debug session,” on page 23
 - Chapter 4, “Updating your processes so you can debug programs with Debug Tool,” on page 61
 - Chapter 9, “Preparing a CICS program,” on page 87
 - Chapter 17, “Starting Debug Tool under CICS,” on page 147

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, Debug Tool displays a list similar to the following list:

```

COBOL      LOCATION: ZCONPRGA  :> 274
Command ==>
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-XXXX) .
273 *          END-EXEC .
274 Move 'PrgA-ChanB-ContC' to dfhc0160 .
275 Move 'PrgA-CHANB' to dfhc0161 .
276 Call 'DFHEI1' using by content x'341670000720000002000000 .
277 - '00f0f0f0f5f3404040' by content x'0000' by reference .
278 PrgA-ChanB-XXXX by reference dfhc0160 by content LENGTH .
279 PrgA-ChanB-XXXX 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, Debug Tool displays the contents of the container in a format similar to the following diagram:

```

COBOL      LOCATION: ZCONPRGA  :> 211.1
Command ==>
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 *Debug Tool Reference and Messages*
- LIST CONTAINER command in *Debug Tool 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 Debug Tool finds a load module, compile unit, or both, whose name matches the name or part of the name (a pattern) that you specified, Debug Tool 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 Debug Tool (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 Debug Tool 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 Debug Tool 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 Debug Tool 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 *Debug Tool Reference and Messages*

ENABLE command in *Debug Tool Reference and Messages*

LIST CADP or DTCN command in *Debug Tool Reference and Messages*

Preventing Debug Tool from stopping at EXEC CICS RETURN

Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool to check for storage violations, enter the command CHKSTGV. Debug Tool checks the task that you are debugging for storage violations.

You can instruct Debug Tool 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 Debug Tool 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 Debug Tool display settings (for example, color settings) while you debug a pseudo-conversational CICS program, Debug Tool might restore the default settings. To ensure that your changes remain in effect every time your program starts Debug Tool, 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 Debug Tool 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, Debug Tool 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 195 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

Debug Tool Reference and Messages

Restrictions when debugging under CICS

The following restrictions apply when debugging programs with the Debug Tool 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 Debug Tool 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, Debug Tool 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, Debug Tool 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, Debug Tool does not run the EQAOPTS LOGDSN command; therefore, Debug Tool 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.)
- Debug Tool disables Omegamon RLIM processing for any CICS task which is being debugged.
- You can start Debug Tool 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 Debug Tool 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, Debug Tool will not start. Non-Language Environment assembler or non-Language Environment COBOL programs that are called in this way are identified by Debug Tool 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 Debug Tool 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 use the DTST transaction to display and modify CICS storage. See Appendix H, "Displaying and modifying CICS storage with DTST," on page 527 for more information.

Chapter 39. 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 Debug Tool 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 Debug Tool 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 Debug Tool output from the emulator session. However, if Debug Tool 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 workstation application such as Rational Developer for System z.

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_workstation_id%8001:*)
```

The 2nd, 3rd, and 4th options above support debugging a batch ISPF program.

- 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, Debug Tool 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%GYOUNG:*
```

- 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%GYOUNG:;'.
```

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**

| **Debug Tool runtime options** in *Debug Tool Reference and Messages*
| "Starting Debug Tool with CEETEST" on page 127

Chapter 40. 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 Debug Tool'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 Debug Tool 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 Debug Tool”

“Debugging without hooks, statement tables, and symbol tables” on page 395

“Debugging optimized COBOL programs” on page 396

Fine-tuning your programs for Debug Tool

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 396. 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(NOH00K) for Enterprise COBOL for z/OS Version 4
 - TEST for Enterprise COBOL for z/OS Version 5

Using the Dynamic Debug facility, Debug Tool 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, Debug Tool is not able to regain control without compiled-in hooks. In such a case you can request an interrupt three times. After the third time, Debug Tool is able to stop program execution and prompt you to enter QUIT or G0. If you enter QUIT, your Debug Tool session ends. If you enter G0, 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 Debug Tool. 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 *Debug Tool 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 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 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

Debug Tool 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, Debug Tool 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 Debug Tool 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, with the TEST compiler option.

Note: 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,SEPERATE) 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. Debug Tool 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 of 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 and using OPT and NOEJPD of the TEST compiler option, 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.

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, Debug Tool 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 Debug Tool commands on those variables or statements. For example, you cannot list the value of a variable removed by the optimizer.

Chapter 41. 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 Debug Tool 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
- 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 Debug Tool when a new process is created by fork or exec, use the `EQAOPTS MULTIPROCESS` command. For more information about `EQAOPTS`, see *Debug Tool Reference and Messages*.

Chapter 42. 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 13, "Starting Debug Tool from the Debug Tool Utilities," on page 123

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 Debug Tool and your user program.

You do not have to use EQANMDBG to initiate a Debug Tool 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, Debug Tool 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 16, "Starting Debug Tool for batch or TSO programs," on page 139
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.

Part 7. Debugging complex applications

Chapter 43. Debugging multilanguage applications

To support multiple high-level programming languages (HLL), Debug Tool 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 Debug Tool 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 Debug Tool tries to let the language itself guide how Debug Tool 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 407

“Debugging multilanguage applications” on page 411

“Handling conditions and exceptions in Debug Tool” on page 409

Related references

“Debug Tool evaluation of HLL expressions”

“Debug Tool interpretation of HLL variables and constants” on page 406

“Debug Tool commands that resemble HLL commands” on page 406

“Coexistence with other debuggers” on page 414

“Coexistence with unsupported HLL modules” on page 414

Debug Tool evaluation of HLL expressions

When you enter an expression, Debug Tool records the programming language in effect at that time. When the expression is run, Debug Tool 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

“Debug Tool evaluation of C and C++ expressions” on page 327

“Debug Tool evaluation of COBOL expressions” on page 295

“Debug Tool evaluation of PL/I expressions” on page 313

Debug Tool interpretation of HLL variables and constants

Debug Tool 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 Debug Tool are:

- Program variables defined by the HLL compiler's symbol table
- Debug Tool variables denoted by the percent (%) sign
- Session variables declared for a given Debug Tool session and existing only for the session

HLL variables

Some variable references require language-specific evaluation, such as pointer referencing or subscript evaluation. Once again, the Debug Tool interprets each case in the manner of the HLL in question. Below is a list of some of the areas where Debug Tool 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. Debug Tool accepts both types of constants in C and C++, COBOL, and PL/I.

Debug Tool commands that resemble HLL commands

To allow you to use familiar commands while in a debug session, Debug Tool provides an *interpretive subset* of commands for each language. This consists of commands that have the same syntax, whether used with Debug Tool or when writing application programs. You use these commands in Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool command.

Multiway

These commands allow you to program multiway logic in the Debug Tool command language.

In addition, Debug Tool supports special kinds of commands for some languages.

Related references

“Debug Tool commands that resemble C and C++ commands” on page 319

“Debug Tool commands that resemble COBOL statements” on page 289

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, Debug Tool 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 Debug Tool 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 Debug Tool how to determine the proper `x`.

You also need a way to change the Debug Tool'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”

“Changing the point of view” on page 409

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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool; 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 Debug Tool, 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 Debug Tool's name 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 335

“Qualifying variables and changing the point of view in COBOL” on page 297

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 Debug Tool 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 Debug Tool 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 Debug Tool. 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 45, “Debugging across multiple processes and enclaves,” on page 417

“Changing the point of view in C and C++” on page 336

“Changing the point of view in COBOL” on page 299

Handling conditions and exceptions in Debug Tool

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, Debug Tool 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 Debug Tool, 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 Debug Tool”

“Handling exceptions within expressions (C and C++ and PL/I only)” on page 411

Handling conditions in Debug Tool

You can use either or both of the two methods during a debugging session to ensure that Debug Tool gains control at the occurrence of HLL conditions.

If you specify TEST(ALL) as a run-time option when you begin your debug session, Debug Tool gains control at the occurrence of most conditions.

Note: Debug Tool recognizes all Language Environment conditions that are detected by the Language Environment error handling facility.

You can also direct Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool and the task ends with an ABEND 4038.

- Control of your program's execution can be returned to the HLL exception handler, using the G0 command, so that processing proceeds as if Debug Tool 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 G0 BYPASS command, bypassing any further processing of this exception either by the user program or the environment.
- PL/I allows G0 T0 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 G0, 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 326

“PL/I conditions and condition handling” on page 309

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 Debug Tool expression, you can use the Debug Tool command SET WARNING to control Debug Tool and program response. During an interactive Debug Tool 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 Debug Tool 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 316

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 Debug Tool.

- 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 Debug Tool.

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”
- “Using session variables across different programming languages”

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. Debug Tool 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”

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
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

“Debug Tool interpretation of HLL variables and constants” on page 406

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 182 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”

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 Debug Tool.

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

Related references

“Coexistence with other debuggers”

Chapter 44. Debugging multithreading programs

You can run your multithreading programs with Debug Tool when POSIX `pthread_create` is used to create new threads under Language Environment. When more than one thread is involved in your program, Debug Tool might be started by any or all of them. Because conflicting use of the terminal or log file, for example, could occur if Debug Tool 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 Debug Tool, Debug Tool accepts the request and begins operating on behalf of thread A. If, during that period, thread B calls Debug Tool, the request from thread B is held until the request from thread A is complete (for example, you issued a STEP or GO command). Debug Tool 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

Chapter 45. Debugging across multiple processes and enclaves

There is a single Debug Tool 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 Debug Tool 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 Debug Tool within an enclave”

“Viewing Debug Tool windows across multiple enclaves”

“Ending a Debug Tool session within multiple enclaves” on page 418

“Using Debug Tool commands within multiple enclaves” on page 418

Starting Debug Tool within an enclave

After an enclave in a process activates Debug Tool, it remains active throughout subsequent enclaves in the process, regardless of whether the run-time options for the enclave specify TEST or NOTEST. Debug Tool 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 Debug Tool 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 Debug Tool in each new process.

If Debug Tool 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, Debug Tool is not active for the parent enclave, even upon return from the child enclave.

Upon activation of Debug Tool, the initial commands string, primary commands file, and the preferences file are run. They run only once, and affect the entire Debug Tool session. A new primary commands file cannot be started for a new enclave.

Viewing Debug Tool 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 Debug Tool session within multiple enclaves

If you specify the NOPROMPT suboption of the TEST runtime option for the next process on the host, Debug Tool restores the saved breakpoints after it gains control of that next process. However, Debug Tool might gain control of the process after many statements have been run. Therefore, Debug Tool might not run some or all of the following breakpoints:

- STATEMENT/LINE
- ENTRY
- EXIT
- LABEL

If you have not used these breakpoint types, you can specify NOPROMPT.

In a single enclave, QUIT closes Debug Tool. For CICS non-Language Environment programs (assembler or non-Language Environment COBOL), QUIT closes Debug Tool and the task ends with an ABEND 4038, regardless of the link level.

In a nested enclave, however, QUIT causes Debug Tool 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.

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 Debug Tool commands within multiple enclaves

Some Debug Tool commands and variables have a specific scope for enclaves and processes. The table below summarizes the behavior of specific Debug Tool commands and variables when you are debugging an application that consists of multiple enclaves.

Debug Tool command	Affects current enclave only	Affects entire Debug Tool session	Comments
%CAAADDRESS	X		
AT GLOBAL		X	
AT TERMINATION		X	
code coverageSTART		X	
code coverageSTOP		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	

Debug Tool command	Affects current enclave only	Affects entire Debug Tool session	Comments
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.
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 Debug Tool session variable names.
MONITOR GLOBAL		X	Applies to Global monitors.
PLAYBACK ENABLE		X	The PLAYBACK command that informs Debug Tool to begin the recording session.
PLAYBACK DISABLE		X	The PLAYBACK command that informs Debug Tool to stop the recording session.
PLAYBACK START		X	The PLAYBACK command that suspends execution of the program and indicates to Debug Tool to enter replay mode.
PLAYBACK STOP		X	The PLAYBACK command that terminates replay mode and resumes normal execution of Debug Tool.
PLAYBACK BACKWARD		X	The PLAYBACK command that indicates to Debug Tool to perform STEP and RUNTO commands backward, starting from the current point and going to previous points.

Debug Tool command	Affects current enclave only	Affects entire Debug Tool session	Comments
PLAYBACK FORWARD		X	The PLAYBACK command that indicates to Debug Tool 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 Debug Tool. 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 Debug Tool. 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.
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 Debug Tool session.

-
2. If no active condition handler exists for the specified condition, the default condition handler can cause the program to end prematurely.

Chapter 46. Debugging a multiple-enclave interlanguage communication (ILC) application

When you debug a multiple-enclave ILC application with Debug Tool, use the SET PROGRAMMING LANGUAGE to change the current programming language setting. The programming language setting is limited to the languages currently known to Debug Tool (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.

Chapter 47. Debugging programs called by Java native methods

This topic describes how to debug, with Debug Tool, 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 (for example, Rational Developer for System z or IBM Problem Determination Tools Studio) 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 Debug Tool 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 Debug Tool with CEETEST” on page 127

"Choosing TEST or NOTEST compiler suboptions for COBOL programs" on page 27

"Choosing TEST or NOTEST compiler suboptions for PL/I programs" on page 33

"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

Chapter 48. 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)

Debug Tool 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 Debug Tool the data set name from which the program was loaded so Debug Tool can pass it to the binder APIs. However, z/OS does not have this information for programs loaded from LLA.

When Debug Tool attempts to debug a program loaded from LLA, Debug Tool does the following steps:

- Debug Tool checks for the allocation of DD name EQALOAD and checks that it contains a member name that matches the program that LLA loaded.
- If Debug Tool does not find a program by the specified name in EQALOAD, Debug Tool checks for the allocation of DD name STEPLIB and checks that it contains a member name that matches the program that LLA loaded.
- If Debug Tool 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, Debug Tool 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:

- Debug Tool 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, Debug Tool cannot find STEPLIB.

Debugging user programs that use system prefixed names

Debug Tool assumes that load module and compile unit names that begin with specific prefixes are system components. For example, EQAxxxxx is a Debug Tool module, CEExxxxx is a Language Environment module, and IGZxxxxx is a COBOL module.

Debug Tool does not try to debug load modules or compile units that have these prefixes for the following reasons:

- Debug Tool might perform improper recursions that might result in abnormally endings (ABENDs) or other erroneous behavior.

- Debug Tool 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.

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, Debug Tool, 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 Debug Tool recognizes as system components.

Displaying system prefixes

You can display a list of prefixes that Debug Tool 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 Debug Tool.

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

Related references

NAMES command in *Debug Tool 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 Debug Tool, use the NAMES INCLUDE command to indicate to Debug Tool 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 *Debug Tool 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, Debug Tool 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 Debug Tool that these are data-only modules that contain no executable code. Debug Tool 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 Debug Tool and Debug Tool 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 *Debug Tool Reference and Messages*

Optimizing the debugging of large applications

Debug Tool was 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 Debug Tool loads debug data only for modules that you indicate that you want to debug.
- Indicate to Debug Tool that you do not want to debug certain modules.

Using explicit debug mode to load debug data for only specific modules

By default, Debug Tool 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 Debug Tool to load the debug data for the compile units that you want to debug.

In most cases, you can use the SET EXPLICITDEBUG command to enable explicit debug mode; however, in some cases you might need to use the EQAOPTS EXPLICITDEBUG command. Because Debug Tool does not process commands until after it processes the initial load module and all the compile units it contains, if you want Debug Tool to *not* load debug data for compile units in the initial load module, use the EQAOPTS EXPLICITDEBUG command.

When explicit debug mode is active, Debug Tool loads debug data in only the following cases:

- For the compile unit where Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool loads a load module that you previously specified on an LDD command.
 - When Debug Tool loads a load module that you previously specified on a NAMES INCLUDE LOADMOD command.
 - It is a load module for which Debug Tool 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 (*).

Debug Tool does not support the SET DISASSEMBLY ON command in explicit debug mode. When explicit debug mode is active, Debug Tool 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 Debug Tool to not load debug data for these load modules or compile units.

You can use the NAMES EXCLUDE command to indicate to Debug Tool 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 Debug Tool 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 Debug Tool 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, Debug Tool 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, Debug Tool will not exclude them even if you specified them in a previous NAMES EXCLUDE command.

In all of the above situations, Debug Tool loads debug data as required and these modules might become known to Debug Tool.

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

Related references

"NAMES command" in *Debug Tool 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 *Debug Tool 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 Debug Tool; therefore, you cannot use the NAMES command to indicate to Debug Tool 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 *Debug Tool Reference and Messages* or *Debug Tool Customization Guide*.

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

Related references

"NAMES command" in *Debug Tool Reference and Messages*

Using delay debug mode to delay starting of a debug session

By default, Debug Tool 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 Debug Tool recognizes prog1.

Debug Tool is in the dormant state during the delay debug mode and monitors only a few events. When Debug Tool recognizes prog1, Debug Tool 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 Debug Tool 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, Debug Tool 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 option B of the Debug Tool Utilities: Delay Debug Profile.

You enable delay debug mode by using the EQAOPTS DLAYDBG command. By default, delay debug mode is NO. When delay debug mode is enabled, you can specify these additional commands:

DLAYDBGEND

You can use this command to indicate whether you want Debug Tool to monitor condition events in the delay debug mode.

The default is DLAYDBGEND,ALL.

DLAYDBGDSN

Delay debug profile data set naming pattern.

The default is *userid*.DLAYDBG.EQAUOPTS.

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 Debug Tool 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 434 for a description of the steps required to use the REPOSITORY option of DLAYDBGXRF

Once Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool Utilities.
- The TEST runtime option in the delay debug profile must have PROMPT in the third sub option, for example, TEST(ALL,*,PROMPT,*).

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

Related references

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 *Debug Tool for z/OS Customization Guide*.
- Delay debug mode must be active. For information about setting delay debug mode, see "Using delay debug mode to delay starting of a debug session" on page 431.
- 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 Debug Tool for non-Language Environment-compliant programs, see "Starting Debug Tool for programs that start outside of Language Environment" on page 143.

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 Debug Tool. Therefore, commands such as LIST CALLS provide information only about the current task. Debug Tool 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 Debug Tool 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 Debug Tool session.

Debugging tasks running under a generic user ID by using Terminal Interface Manager

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 started task must be started in repository mode by using the `-r` startup option. See *Starting the Terminal Interface Manager* in the *Debug Tool 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 started 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 TSO user ID and password. The following panel is displayed:

```
DEBUG TOOL TERMINAL INTERFACE MANAGER

EQAY001I Terminal TRMLU004 connected for user USRT001
EQAY001I Ready for Debug Tool

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":

```
DEBUG TOOL 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, Debug Tool 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 Debug Tool. Debug Tool 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 Debug Tool user interface.

Part 8. Appendixes

Appendix A. Data sets used by Debug Tool

Debug Tool 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 file. The data set must be a single file, not a concatenation of files. Debug Tool 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. Debug Tool 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 Debug Tool.

As this data set might be read many times by Debug Tool, 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 *Debug Tool Reference and Messages* or the *Debug Tool 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` command¹⁰, which requires Debug Tool 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 HFS file. Debug Tool uses it to show you the program as it is executing.

10. 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.

The COBOL compiler stores the name of the listing data set inside the load module. Debug Tool uses this data set name to access the listing.

Debug Tool 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, program listings do not need to be saved. The debug data and the source code is saved in the program object.

The VS COBOL II compilers do not store the name of the listing data set. Debug Tool 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 Debug Tool, 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

Debug Tool 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 Debug Tool. 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 file. Debug Tool 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. Debug Tool 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 Debug Tool.

Because this data set might be read many times by Debug Tool, 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 *Debug Tool Reference and Messages* or the *Debug Tool 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. Debug Tool 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. Debug Tool looks for the listing in a data set with the name in the form of `userid.cuname.LIST`.

Debug Tool 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 Debug Tool, 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 Debug Tool. 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 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 name 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 name 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, a HFS 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. Debug Tool uses this data set name to access the debug information, unless you provide another data set name as described in Appendix B, "How does Debug Tool locate source, listing, or separate debug files?," on page 445.

Because this data set might be read many times by Debug Tool, 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, Debug Tool 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 Debug Tool commands that customize your session. You can use it, for example, to change the default screen colors set by Debug Tool. 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 165.

A CICS region must have read authorization to the preferences file.

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 *Debug Tool Reference and Messages* or the *Debug Tool Customization Guide*. If a global preferences file exists, Debug Tool 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.

Commands file

This data set contains Debug Tool 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, Debug Tool 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 COMMANDSDSN 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 182.

A CICS region must have read authorization to the commands file.

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 *Debug Tool Reference and Messages* or the *Debug Tool Customization Guide*.

The record format must be either F or FB and the logical record length must be 80.

A CICS region must have read authorization to the EQAOPTS file.

Log file

Debug Tool uses this file to record the progress of the debugging session. Debug Tool 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 Debug Tool writes to this data set, store the log file as a sequential file to relieve any contention for this file.

Debug Tool 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 184.

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 386.

Save settings file (SAVESETS)

Debug Tool uses this file to save and restore, between Debug Tool 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 Debug Tool 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 191.

Save breakpoints and monitor specifications file (SAVEBPS)

Debug Tool uses this file to save and restore, between Debug Tool 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*.DBGT00L.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 Debug Tool 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 191.

Appendix B. How does Debug Tool locate source, listing, or separate debug files?

Debug Tool 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. Debug Tool 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 programs, Debug Tool uses the debug information and the source files that are in the program object.
- For COBOL and PL/I CUs compiled with the SEPARATE suboption of the TEST compiler option, Debug Tool 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, Debug Tool 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 command¹¹
- For C and C++ CUs created and debugged under the following conditions, Debug Tool 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 command (which requires Debug Tool to search for .dbg and source files in a .mdbg file)¹²
- For assembler and LangX COBOL CUs, Debug Tool 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, 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.

Debug Tool 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

11. 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.

12. 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.

- the EQADEBUG DD statement

For C and C++ CUs, Debug Tool 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 command.¹³ For CUs created and debugged under the following conditions, Debug Tool 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 command¹⁴

Debug Tool 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

Debug Tool 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, Debug Tool 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 command (which requires Debug Tool to search for a .dbg file in a .mdbg file)¹⁵

Debug Tool 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

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.

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.

15. 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 each type of file (source, listing, separate debug file, .dbg, or .mdbg), Debug Tool 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 Debug Tool finds that file.

For .dbg and .mdbg files, Debug Tool does not search for the source until it finds a valid .dbg or .mdbg file.

How does Debug Tool locate source and listing files?

Debug Tool 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 Debug Tool 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 Debug Tool locate COBOL and PL/I separate debug file files?

Debug Tool might read from a Enterprise COBOL for z/OS Version 4 compiler (and earlier) or PL/I separate debug file more than once but it always reads the separate debug file from the same data set. After Debug Tool locates a valid separate debug file, you cannot direct Debug Tool to a different separate debug file. When the CU first appears, Debug Tool looks for the separate debug file in the following order:

1. SET SOURCE command
2. 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.
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 statement

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 Debug Tool 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 Debug Tool locates a valid EQALANGX file, you cannot direct Debug Tool 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 Debug Tool), Debug Tool 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 an 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 an EQADEBUG DD statement is not specified, the data set name might be modified by the EQAUEDAT user exit.
5. the EQADEBUG DD statement

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 Debug Tool does not find the EQALANGX file.
- If the CU is not known to Debug Tool when the LDD command is entered and then Debug Tool runs the LDD after the CU becomes known to Debug Tool.

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 Debug Tool 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 command¹⁶, Debug Tool needs the source file and the .dbg file. The following list describes how Debug Tool searches for those files:

- Debug Tool reads the source files for a CU each time it needs to display the source code. Debug Tool 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, Debug Tool 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.

¹⁶ 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.

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.
- Debug Tool might read the .dbg file more than once, but it always reads this file from the same data set. After Debug Tool locates this file and validates its contents with the load module being debugged, you cannot redirect Debug Tool to search a different file. Debug Tool 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, Debug Tool 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 Debug Tool locate the C/C++ .mdbg file?

For the following conditions, Debug Tool 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.
 - 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.

Debug Tool might read the .mdbg file more than once, but it always reads this file from the same data set. After Debug Tool locates this file and validates its contents with the load module being debugged, you cannot redirect Debug Tool to search a different file. Debug Tool 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 Debug Tool 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 Debug Tool Utilities

These examples show you how to use Debug Tool 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 Debug Tool Utilities.
3. Compiling or assembling your program by using Debug Tool Utilities. If you do not use Debug Tool 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 20. Names and attributes to use when you create your own data sets.

Data set name	LRECL	BLKSIZE	RECFM	DSORG
<i>prefix</i> .SAMPLE.COBOL	80	*	FB	PO
<i>prefix</i> .SAMPLE.PLI	80	*	FB	PO
<i>prefix</i> .SAMPLE.C	80	*	FB	PO
<i>prefix</i> .SAMPLE.ASM	80	*	FB	PO
<i>prefix</i> .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	<i>prefix</i> .SAMPLE.COBOL(WPP1)	COBOL source code
EQAWPP3	<i>prefix</i> .SAMPLE.PLI(WPP3)	PL/I source code
EQAWPP4	<i>prefix</i> .SAMPLE.C(WPP4)	C source code
EQAWPP5	<i>prefix</i> .SAMPLE.ASM(WPP5)	Assembler source code
EQAWSU1	<i>prefix</i> .SAMPLE.DTSF(WSU1)	setup file for EQAWPP1
EQAWSU3	<i>prefix</i> .SAMPLE.DTSF(WSU3)	setup file for EQAWPP3
EQAWSU4	<i>prefix</i> .SAMPLE.DTSF(WSU4)	setup file for EQAWPP4
EQAWSU5	<i>prefix</i> .SAMPLE.DTSF(WSU5)	setup file for EQAWPP5

Starting Debug Tool Utilities

To start Debug Tool Utilities, do one the following options:

- If Debug Tool Utilities was installed as an option on an existing ISPF panel, then select that option.
- If Debug Tool Utilities data sets were installed as part of your log on procedure, enter the following command from ISPF option 6:

```
EQASTART
```

- If Debug Tool Utilities was installed as a separate application, enter the following command from ISPF option 6:

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

The Debug Tool 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 Debug Tool 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 Debug Tool Utilities” on page 452:
 - 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 Debug Tool Utilities” on page 452:
 - b. Replace &EQAPRFX. with the prefix your EQAW (Debug Tool) library.
 - c. Replace &CEEPRFX. with the prefix your CEE (Language Environment) library.
 - d. Enter "e" in Cmd field next to CMD5 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. Debug Tool is started and the Debug Tool window is displayed. Enter any valid Debug Tool commands to verify that you can debug the program. Enter "qq" in the command line to stop Debug Tool and close the Debug Tool 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 Debug Tool Utilities" on page 452:
 - 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 = WSU5Press Enter.
5. If you ran the steps beginning in step 1 of topic "Run the program in foreground" on page 455 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 Debug Tool Utilities" on page 452.
 - b. Replace &EQAPRFX. with the prefix your EQAW (Debug Tool) library.
 - c. Replace &CEEPRFX. with the prefix your CEE (Language Environment) library.
6. Enter "e" in the Cmd field next to CMDSD 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. Debug Tool JCL Wizard

Debug Tool JCL Wizard introduction

By using this ISPF edit macro, you can modify a JCL or procedure member and create statements to invoke Debug Tool in various environments. By using the ISPF edit macro, you can also create statements to invoke Debug Tool using the Terminal Interface Manager, Dedicated Terminal, or Remote GUI available with RD/z and PD Tools Studio.

By using the Debug Tool JCL Wizard, you can build control statements to complete following tasks:

- Invoke Debug Tool, accessing the Terminal Interface Manager, Dedicated Terminal, or Remote GUI.
- Invoke Debug Tool for Language Environment (LE) or non-Language Environment (non-LE) programs.
- Include a request to invoke the Automonitor.
- Include a request to set AT ENTRY breakpoints.
- Include a request to SET WARN OFF or ON.
- Define the libraries to search for Debug Tool source and debug information.

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.

- Provide a panel to enter the programs which require LDD statements.
- Request Code Coverage invocation with or without an interactive Debug session.
- Request a Delayed Debug session.
- Remove Debug Tool statements.
- Show comments depicting how to access subprogram source and debug information before being loaded into storage.

The user identifies the location of the statements by using an **A** (after) or **B** (before) line command. If no line command is supplied and more than one program was identified in the JCL or procedure member, the wizard lists all programs. You can select the program that you want to debug from the list. If you want to provide an override value of procedure step, specify an **A** or **B** line command.

The Debug Tool JCL Wizard can create in-stream data. Therefore, it works with a procedure member for JES2 under z/OS 1.13 or later, and with JES3 under z/OS 2.1 or later. If you do not run Debug Tool JCL Wizard in one of the environments that are described above, submitting JCL invoking procedures with in-stream control statements fails.

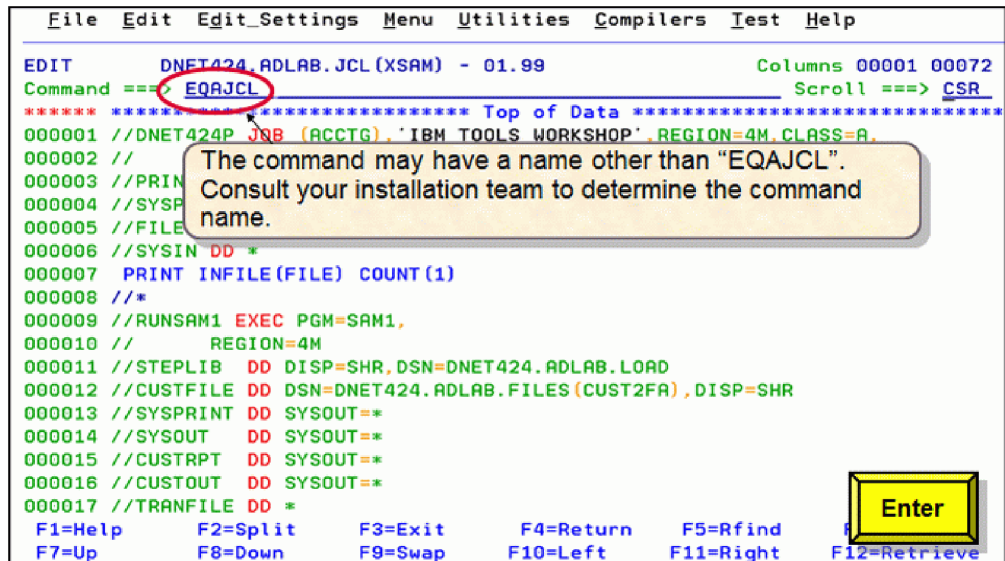
Generation of the EQAMDBG DD statement for C/C++ mdbg files is not supported.

Debug Tool JCL Wizard use cases

Help information

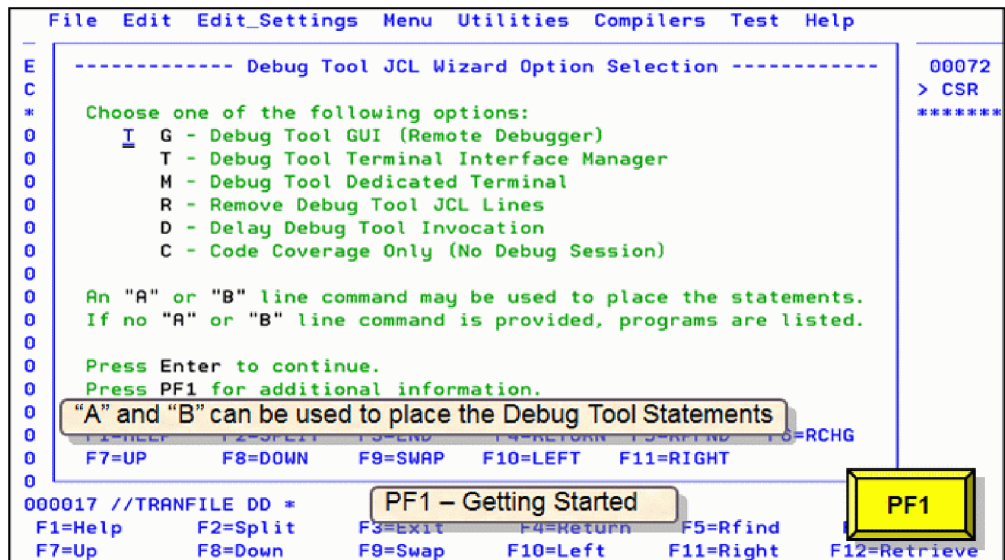
To see the help information, complete the following steps:

1. You can edit or view JCL, a JCL procedure, or an include member in ISPF to invoke Debug Tool JCL Wizard. Your installer can customize your environment to use another name rather than EQAJCL, such as DEBUG. In this use case, the name EQAJCL is used to invoke the Debug Tool JCL Wizard.



```
File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT      DNET424.ADLAB.JCL (XSAM) - 01.99          Columns 00001 00072
Command ==> EQAJCL                               Scroll ==> CSR
***** Top of Data *****
000001 //DNET424P JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //
000003 //PRINT DD SYSOUT=*
000004 //SYSPRINT DD SYSOUT=*
000005 //FILE DD SYSOUT=*
000006 //SYSIN DD *
000007 PRINT INFILE(FILE) COUNT(1)
000008 /**
000009 //RUNSAM1 EXEC PGM=SAM1,
000010 //          REGION=4M
000011 //STEPLIB DD DISP=SHR, DSN=DNET424.ADLAB.LOAD
000012 //CUSTFILE DD DSN=DNET424.ADLAB.FILES(CUST2FA), DISP=SHR
000013 //SYSPRINT DD SYSOUT=*
000014 //SYSOUT DD SYSOUT=*
000015 //CUSTRPT DD SYSOUT=*
000016 //CUSTOUT DD SYSOUT=*
000017 //TRANFILE DD *
F1=Help    F2=Split  F3=Exit    F4=Return  F5=Rfind   F6=Rchg
F7=Up      F8=Down    F9=Swap    F10=Left   F11=Right  F12=Retrieve
Enter
```

2. The Debug Tool JCL Wizard Option Selection panel is displayed as follows. To see the **Getting Started** help panel, select **PF1** from this panel.



```
File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool JCL Wizard Option Selection -----
E
C
* Choose one of the following options:
  I G - Debug Tool GUI (Remote Debugger)
  T - Debug Tool Terminal Interface Manager
  M - Debug Tool Dedicated Terminal
  R - Remove Debug Tool JCL Lines
  D - Delay Debug Tool Invocation
  C - Code Coverage Only (No Debug Session)
0
0 An "A" or "B" line command may be used to place the statements.
0 If no "A" or "B" line command is provided, programs are listed.
0
0 Press Enter to continue.
0 Press PF1 for additional information.
0 "A" and "B" can be used to place the Debug Tool Statements
0
0 F1=Help    F2=Split  F3=Exit    F4=Return  F5=Rfind   F6=Rchg
0 F7=UP      F8=DOWN    F9=SWAP    F10=LEFT   F11=RIGHT
0
000017 //TRANFILE DD *
F1=Help    F2=Split  F3=Exit    F4=Return  F5=Rfind   F6=Rchg
F7=Up      F8=Down    F9=Swap    F10=Left   F11=Right  F12=Retrieve
PF1
```

3. To invoke Debug Tool, the Debug Tool JCL Wizard needs to create the necessary JCL statements. The following options are provided:

```

Debug Tool Wizard - Getting Started
More: +
The EQAJCL command will provide panels to enter information to create JCL
or procedure lines to invoke Debug Tool, remove Debug Tool lines, or
create Code Coverage information.

The following parameters may be entered:

o EQAJCL G - Invoke Debug Tool using the Remote Graphical User Interface
o EQAJCL T - Invoke Debug Tool using the Terminal Interface Manager
o EQAJCL M - Invoke Debug Tool using a Dedicated Terminal
o EQAJCL R - Remove Debug Tool JCL lines added by this tool
o EQAJCL D - Invoke Debug Tool using Delay Debug mode Invocation
o EQAJCL C - Code Coverage Only - No Debug Tool Invocation

The EQAJCL command can be issued when editing or viewing a JCL member or a
procedure member. The macro will create instream DD statements. Instream
Continue reviewing "Getting Started" with Enter key
F7=PrvPage F8=NxtPage F9=Swap F10=PrvPage F11=NxtPage F12=
Enter

```

4. You can invoke the Debug Tool JCL Wizard when you view or edit a JCL member, a procedure member, or an include member. Debug Tool JCL Wizard can create instream JCL statements. For procedures or included members, instream JCL statements such as //SYSIN DD * are valid only when you use JES2 systems with z/OS V1.R13 or later, or JES3 systems with z/OS V2.1.0 or later.

```

Debug Tool Wizard - Getting Started
More: - +
procedure member. The macro will create instream DD statements. Instream
DD statements for procedures are permitted with JES2 for z/OS V1.R3.0, and
for JES3 with z/OS V2.R1.0.

Changes made from the keyboard are not processed by the EQAJCL command
unless you press the ENTER key. If you are modifying JCL or procedure
lines, press ENTER, then enter the EQAJCL command to achieve the correct
results. Otherwise, the most recent changes will not be recognized by the
command.

The Debug Tool JCL Wizard has two methods used to determine where to place
the Debug Tool invocation statements:

o An "A" or "B" line command can be used to place the statements. For
this method, enter the line "A" or "B" line command, then enter the
EQAJCL command with the appropriate parameter.

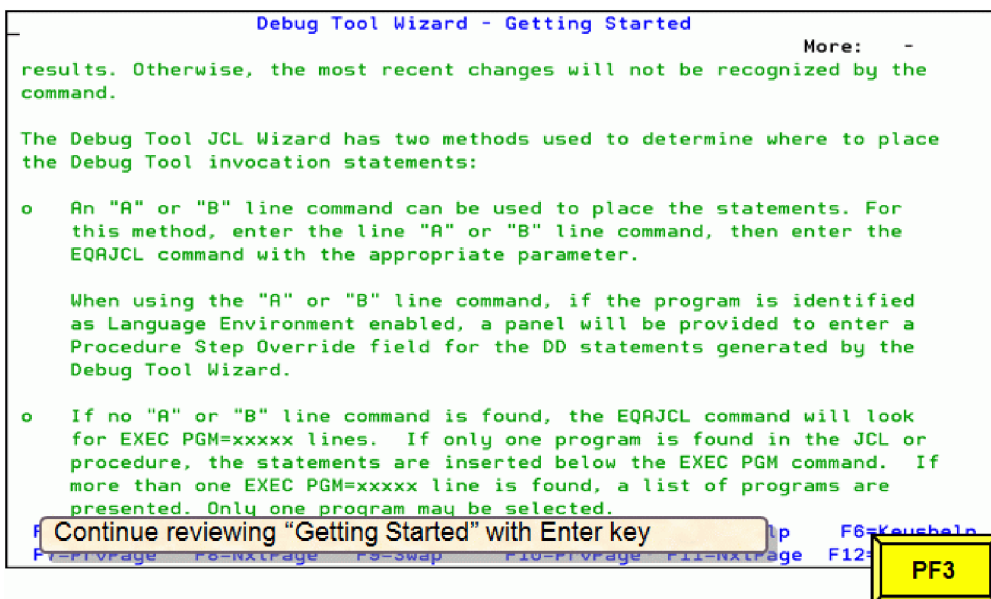
When using the "A" or "B" line command, if the program is identified
as Language Environment enabled, a panel will be provided to enter a
Procedure Step Override field for the DD statements generated by the
Continue reviewing "Getting Started" with Enter key
F7=PrvPage F8=NxtPage F9=Swap F10=PrvPage F11=NxtPage F12=
Enter

```

5. To choose where new JCL lines are placed, use an **A** or **B** line command. If the **A** or **B** line commands are not used, a list of job steps is displayed for your selection.

Note: The Debug Tool JCL Wizard does not create the JCL for all scenarios. For example, if you want to debug multiple job steps in a JCL member, you can create the JCL for the first Debug Tool invocation by using the Debug Tool JCL Wizard. However, you need to manually code the JCL to invoke Debug Tool for

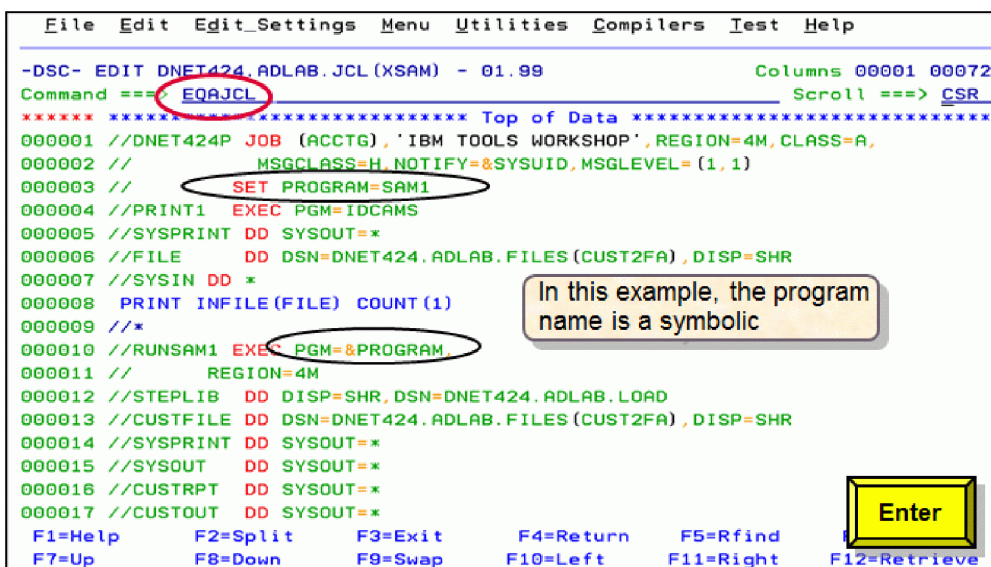
the second step.



Debug a Language Environment program by using the Terminal Interface Manager

To debug a Language Environment program by using the Debug Tool JCL wizard, you can use the Terminal Interface Manager. To do the job, complete the following steps:

1. In the following panel, the program name is a symbolic name that is defined by a SET statement. The Debug Tool JCL Wizard can process symbolic program names correctly. To invoke the Debug Tool JCL Wizard Options panel, type EQAJCL.



2. Select the Terminal Interface Manager option from the following panel.


```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool JCL Wizard Option Selection -----
00072
> CSR
*****
* Choose one of the following options:
0  I G - Debug Tool GUI (Remote Debugger)
0  T - Debug Tool Terminal Interface Manager
0  M - Debug Tool Dedicated Terminal
0  R - Remove Debug Tool JCL Lines
0  D - Delay Debug Tool Invocation
0  C - Code Coverage Only (No Debug Session)
0
0  An "A" or "B" line command may be used to place the statements.
0  If no "A" or "B" line command is provided, programs are listed.
0
0  Press Enter to continue.
0  Press PF1 for additional information.
0
0  F1=HELP   F2=SPLIT  F3=END     F4=RETURN  F5=RPFND   F6=RCHG
0  F7=UP     F8=DOWN   F9=SWAP   F10=LEFT  F11=RIGHT
0
000017 //TRANFILE DD *
F1=Help   F2=Split  F3=Exit   F4=Return  F5=Rfind
F7=Up     F8=Down   F9=Swap   F10=Left  F11=Right  F12=Retrieve
Enter

```

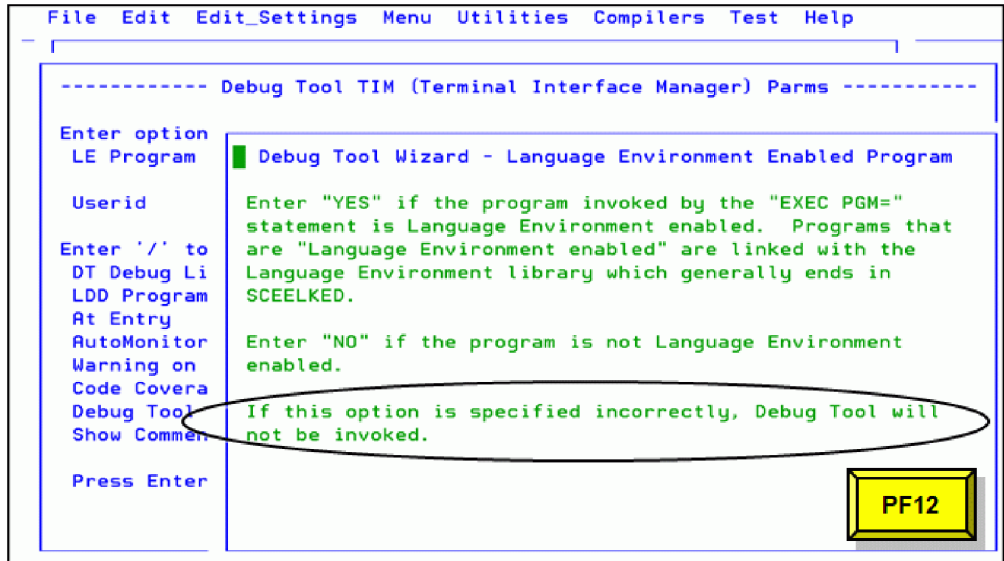
3. All fields have field help available. To view the field help that is associated with the LE Program field, place the cursor in the LE Program field and press PF1.

```

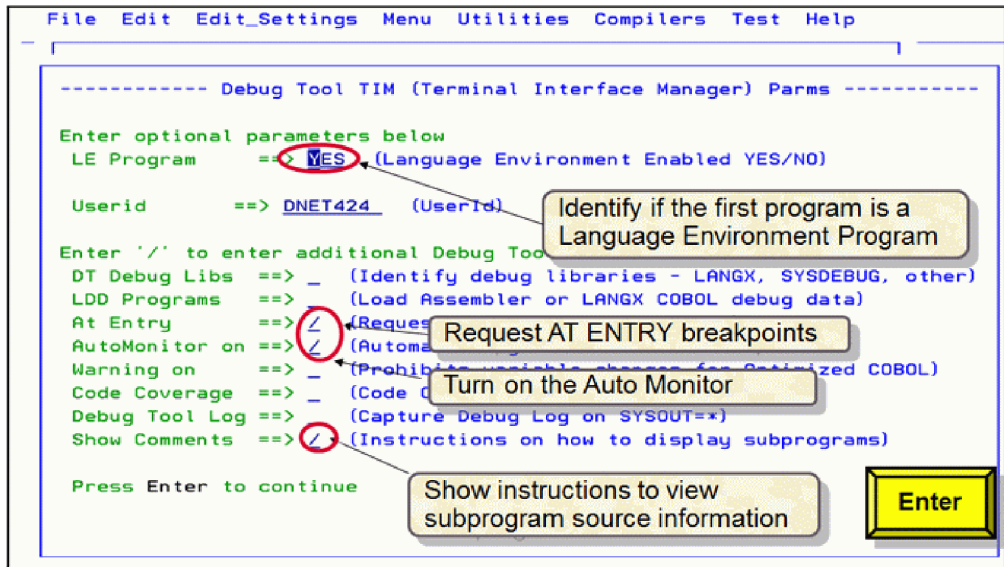
File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager) Parms -----
Enter optional parameters below
LE Program == YES (Language Environment Enabled YES/NO)
Userid ==> DNET424 (UserId)
Enter '/' to enter additional Debug Tool information
DT Debug Libs ==> - (Identify debug libraries - LANGX, SYSDEBUG, other)
LDD Programs ==> - (Load Assembler or LANGX COBOL debug data)
At Entry ==> / (Request to stop in subprograms)
AutoMonitor on ==> / (Automatically monitor variables)
Warning on ==> - (Prohibits variable changes for Optimized COBOL)
Code Coverage ==> - (Code Coverage)
Debug Tool Log ==> - (Capture Debug Log on SYSQUT=x)
Show Comments ==> / (I
Every field has Field Help Available
Press Enter to continue
PF1

```

4. Enter YES if the program invoked by the "EXEC PGM=" statement is Language Environment enabled. Enter NO if the program is not Language Environment enabled. If the LE Program field is set incorrectly, Debug Tool is not invoked.



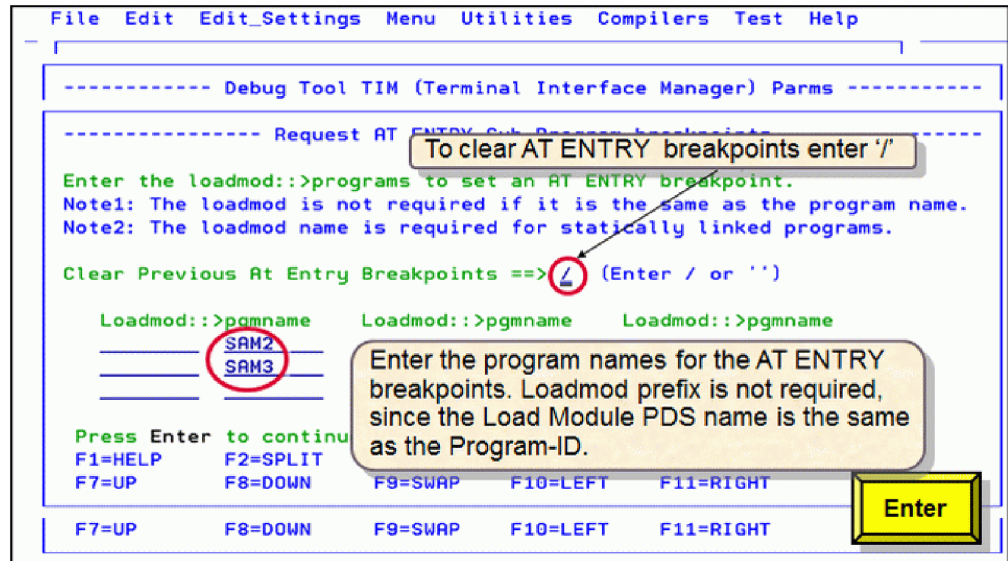
5. Type YES in the LE Program field if the program is a Language Environment enabled program. If you want to set breakpoints at subprograms using the AT ENTRY command, and request the Automonitor, specify a forward slash (/) for **At Entry** and **Automonitor on**.



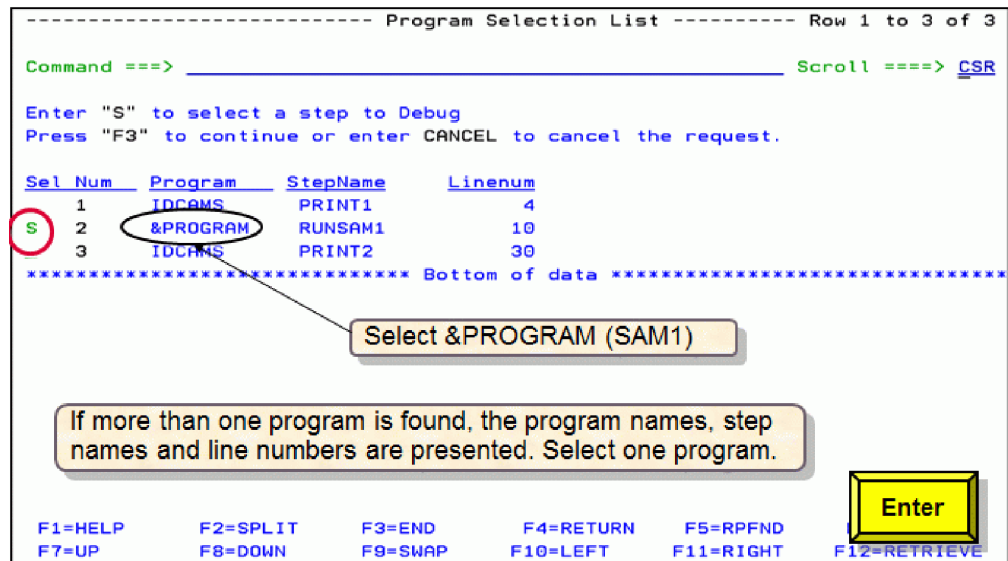
6. After you choose to set AT ENTRY breakpoints for subprograms, the following panel is displayed. If you are saving your settings or using the remote debugger, AT ENTRY breakpoints are remembered between debug sessions. To clear previous AT ENTRY Breakpoints before setting these breakpoints, enter the forward slash (/) in the **Clear Previous AT ENTRY Breakpoints** line.

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 that 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 will need the load module name of the main program, and the program name of the subprogram name where you want to stop.



7. If more than one EXEC PGM statement is found in the JCL or procedure, and you did not specify the A or B line command, a list of job steps is displayed. Choose the program you want to debug. You can choose only one step.



8. Debug Tool JCL Wizard has generated the appropriate statements to invoke Debug Tool. The first and last generated lines are comment lines. Do not modify these comment lines. The //CEEOPTS statement defines the EQACMD command file DD name, and the VTAM% userid information to invoke the Terminal Interface Manager.

The **SET LOG OFF** command indicates you do not want to retain your log information. Previous AT ENTRY breakpoints are cleared, and breakpoints for subprograms SAM2 and SAM3 are set.

The comment lines explain how to locate a program, and set breakpoints before invocation of the subprogram for the Terminal Interface Manager.

To initiate this session, start the Terminal Interface Manager, sign on to the Terminal Interface Manager with your userid and password. 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.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL (XSAM) - 01.25 Columns 00001 00072
Command ==> SUBMIT Scroll ==> CSR
000009 //RUNSAM1 EXEC PGM=&PROGRAM,
000010 // REGION=4M
000011 /*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM &PROGRAM
000012 //CEEOPPTS DD * !INVOCATION FOR TERMINAL INTERFACE MANAGER
000013 TEST(,EQACMD,,VTAM%DNET424:)
000014 //EQACMD DD *
000015 SET LOG OFF;
000016 SET AUTO ON BOTH;
000017 SET WARN OFF;
000018 CLEAR AT ENTRY;
000019 AT ENTRY SAM2;
000020 AT ENTRY SAM3;
000021 COMMENT TO VIEW THE SOURCE OF A SUBPROGRAM FOLLOW THE STEPS BELOW;;
000022 COMMENT 1) ENTER A "STEP" COMMAND;
000023 COMMENT 2) ENTER "QUALIFY PROGRAM" PGMNAME;
000024 COMMENT 3) IF THE LOAD MODULE NAME IS NOT THE SAME AS PROGRAM NAME;;
000025 COMMENT -- ENTER "QUALIFY PROGRAM LOADMOD:;>PGMNAME";
000026 /*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM &PROGRAM
000027 //STEPLIB DD DISP=SHR,DSN=&SYSUID..ADLAB.LOAD
000028 //CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
  
```

Annotations in the image:

- Show current and previous variables in Monitor** (points to SET LOG OFF)
- Allow variable change to optimized COBOL Code** (points to SET AUTO ON BOTH)
- Stop at subprograms SAM2 and SAM3** (points to AT ENTRY SAM2 and AT ENTRY SAM3)
- The comments depicting how to view the source of a subprogram** (points to the COMMENT lines)

- To remove the statements that are previously generated by the Debug Tool JCL Wizard, use the **EQAJCL R** command. The first and last lines generated are comment lines. If you modify the comment lines, the statements that are generated by the Debug Tool JCL Wizard cannot be removed properly.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL (XSAM) - 01.25 Columns 00001 00072
Command ==> EQAJCL R Scroll ==> CSR
000009 //RUNSAM1 EXEC PGM=&PROGRAM,
000010 // REGION=4M
000011 /*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM &PROGRAM
000012 //CEEOPPTS DD * !INVOCATION FOR TERMINAL INTERFACE MANAGER
000013 TEST(,EQACMD,,VTAM%DNET424:)
000014 //EQACMD DD *
000015 SET LOG OFF;
000016 SET AUTO ON BOTH;
000017 SET WARN OFF;
000018 CLEAR AT ENTRY;
000019 AT ENTRY SAM2;
000020 AT ENTRY SAM3;
000021 COMMENT TO VIEW THE SOURCE OF A SUBPROGRAM FOLLOW THE STEPS BELOW;;
000022 COMMENT 1) ENTER A "STEP" COMMAND;
000023 COMMENT 2) ENTER "QUALIFY PROGRAM" PGMNAME;
000024 COMMENT 3) IF THE LOAD MODULE NAME IS NOT THE SAME AS PROGRAM NAME;;
000025 COMMENT -- ENTER "QUALIFY PROGRAM LOADMOD:;>PGMNAME";
000026 /*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM &PROGRAM
000027 //STEPLIB DD DISP=SHR,DSN=&SYSUID..ADLAB.LOAD
000028 //CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
  
```

Annotation in the image:

- Remove Debug Tool commands** (points to the SET LOG OFF, SET AUTO ON BOTH, and SET WARN OFF lines)

- The JCL statements that are generated by Debug Tool JCL Wizard are removed.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424.ADLAB.JCL (XSAM) - 01.25 Columns 00001 00072
Command ==> Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET424Q JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //          SET PROGRAM=SAM1
000004 //PRINT1 EXEC PGM=IDCAMS
000005 //SYSPRINT DD SYSOUT=*
000006 //FILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA), DISP=SHR
000007 //SYSIN DD *
000008 PRINT INFILE(FILE) COUNT(1)
000009 //RUNSAM1 EXEC PGM=&PROGRAM,
000010 //          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 COMMENT)
000019 *-----
  
```

The Debug Tool JCL Wizard JCL statements are removed

Debug a Language Environment program with the Remote GUI by using the A line command with a Procedure Step Override

You can use the remote debugger, or GUI to debug Enterprise COBOL, COBOL for MVS and VM, Enterprise PL/I, later versions of C/C++ and assembler. To invoke the Debug Tool remote debugger with a line command to indicate where the JCL is placed, complete the following steps:

- In the following panel, issue the **EQAJCL G** command to bypass the options screen, and request a debug session with remote debugger. Then, place the generated statements after the line that contains the TESTSAM1 EXEC statement at line 5.

```

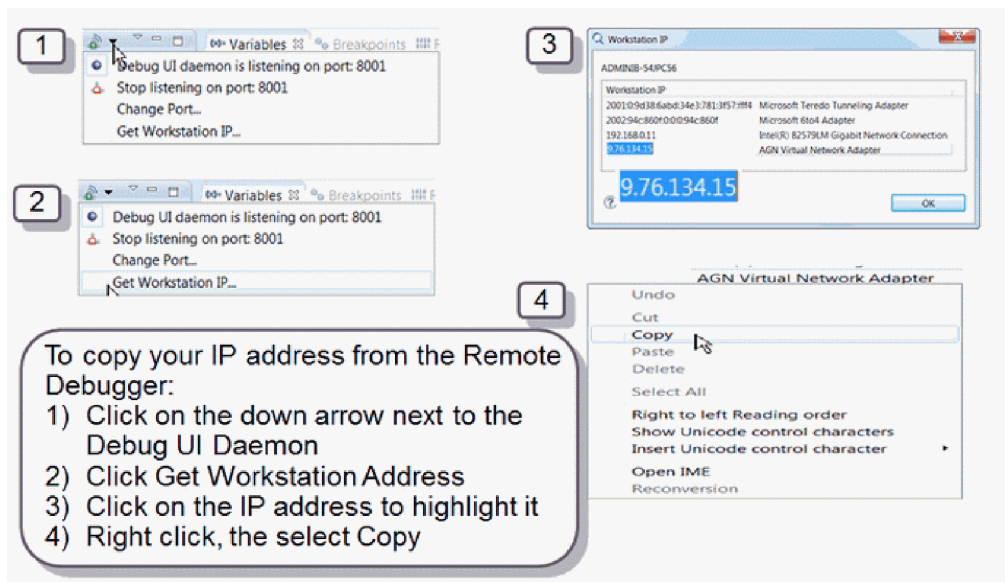
File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424.ADLAB.JCL (XSAMG) - 01.09 Debug statements removed
Command ==> EQAJCL G Debug the SAM1 program using the GUI Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET424A JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //          JCLLIB ORDER=DNET424.ADLAB.JCL
000004 /**
000005 //TESTSAM1 EXEC TESTSAM1
000006 /**
000007 //PRINT2 EXEC PGM=IDCAMS
000008 //SYSPRINT DD SYSOUT=*
000009 //FILE DD DSN=&SYSUID..ADLAB.
000010 //SYSIN DD *
000011 PRINT INFILE(FILE) COUNT(1)
***** ***** Bottom *****
  
```

Alternate Method:
Use the A (After) or B (Before) line command to position where the JCL lines will be placed.

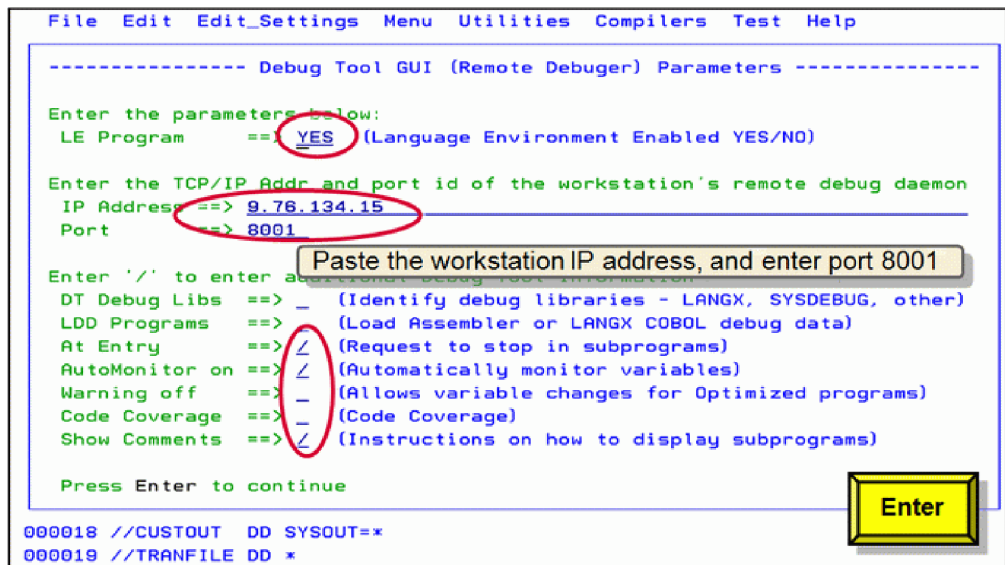
Enter

- Then, you need your IP Address from your workstation. To locate this IP address, start the remote debugger, and open the DEBUG perspective. Note

your port number. It is generally 8001.



3. Paste your IP address into the IP address field. Enter the port number that is provided by the remote debugger. You can optionally choose to add AT ENTRY breakpoints, set the Automonitor on, and show comments on how to display subprograms before invocation.



4. 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.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
----- Request AT ENTRY Sub-Program breakpoints -----

Enter the loadmod::>programs to set an AT ENTRY breakpoint.
Note1: The loadmod is not required if it is the same as the program name.
Note2: The loadmod name is required for statically linked programs.

Clear Previous At Entry Breakpoints ==> \ (Enter / or '')

Loadmod::>pgmname      Loadmod::>pgmname      Loadmod::>pgmname
=====              =====              =====
                     SAM2
                     SAM3

Subprogram names are remembered,
unless you edit another member

Press Enter to continue
F1=HELP      F2=SPLIT      F3=END      F4=IPT View      F5=RFIND
F6=RCHANGE   F7=UP           F8=DOWN    F9=SWAP         F10=>BACK

F1=Help      F2=Split      F3=Exit     F4=Return      F5=Rfind
F7=Up        F8=Down       F9=Swap    F10=Left      F11=Right     F12=Retrieve
Enter

```

5. In this use case, the JCL points to a procedure. If you choose the A line command, you can enter the Procedure Step Override. In this use case, RUNSAM1 is entered for this value.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool - Proc Step Override -----
00072 > CSR *****
C
* Optionally enter a Proc Step Override
0 Proc Step Override ==> RUNSAM1
0
0 Press Enter to continue
0
0 Optionally, you may enter the Procedure Step Override
0
0 F1=HELP      F2=SPLIT      F3=END      F4=IPT View      F5=RFIND
0 F6=RCHANGE   F7=UP         F8=DOWN    F9=SWAP         F10=>BACK
0
000009 /**
A00010 //TESTSAM1 EXEC PROC=TESTSAM1
000011 /**
000012 //PRINT2 EXEC PGM=IDCAMS
000013 //SYSPRINT DD SYSOUT=*
000014 //FILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
000015 //SYSIN DD *
000016 PRINT INFILE(FILE) COUNT(1)
000017 //IDIOPPTS DD *
F1=Help      F2=Split      F3=Exit     F4=Return      F5=Rfind      F6=
F7=Up        F8=Down       F9=Swap    F10=Left      F11=Right     F12=
PF3

```

6. The procedure TESTSAM1 contains a step RUNSAM1, which invokes the SAM1 program. Use the Procedure Step Override to define the EQACMD DD (and its contents) for the RUNSAM1 STEP.

The CEEOPTS DD Statement is generated with the parameter TCPIP, 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 provided on how to view subroutines before invocation.

To start this session, start the remote debugger, and submit the job. If the remote debugger does not depict the initiation of a debug session, verify the job is not waiting for an initiator, or failed with a JCL error.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT DNET424 ADLAB.JCL (XSAMG) 00072
Command ==> SUBMIT > CSR
000005 //TESTSAM1 EXEC TESTSAM1
000006 /*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM
000007 //RUNSAM1.CEEOPTS DD * !INVOCATION FOR REMOTE GUI
000008 TEST(,EQACMD,,TCP(9.76.134.15%8001:))
000009 //RUNSAM1.EQACMD NO *
000010 SET AUTO ON BOTH;
000011 SET WARN OFF;
000012 CLEAR AT ENTRY;
000013 AT ENTRY SAM2;
000014 AT ENTRY SAM3;
000015 /* TO VIEW THE SOURCE OF A SUBPROGRAM FOLLOW THE STEPS BELOW: */
000016 /* 1) LEFT CLICK THE "STEP INTO" ICON */
000017 /* 2) IN THE DEBUG TOOL ENGINE COMMAND PANE, ENTER: */
000018 /* 3) QUALIFY PROGRAM PGMNAME; OR */
000019 /* QUALIFY PROGRAM LOADMOD::>PGMNAME (IF LOADMOD NOT = PGMNAME) */
000020
000021
000022
000023
000024

```

Debug a non-Language Environment program by using the Terminal Interface Manager

You can use the Debug Tool JCL Wizard to invoke non-Language Environment programs. If you do not plan to test non-Language Environment programs, skip this use case.

To debug a non-Language Environment program by using Terminal Interface Manager, complete the following steps:

1. To invoke the Debug Tool JCL Wizard for the terminal interface manager, type **EQAJCL T**.

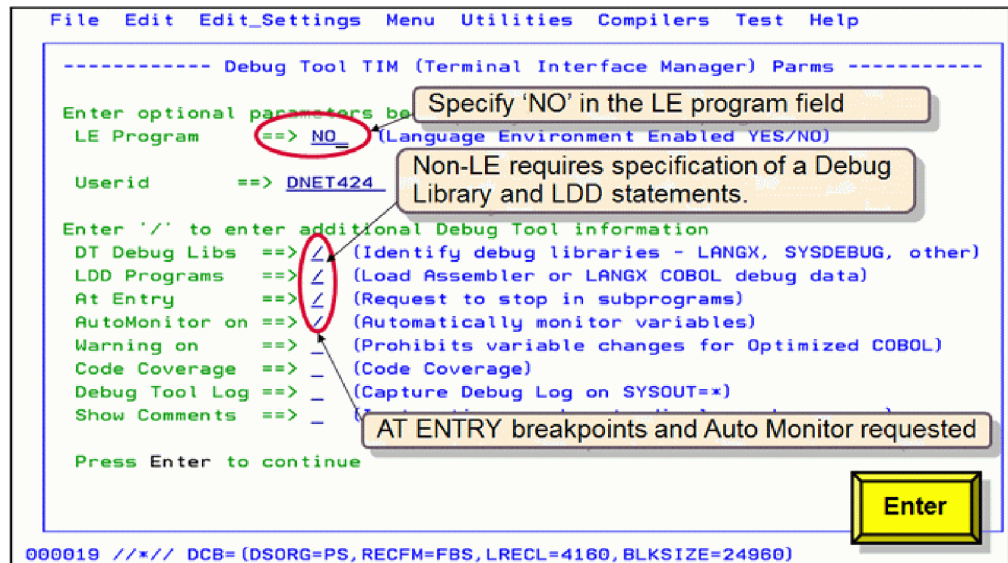
```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL (XASAM1) - 01.99 Invalid command
Command ==> EQAJCL T Scroll ==> CSR
***** Top of Data *****
000001 //DNET424X JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 // MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //ASAM1 EXECEQAM=ASAM1, PARM='ABCD'
000004
000005
000006
000007 //FILEIN DD *,DCB=(LRECL=80)
000008 INPUT RECORD ONE
000009 INPUT RECORD TWO
000010 INPUT RECORD THREE
000011 INPUT RECORD FOUR
000012 INPUT RECORD FIVE
000013 INPUT RECORD SIX
000014 INPUT RECORD SEVEN
000015 INPUT RECORD EIGHT
000016 ABEND <== "ABEND" WILL CAUSE THE SAMPLE PROGRAM TO ABEND
000017 //FILEOUT DD SYSOUT=*
000018 //SYSUDUMP DD SYSOUT=*
***** Bottom of Data *****

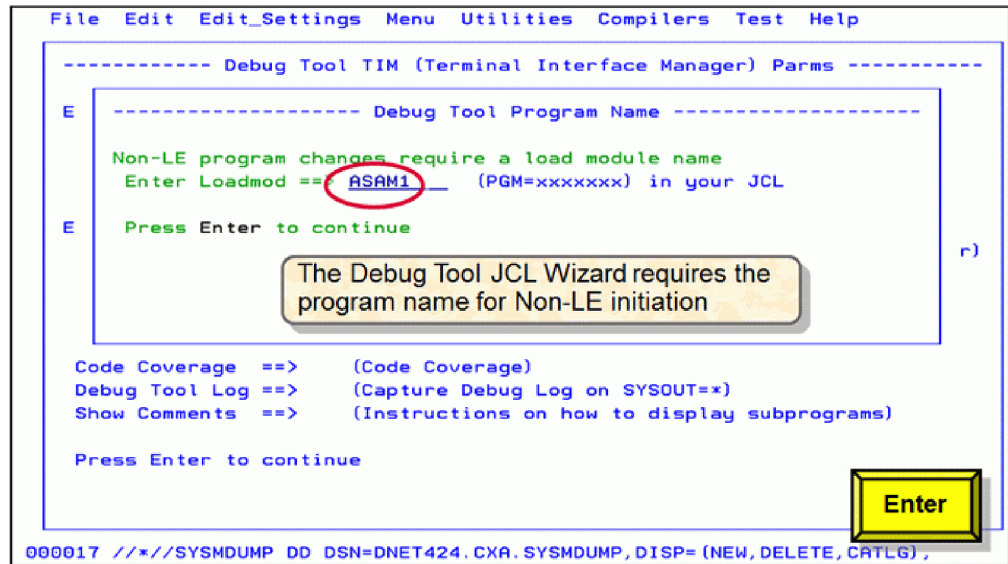
```

2. If you want to debug a non-Language Environment assembler program, enter **NO** in the LE Program field.
For non-Language Environment programs, to identify where the side file information is located, you need debug libraries. You also need a **Load Debug**

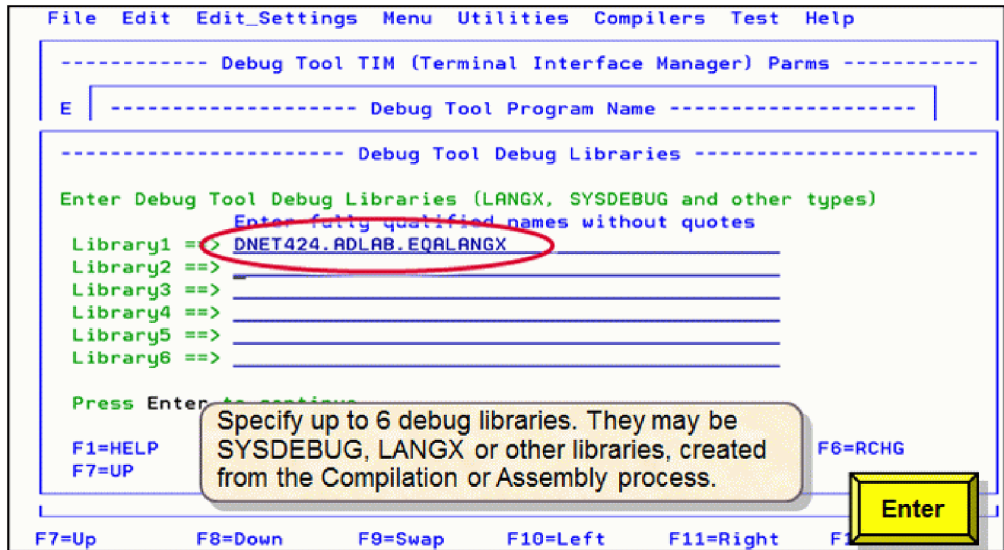
Data or LDD command. To request a panel for this information, enter a forward slash (/) next to **DT Debug libs** and **LDD Programs**. Set **AT ENTRY** breakpoints, and turn on the auto monitor if you want.



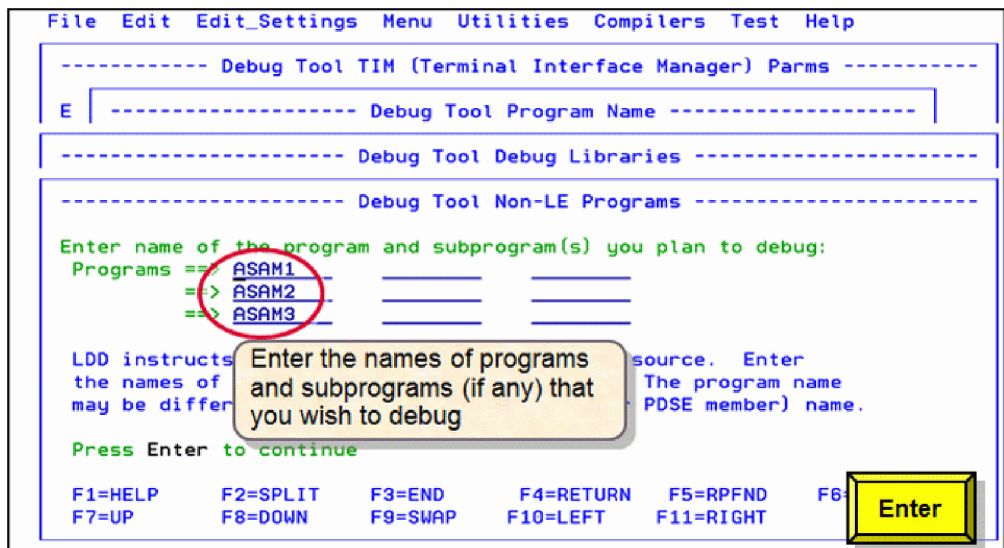
3. The program name must be explicitly identified for non-Language Environment programs. Enter the name of the non-Language Environment program you want to debug. That name is the name shown on the EXEC PGM= statement.



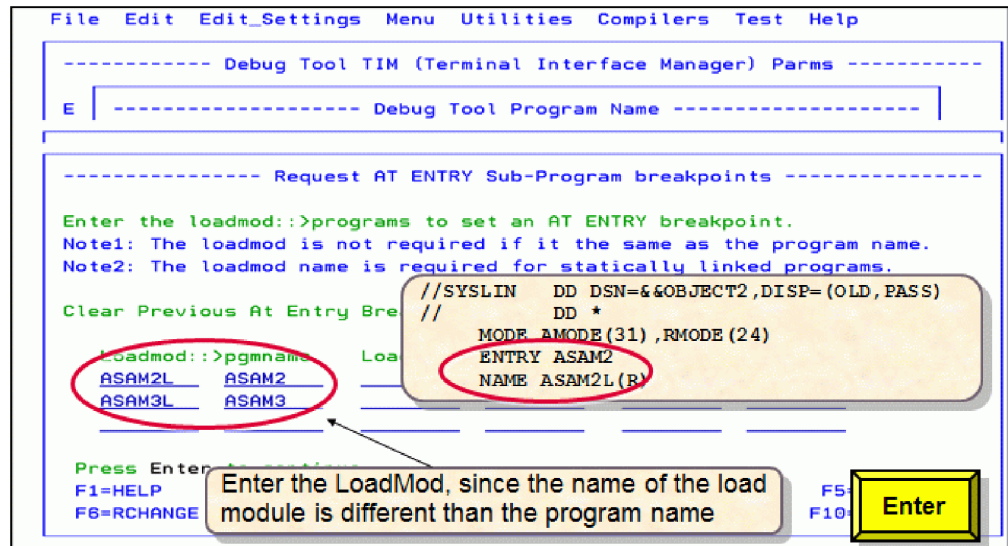
4. In the following panel, you can identify up to six Debug Tool side file data sets. These side files are created during the compilation or assembly process. You can add libraries of various types in this panel. For example, some languages use a LANGX file, others use SYSDEBUG or listing data sets. If you require more than six libraries, modify the JCL after the Debug Tool JCL Wizard creates the appropriate statements.



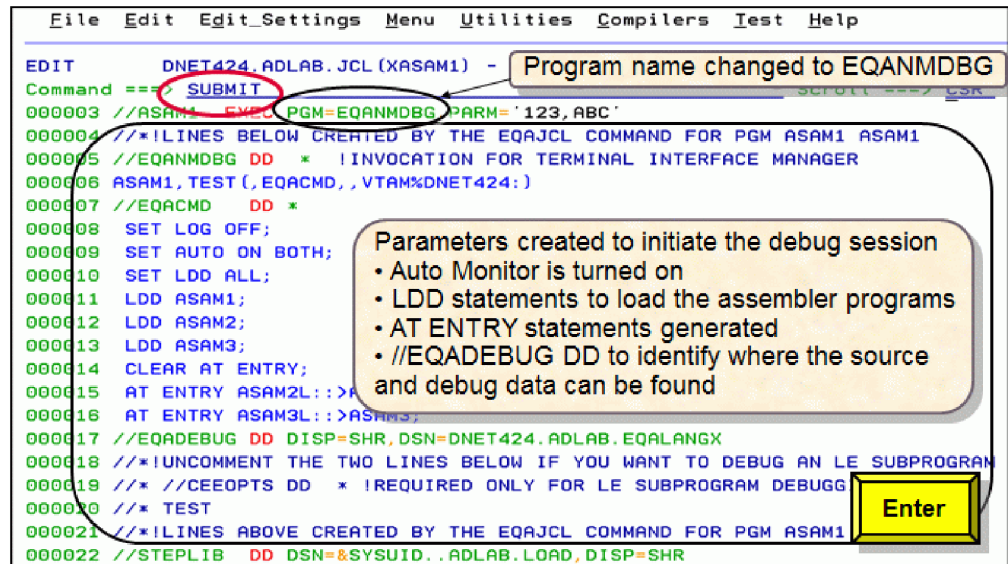
5. Enter the program names and subprogram names that are being debugged. These names are used for setting AT ENTRY breakpoints. Use this panel to identify the non-Language Environment programs before you invoke the Debug tool for the debugging session.



6. When you set breakpoints, generally only the program name is required. However, if the load module name is different from the program name, enter the load module name next to the program name. In this use case, the ASAM2 program load module name is ASAM2L.



7. The JCL to invoke Debug Tool is generated. Note that the program name on line 3 was changed from ASAM1 to EQANMDBG. This program will initiate the debug session, debugging ASAM1, the first program to be invoked. The VTAM%DNET424 requests Debug Tool to invoke the Terminal Interface Manager. This information is passed to the EQANMDBG program via the EQANMDBG DD statement.
 LDD statements are generated for the programs ASAM1, 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.



8. To remove the Debug Tool JCL statements, enter EQAJCL R.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT      DNET424.ADLAB.JCL (XASAM1) - 01.14      Columns 00001 00072
Command ==> EQAJCL R                               Scroll ==> CSR
000003 //ASAM1 EXEC PGM=EQANMDBG
000004 // * LINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM ASAM1 ASAM1
000005 //EQANMDBG DD * !INVOCATION FOR TERMINAL INTERFACE MANAGER
000006 ASAM1, TEST (,EQACMD,,VTAM%DNET424:)
000007 //EQACMD DD *
000008 SET LOG OFF;
000009 SET AUTO ON BOTH;
000010 SET LDD ALL;
000011 LDD ASAM1;
000012 LDD ASAM2;
000013 LDD ASAM3;
000014 CLEAR AT ENTRY;
000015 AT ENTRY ASAM2L::>ASAM2;
000016 AT ENTRY ASAM3L::>ASAM3;
000017 //EQADEBUG DD DISP=SHR,DSN=DNET424.ADLAB.EQALANGX
000018 // * UNCOMMENT THE TWO LINES BELOW IF YOU WANT TO DEBUG AN LE SUBPROGRAM
000019 // * //CEEPTS DD * !REQUIRED ONLY FOR LE SUBPROGRAM DEBUGG
000020 // * TEST
000021 // * LINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM ASAM1
000022 //STEPLIB DD DSN=&SYSUID..ADLAB.LOAD,DISP=SHR

```

Request lines to be removed

Enter

- Note that the PGM=EQANMDBG statement has been modified back to the original program name, ASAM1.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT      DNET424.ADLAB.JCL (XASAM1) - 01.99      Columns 00001 00072
Command ==>                                         Scroll ==> CSR
***** Top of Data *****
000001 //DNET424X JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //ASAM1 EXEC PGM=ASAM1 PARM='ABCD',
000004 // COND=(4,LT)
000005 //STEPLIB DD DSN=&SYSUID..ADLAB.LOAD,DISP=SHR
000006 //FILEIN DD *,DCB=(LRECL=80)
000007 INPUT RECORD ONE
000008 INPUT RECORD TWO
000009 INPUT RECORD THREE
000010 INPUT RECORD FOUR
000011 INPUT RECORD FIVE
000012 INPUT RECORD SIX
000013 INPUT RECORD SEVEN
000014 INPUT RECORD EIGHT
000015 // * ABEND <=== "ABEND" WILL CAUSE THE SAMPLE PROGRAM TO ABEND
000016 //FILEOUT DD SYSOUT=*
000017 //SYSUDUMP DD SYSOUT=*
F1=Help      F2=Split      F3=Exit      F4=Return      F5=Rfind      F6=Rchange
F7=Up        F8=Down       F9=Swap     F10=Left     F11=Right    F12=Retrieve

```

JCL lines removed and program name changed back to ASAM1

Debug a Language Environment DB2 program by using the Remote GUI


You can generate the Debug Tool JCL statements to debug a DB2 batch program by using the remote GUI. To do the job, complete the following steps:

- Enter EQAJCL G to create the required statements.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(TRADER) - 01.52          Columns 00001 00072
Command ==> EQAJCL G                               Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET4240 JOB (JIM), 'FA UTILITY',
000002 //          NOTIFY=DNET424,
000003 //          MSGCLASS=A, REGION=0M,
000004 //          CLASS=A
000005 //TRADEDB2 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT), REGION=4M
000006 //SYSTSIN DD *
000007 DSN SYSTEM(DSNC)
000008 RUN PLAN(TRADERIC) -
000009 PROGRAM( TRADERD )
000010 END
000011 //STEPLIB DD DSN=DNET424.TRADER.LOAD, DISP=SHR
000012 //          DD DSN=DB2.V10.SDSNLOAD, DISP=SHR
000013 //DBRMLIB DD DISP=SHR, DSN=DNET424.TRADER.DBRMLIB
000014 //TRANSACT DD DISP=SHR, DSN=DNET424.TRADER.BATCH.TRANFILE
000015 //REPOUT DD SYSOUT=*
000016 //TRANREP DD SYSOUT=*
000017 //SYSTSPRT DD SYSOUT=*
000018 //SYSPRINT DD SYSOUT=*
000019 //SYSUDUMP DD SYSOUT=*

```



- The program is a Language Environment program, therefore, type **YES** to the LE Program field. The **Automonitor** is not needed for the remote debugger when debugging Enterprise COBOL or PL/I programs. You can monitor variables by right-clicking the variables pane, and requesting filter locals, then select **Automonitor current** or **Automonitor Previous**.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
Enter the parameters below:
LE Program ==> YES (Language Environment Enabled YES/NO)


Enter the TCP/IP Addr and port id of the workstation's remote debug daemon
IP Address ==> 9.76.134.15
Port ==> 8001

Enter '/' to enter additional Debug Tool information
DT Debug Libs ==> _ (Identify debug libraries - LANGX, SYSDEBUG, other)
LDD Programs ==> _ (Load Assembler or LANGX COBOL debug data)
At Entry ==> _ (Request to stop in subprograms)
AutoMonitor on ==> _ (Automatically monitor variables)
Warning off ==> _ (Allows variable changes for Optimized programs)
Code Coverage ==> _ (Code Coverage)
Show Comments ==> _ (Show Comments for all programs)

Press Enter to
000018 //SYSPRINT DD SYSOUT=*
000019 //SYSUDUMP DD SYSOUT=*

```

The Auto Monitor is not needed with the GUI with Language Environment programs. Use "Filter Locals" in the GUI Variable pane to set the monitor variables in the GUI.



- The JCL statements were generated for the batch DB2 program. This process is identical to a non-DB2 program invocation.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT      DNET424.ADLAB.JCL (TRADER) - 01.54          Columns 00001 00072
Command ==> SUBMIT                                     Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET4240 JOB (JIM), 'FA UTILITY',
000002 //          NOTIFY=DNET424,
000003 //          MSGCLASS=A, REGION=0M,
000004 //          CLASS=A
000005 //TRADEDB2 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT), REGION=4M
000006 //*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM IKJEFT01
000007 //CEEOPDS DD * !INVOCATION FOR REMOTE GUI
000008 TEST(, EQACMD, , TCP/IP&9.76.134.15%8001:)
000009 //EQACMD DD *
000010 SET AUTO OFF;
000011 SET WARN OFF;
000012 //*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM IKJEFT01
000013 //SYSTSIN DD *
000014 DSN SYSTEM(DSNC)
000015 RUN PLAN(TRADERIC) -
000016 PROGRAM( TRADERD )
000017 END
000018 //STEPLIB DD DSN=DNET424. TRADER. LOAD, DISP=SHR
000019 //          DD DSN=DB2.V13.SDSNLOAD, DISP=SHR

```

JCL statements created for DB2 program

Enter

Debug a non-Language Environment DB2 program by using the Remote GUI

You can create the JCL to start a debug tool session for a non-Language Environment DB2 batch program. To do the job, complete the following steps:

1. Request to debug the non-Language Environment DB2 program on the GUI.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424.ADLAB.JCL (TRADER) - 01.52          Columns 00001 00072
Command ==> EQAJCL G                                   Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET4240 JOB (JIM), 'FA UTILITY',
000002 //          NOTIFY=DNET424,
000003 //          MSGCLASS=A, REGION=0M,
000004 //          CLASS=A
000005 //TRADEDB2 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT), REGION=4M
000006 //SYSTSIN DD *
000007 DSN SYSTEM(DSNC)
000008 RUN PLAN(TRADERIC) -
000009 PROGRAM( TRADERD )
000010 END
000011 //STEPLIB DD DSN=DNET424. TRADER. LOAD, DISP=SHR
000012 //          DD DSN=DB2.V10.SDSNLOAD, DISP=SHR
000013 //DBRMLIB DD DISP=SHR, DSN=DNET424. TRADER. DBRMLIB
000014 //TRANSACTION DD DISP=SHR, DSN=DNET424. TRADER. BATCH. TRANFILE
000015 //REPOUT DD SYSOUT=*
000016 //TRANREP DD SYSOUT=*
000017 //SYSTSPRT DD SYSOUT=*
000018 //SYSPRINT DD SYSOUT=*
000019 //SYSUDUMP DD SYSOUT=*

```

Request to debug Non-LE program invoked by TSO Batch

Enter

2. Set the Language Environment Program option to NO if the DB2 program is non-Language Environment, and request the Debug Tool debug libraries, the LDD statements, and the Automonitor.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
Enter the parameters below:
LE Program ==> NO (Language Environment Enabled YES/NO)

Enter the TCP/IP Addr and port id of the workstation's remote debug daemon
IP Address ==> 9.76.134.15
Port ==> 8001

Enter '/' to enter additional Debug Tool information
DT Debug Libs ==> / (Identify debug libraries - LANGX, SYSDEBUG, other)
LDD Programs ==> / (Load Assembler or LANGX COBOL debug data)
At Entry ==> / (Request to stop in subprograms)
AutoMonitor on ==> _ (Automatically monitor variables)
Warning off ==> _ (Allows variable changes for Optimized programs)
Code Coverage ==> _ (Code Coverage)
Show Comments ==> _ (Instructions on how to display subprograms)

Press Enter to continue

```

Enter

```

000018 //STEPLIB DD DSN=DNET424.TRADER.LOAD,DISP=SHR
000019 // DD DSN=DB2.V13.SDSNLOAD,DISP=SHR

```

3. Enter the name of the non-Language Environment program, **TRADERD**.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
E ----- Debug Tool Program Name -----
E Non-LE program changes require a load module name
E Enter Loadmod ==> TRADERD (PGM=xxxxxxx) in your JCL
E
E Press Enter to continue
E

AutoMonitor on ==> (Automatically monitor variables)
Warning off ==> (Allows variable changes for Optimized programs)
Show Comments ==> (Instructions on how to display subprograms)

Press Enter to continue

```

Enter

```

000017 //SYSTSPRT DD SYSOUT=*
000018 //SYSPRINT DD SYSOUT=*
000019 //SYSUDUMP DD SYSOUT=*

```

4. Enter one or more debug libraries that are associated with the **TRADERD** program and subprograms that you want to debug.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
E |----- Debug Tool Program Name -----|
|----- Debug Tool Debug Libraries -----|
Enter Debug Tool Debug Libraries (LANGX, SYSDEBUG and other types)
Enter fully qualified names without quotes
Library1 ==> DNET424.ADLAB.EQALANGX
Library2 ==>
Library3 ==>
Library4 ==>
Library5 ==>
Library6 ==>
Press Enter to continue
F1=HELP F2=SPLIT F3=END F4=RETURN F5=RPFND F6=RCHG
F7=UP F8=DOWN F9=SWAP F10=LEFT F11=RIGHT
F1=Help F2=Split F3=Exit F4=Return F5=Rfind F12=Retrieve
F7=Up F8=Down F9=Swap F10=Left F11=Right
Enter

```

5. The program name is required for the LDD statements.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool GUI (Remote Debugger) Parameters -----
E |----- Debug Tool Program Name -----|
|----- Debug Tool Debug Libraries -----|
|----- Debug Tool Non-LE Programs -----|
Enter name of the program and subprogram(s) you plan to debug:
Programs ==> TRADERD
==>
==>
LDD instructs Debug Tool to load the program source. Enter
the names of the programs you plan to debug. The program name
may be different from the load module (PDS or PDSE member) name.
Press Enter to continue
F1=HELP F2=SPLIT F3=END F4=RETURN F5=RPFND F6=
F7=UP F8=DOWN F9=SWAP F10=LEFT F11=RIGHT
Enter

```

6. To invoke non-Language Environment DB2 program, change the program name in the SYSTSIN statements from **TRADERD** to **EQANMDBG**, and enter the DD name **EQANMDBG** 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 previous example to create the appropriate statements.


```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool Wizard - Assembler DB2 Invocation -----
Debug Tool Wizard does not support DB2 non-LE JCL changes. To debug a
DB2 non-LE program, manually make the changes shown below in White.

//STEP50 EXEC PGM=IKJEFT01
....
//SYSTSIN DD *
DSN SYSTEM(DSNA)
RUN PROGRAM( EQANMDBG ) PLAN(MYPROG) -
  PARM('ABC,123')
END
/*
//EQANMDBG DD *
myprog,TEST(,,VTAM userid:)
//INSPREF DD *
  LDD myprog

Press PF3 to exit
F1=HELP    F2=SPLIT    F3=END      F4=RETURN   F5=RFIND   F6=
F7=UP      F8=DOWN     F9=SWAP    F10=LEFT   F11=RIGHT

```

Start Code Coverage without an interactive Debug Tool session

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, Enterprise PL/I and z/OS XL C.

You can invoke Code Coverage with or without an interactive debug session.

Code Coverage is enabled in one of two ways in EQAOPTS:

- A customized Debug Tool EQAOPTS module to identify the 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 will generate the following statements:

```

//EQAOPTS DD *
EQAXOPT CCPRGSELECTDSN,'&&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

```

For information about what compilers are supported and which compiler options are required for Code Coverage, see Appendix E, “Debug Tool Code Coverage,” on page 491.

You can start Code Coverage without invoking an interactive Debug Tool session. To do the job, complete the following steps:

1. When you use the `EQAJCL C` option, Code Coverage data is generated without invoking a debugging session.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XSAM) - 01.99          Columns 00001 00072
Command ==> EQAJCL C                               Scroll ==> CSR
***** Top of Data *****
000001 //DNET424T JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //          SET &PROGRAM=IJCAMS
000004 //PRINT1 EXEC PGM=IJCAMS
000005 //SYSPRINT DD SYSOUT=*
000006 //FILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA), DISP=SHR
000007 //SYSIN DD *
000008 PRINT INFILE(FILE) COUNT(1)
000009 /*
000010 //RUNSAM1 EXEC PGM=IJCAMS, 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 COMMENT)
000019 *-----

```

Request Code Coverage without Debug Session

Enter

2. Select the program step name that you want to gather Code Coverage for.

```

----- Program Selection List ----- Row 1 to 3 of 3
Command ==> _____ Scroll ==> CSR
Enter "S" to select a step to Debug
Press "F3" to continue or enter CANCEL to cancel the request.

Sel Num  Program  StepName  Linenum
-----  -
1        IDCAMS   PRINT1    4
2        &PROGRAM RUNSAM1   9
3        IDCAMS   PRINT2   30
-----
***** Bottom of data *****

```

Enter

3. To invoke Debug tool and do Code Coverage, a Language Environment variable EQA_STARTUP_KEY=CC is added to the 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 Debug Tool Utilities option E Debug Tool Code Coverage.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XSAM) - 01.26 Columns 00001 00072
Command ==> SUBMIT Scroll ==> CSR
000006 //FILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
000007 //SYSIN DD *
000008 Submit the job to create code coverage data in the CCOUTPUT dsn
000009 //RUNSAM1 EXEC PGM=&PROGRAM,
000010 // REGION=4M
000011 /*ILINES BELOW CREATED BY THE EQAJCL Command
000012 //CEEOPTS DD * !INVOCATION FOR
000013 TEST (ALL,*,PROMPT,MFI:*,ENVAR("EQA_STARTUP_KEY=CC")
000014 //EQAOPTS DD *
000015 EQAXOPT CCPRGSELECTDSN,'&&USERID.DBGTOOL.CCPRGSEL'
000016 EQAXOPT CCOUTPUTDSN,'&&USERID.DBGTOOL.CCOUTPUT'
000017 EQAXOPT CCOUTPUTDSNALLOC,'MGHTCLAS(STANDARD)
000018 STORCLAS(DEFAULT) LRECL(255) BLKSIZE(0) RECFM(V,B)
000019 DSORG(PS) SPACE(2,2) CYL'
000020 EQAXOPT END
000021 /*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM &PROG
000022 //STEPLIB DD DISP=SHR,DSN=&SYSUID..ADLAB.LOAD
000023 //SYSOUT DD SYSOUT=*
000025 //SYSOUT DD SYSOUT=*

```

Code Coverage invocation requested via ENVAR EQA_STARTUP_KEY

Code Coverage reporting can be generated using Debug Tool Utilities, option E

Enter

Start Code Coverage with an interactive Debug Tool session using the Terminal Interface Manager

You can also create code coverage information during an interactive debugging session. To do the job, complete the following tasks:

1. Enter EQAJCL T to navigate to the Debug Tool Terminal Interface Manager menu.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XSAM) - 01.99 Columns 00001 00072
Command ==> EQAJCL T Scroll ==> CSR
***** Top of Data *****
000001 //DNET424T JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 // MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 // SET &PROGRAM=SAM1
000004 //PRINT1 EXEC PGM=IDCAMS
000005 //SYSPRINT DD SYSOUT=*
000006 //FILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR
000007 //SYSIN DD *
000008 PRINT INFILE(FILE) COUNT(1)
000009 /*
000010 //RUNSAM1 EXEC PGM=SAM1, 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 COMMENT)
000019 *-----

```

Enter

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 (/) for Code Coverage.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager) Parms -----
Enter optional parameters below
LE Program ==> YES (Language Environment Enabled)
Userid ==> DNET424 (UserId)

Enter '/' to enter additional Debug Tool information
DT Debug Libs ==> _ (Identify debug libraries - LANGX, SYSDEBUG, other)
LDD Programs ==> _ (Load Assembler or LANGX COBOL debug data)
At Entry ==> / (Request to stop in subprograms)
AutoMonitor on ==> _ (Automatically monitor variables)
Warning on ==> _ (Prohibits variable changes for Optimized COBOL)
Code Coverage ==> / (Code Coverage)
Debug Tool Log ==> _ (Capture Debug Tool Log)
Show Comments ==> _ (Instructions for Debug Tool programs)

Press Enter to continue

```

Program must be LE enabled for Code Coverage

Request Code Coverage during Debug Session

Enter

000017 //CUSTRPT DD SYSOUT=*

3. Select the program and step name that you want to debug and gather Code Coverage for.

```

----- Program Selection List ----- Row 1 to 3 of 3
Command ==> _____ Scroll ==> CSR
Enter "S" to select a step to Debug
Press "F3" to continue or enter CANCEL to cancel the request.

Sel Num  Program  StepName  Linenum
-----  -
1  IDCAMS  PRINT1  4
2  &PROGRAM  RUNSAM1  9
3  IDCAMS  PRINT2  30
=====
***** Bottom of data *****

```

S

Enter

4. The JCL shown will start a debug session on the Terminal Interface Manager, and collect code coverage information. After the debugging session completes, you can view this information by using the Debug Tool Utilities menu, option E Debug Tool Code Coverage.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XSAM) - 01.26 Columns 00001 00072
Command ==> SUBMIT Scroll ==> CSR
000009 //RUNSAM EXEC PGM=&PROGRAM,
000010 // REGION=4M
000011 /*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM &PROGRAM
000012 //CEEOPTS DD * INVOCATION FOR TERMINAL INTERFACE MANAGER
000013 TEST(,EQACMD,,VTAM%DNET424:),
000014 ENVAR("EQA_STARTUP_KEY=DCC")
000015 //EQACMD DD *
000016 SET LOG OFF;
000017 SET AUTO OFF;
000018 //EQA0PTS DD *
000019 EQAXOPT CCPRGSELECTDSN, '&&USERID.DBGTOOL.CCPRGSEL'
000020 EQAXOPT CCOUTPUTDSN, '&&USERID.DBGTOOL.CCOUTPUT'
000021 EQAXOPT CCOUTPUTSNALOC, 'MGMTCLAS(STANDARD)
000022 STORCLAS(DEFAULT) LRECL(255) BLKSIZE(0) RECFM(V,B)
000023 DSORG(PS) SPACE(2,2) CYL'
000024 EQAXOPT END
000025 SET WARN OFF;
000026 /*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM &PROG
000027 //STEPLIB DD DISP=SHR,DSN=&SYSUID..ADLAB.LOAD
000028 //CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA),DISP=SHR

```

Debug Tool Session initiated, and Code Coverage collected

Enter

Debug a Language Environment VS COBOL II program compiled with the NOTEST option by using the Terminal Interface Manager

If you do not debug VS COBOL II programs, skip this use case.

You can use one of the following ways to debug a VS COBOL II program:

- Use the TEST compilation option.
- Use the NOTEST compilation option, which is described in this use case. This method is called LANGX COBOL in the Debug Tool manuals.


To debug Language Environment with a VS COBOL II Program compiled with NOTEST option by using the Terminal Interface Manager, complete the following steps:

1. To debug VS COBOL II programs by using the Terminal Interface Manager, enter EQAJCL T.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XSAMII1) - 01.49          Columns 00001 00072
Command ==> EQAJCL T                               Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET424B JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //*****
000004 /**  RUN COBOL II SAMPLE PROGRAM SAMII1
000005 //*****
000006 //RUNSAM1 EXEC PGM=SAMII1,
000007 //          REGION=4M
000008 //STEPLIB DD DSN=&SYSUID..ADLAB.LOAD, DISP=SHR
000009 //CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2), DISP=SHR
000010 /***  CUSTFILE DD DSN=&SYSUID..ADLAB.FILES(CUST2FA), DISP=SHR
000011 //SYSPRINT DD SYSOUT=*
000012 //SYSOUT DD SYSOUT=*
000013 //CISTRPT DD SYSOUT=*
000014 //TRANFILE DD *
000015 *TRAN
000016 *-----
000017 PRINT      <== PRINT CUSTOMER LIST
000018 XXXXXX     <== BAD TRANSACTION REQUEST
000019 TOTALS     <== PRINT TOTALS

```




- VS COBOL II programs might be linked either as Language Environment (LE) programs or non-Language Environment programs. In this use case, the program is linked as Language Environment enabled. Although this is a Language Environment program, you must still identify the Debug Tool debug libraries, and issue **LDD statements** for the modules that you want to debug. In addition, you can set breakpoints for subprograms, and set the **Automonitor ON** if you want.

```

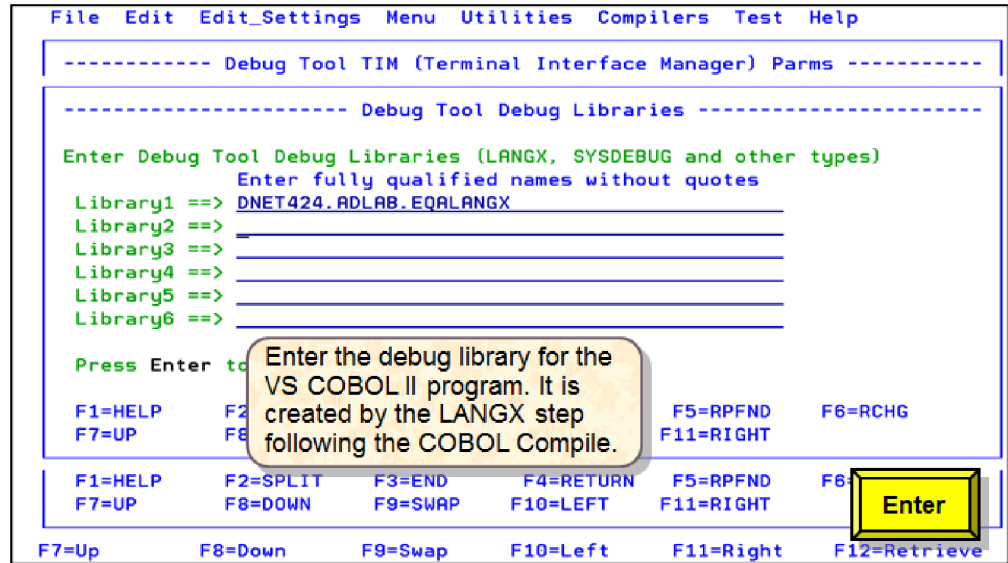
File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager)Parms -----
Enter optional parameters below
LE Program   ==> YES (Language Environment Enabled YES/NO)
Userid      ==> DNET424 (UserId)
Enter '/' to enter additional Debug
DT Debug Libs ==> / (Identify debug libraries)
LDD Programs  ==> / (Load Assembler modules)
At Entry      ==> / (Request to stop in subprograms)
AutoMonitor on ==> / (Automatically monitor variables)
Warning on    ==> - (Prohibits variable changes for Optimized COBOL)
Code Coverage ==> - (Code Coverage)
Debug Tool Log ==> - (Capture Debug Log on SYSOUT=*)
Show Comments ==> - (Instructions on how to display subprograms)
Press Enter to continue

```

Most VS COBOL II programs are LE. If a SCEELKED library is allocated to the SYSLIB statement in the Link-Edit step, then the program is LE enabled.



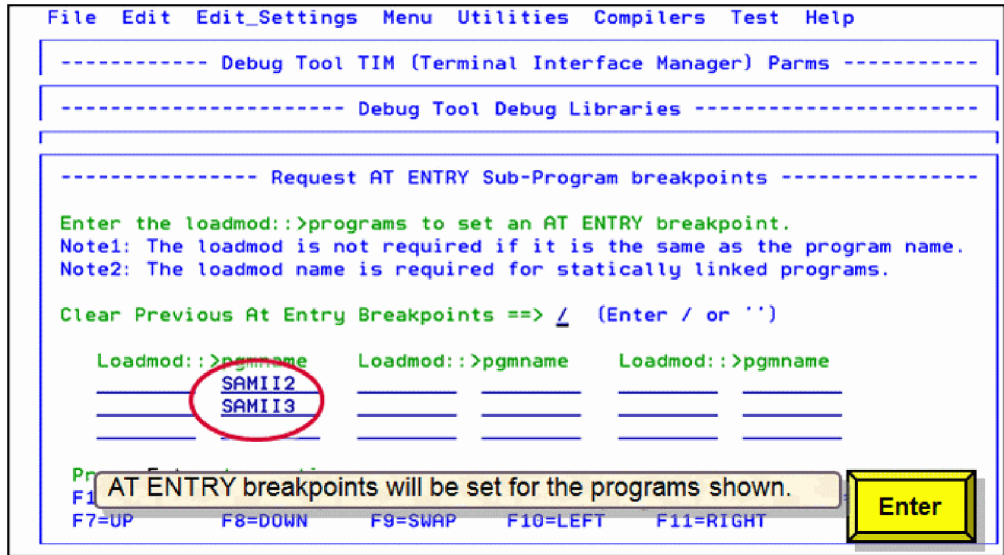
- Enter one or more names of the Debug Tool side file data sets that you want to use.



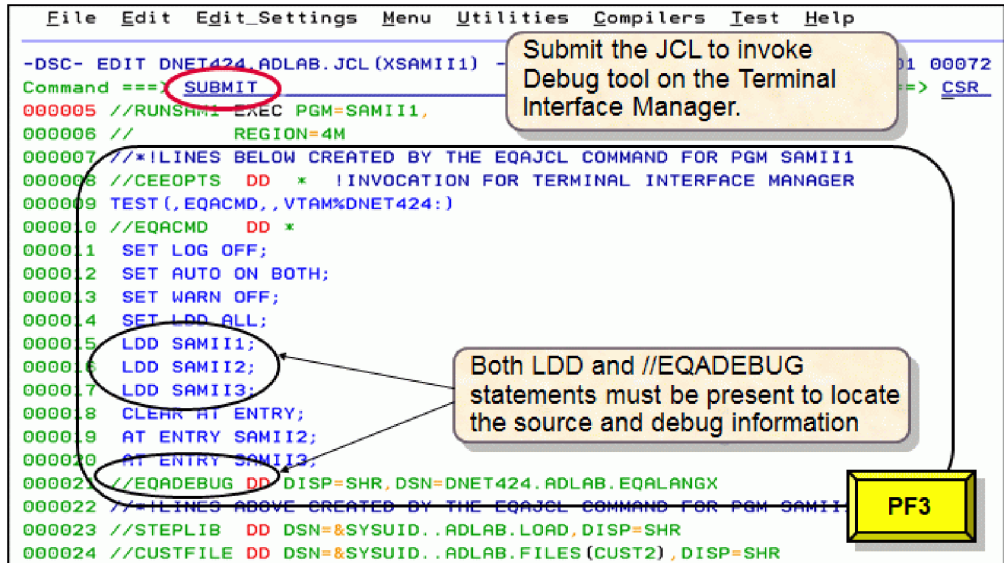
4. Enter the names of the main program and subprograms that you want to debug.



5. Enter the names of the subprograms that you want to set an entry breakpoint for.



- The JCL for VS COBOL II Debug Tool invocation is created. To define the source and debug information during the debug session, the LDD statements and EQADEBUG libraries are required.



Debug a non-Language Environment program when the debug member does not match the program name

- To invoke Debug Tool JCL Wizard by using the Terminal Interface Manager, enter EQAJCL T.


```

File Edit Edit_Settings Menu Utilities Compilers Test Help
-DSC- EDIT DNET424 ADLAB.JCL(XASAM1) - 01.99          Debug statements removed
Command ==> EQAJCL T                               Scroll ==> CSR
***** ***** Top of Data *****
000001 //DNET424X JOB (ACCTG), 'IBM TOOLS WORKSHOP', REGION=4M, CLASS=A,
000002 //          MSGCLASS=H, NOTIFY=&SYSUID, MSGLEVEL=(1,1)
000003 //ASAM1 EXEC PGM=ASAM1, PARM='ABCD',
000004 // COND=(4,LT)
000005 //STEPLIB DD DSN
000006 //INSPREF DD D
000007 //FILEIN DD *,DCB=(LRECL=80)
000008 INPUT RECORD ONE
000009 INPUT RECORD TWO
000010 INPUT RECORD THREE
000011 INPUT RECORD FOUR
000012 INPUT RECORD FIVE
000013 INPUT RECORD SIX
000014 INPUT RECORD SEVEN
000015 INPUT RECORD EIGHT
000016 ABEND <== "ABEND" WILL CAUSE THE SAMPLE PROGRAM TO ABEND
000017 //FILEOUT DD SYSOUT=*
000018 //SYSUDUMP DD SYSOUT=*
***** ***** Bottom of Data *****

```

Debug Assembler program ASAM1 using the Terminal Interface Manager

Enter

2. Enter No if the program is not a Language Environment program.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager) Params -----
Enter optional parameters below
LE Program ==> YES (Language Environment Enabled YES/NO)
Userid ==> DNET424 (UserId)
Enter '/' to enter additional Debug Tool Parameters
DT Debug Libs ==> / (Identify debug libraries - LANGX, SYSDEBUG, other)
LDD Programs ==> / (Load Assembler or LANGX COBOL debug data)
At Entry ==> / (Request to stop in subprograms)
AutoMonitor on ==> / (Automatically monitor variables)
Warning on ==> - (Prohibits variable changes for Optimized COBOL)
Code Coverage ==> - (Code Coverage)
Debug Tool Log ==> - (Capture Debug Log on SYSOUT=*)
Show Comments ==> - (Instructions on how to display subprograms)
Press Enter to continue

```

Request debug libraries and LDD Statements

Enter

```

000017 //CUSTRPT DD SYSOUT=*

```

3. Enter the name of the non-Language Environment program that you want to debug. The program name is ASAM1 in this use case.

```

File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager) Parms -----
E ----- Debug Tool Program Name -----
Non-LE program changes require a program name
Enter Program => ASAM1
E Press Enter to continue
ASAM1 is the name of the Non-LE program we want to debug

F1=HELP      F2=SPLIT     F3=END       F4=IPT View  F5=RFIND
F6=RCHANGE   F7=UP        F8=DOWN      F9=SWAP      F10=>BACK

F1=HELP      F2=SPLIT     F3=END       F4=IPT View  F5=RFIND
F6=RCHANGE   F7=UP        F8=DOWN      F9=SWAP      F10=>BACK

000014 INPUT RECORD ONE
000015 INPUT RECORD TWO
000016 INPUT RECORD THREE
000017 INPUT RECORD FOUR
F1=Help      F2=Split     F3=Exit      F4=Return    F5=Rfind
F7=Up        F8=Down      F9=Swap      F10=Left     F11=Right   F12=Retrieve
Enter

```

4. Enter the name of Debug Tool debug libraries.

```

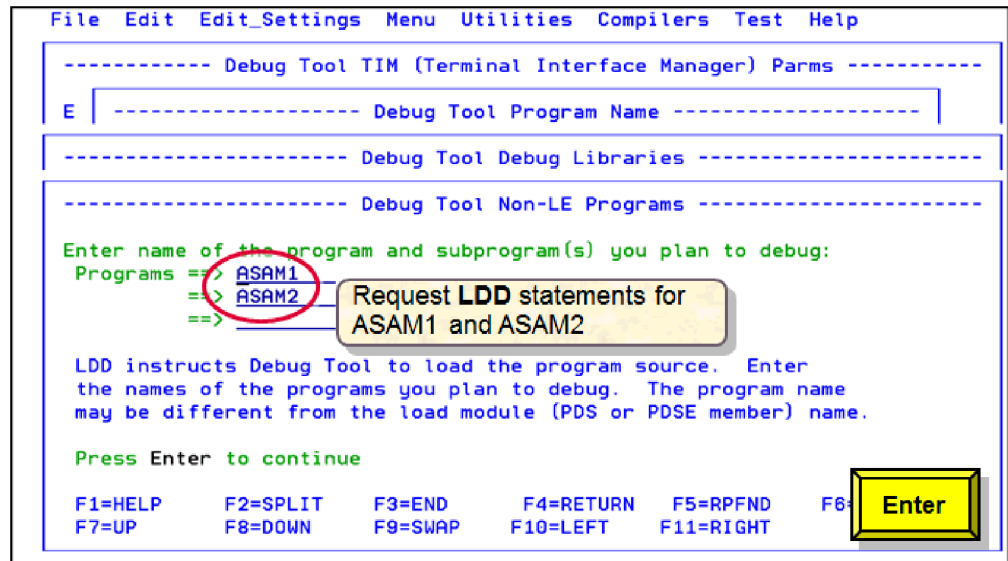
File Edit Edit_Settings Menu Utilities Compilers Test Help
----- Debug Tool TIM (Terminal Interface Manager) Parms -----
E ----- Debug Tool Program Name -----
----- Debug Tool Debug Libraries -----
Enter Debug Tool Debug Libraries (LANGX, SYSDEBUG and other types)
Enter fully qualified names without quotes
Library1 ==> DNET424.ADLAB.EQALANGX
Library2 ==>
Library3 ==>
Library4 ==>
Library5 ==>
Library6 ==>
The debug library is identified.
Press Enter to continue

F1=HELP      F2=SPLIT     F3=END       F4=RETURN    F5=RPFND    F6=RCHG
F7=UP        F8=DOWN      F9=SWAP      F10=LEFT     F11=RIGHT

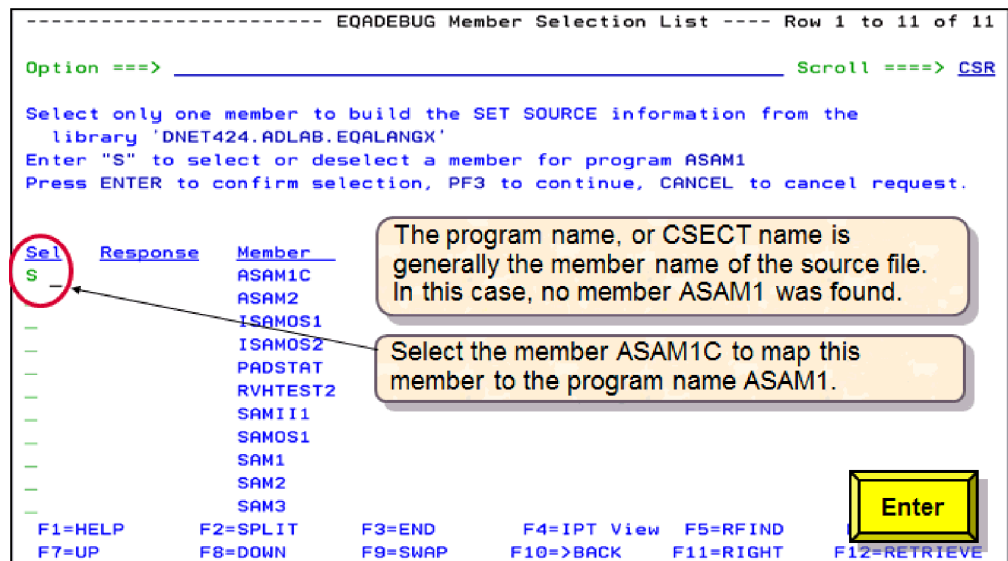
F7=Up        F8=Down      F9=Swap      F10=Left     F11=Right   F12=Retrieve
Enter

```

5. Enter the names of the programs that you want to request LDD statements for. The names are ASAM1 and ASAM2 in this use case.



6. Enter **S** to select or deselect a debug member for program ASAM1. If the program name or CSECT name is the member name of the debug file, no member ASAM1 was found.



7. The **Set Source** statement is built to map the program name with the debug member name.

```

----- EQADEBUG Member Selection List ----- Row 1 to 11 of 11
Option ==> _____ Scroll ==> CSR
Select only one member to build the SET SOURCE information from the
library 'DNET424.ADLAB.EQALANGX'
Enter "S" to select or deselect a member for program ASAM1
Press ENTER to confirm selection, PF3 to continue, CANCEL to cancel request.
SET SOURCE ON("ASAM1") DNET424.ADLAB.EQALANGX(ASAM1C)
Sel  Response  Member
*   *Selected  ASAM1C
-
-   ASAM2
-   ISAMOS1
-   ISAMOS2
-   PADSTAT
-   RVHTEST2
-   SAMI11
-   SAMOS1
-   SAM1
-   SAM2
-   SAM3
F1=HELP   F2=SPLIT   F3=END     F4=IPT View  F5=RFIND   F6=
F7=UP     F8=DOWN    F9=SWAP    F10=>BACK   F11=RIGHT  F12=

```

The Set Source statement is built to map the program name with the source member.

PF3

8. Enter S to select the step that runs program ASAM1.

```

----- Program Selection List ----- Row 1 to 2 of 2
Command ==> _____ Scroll ==> CSR
Enter "S" to select a step to Debug ASAM1
Press "F3" to continue or enter CANCEL to cancel the request.
Sel Num  Program  StepName  Linenum
S  2  ASAM1  ASAM1  9
***** Bottom of data *****

```

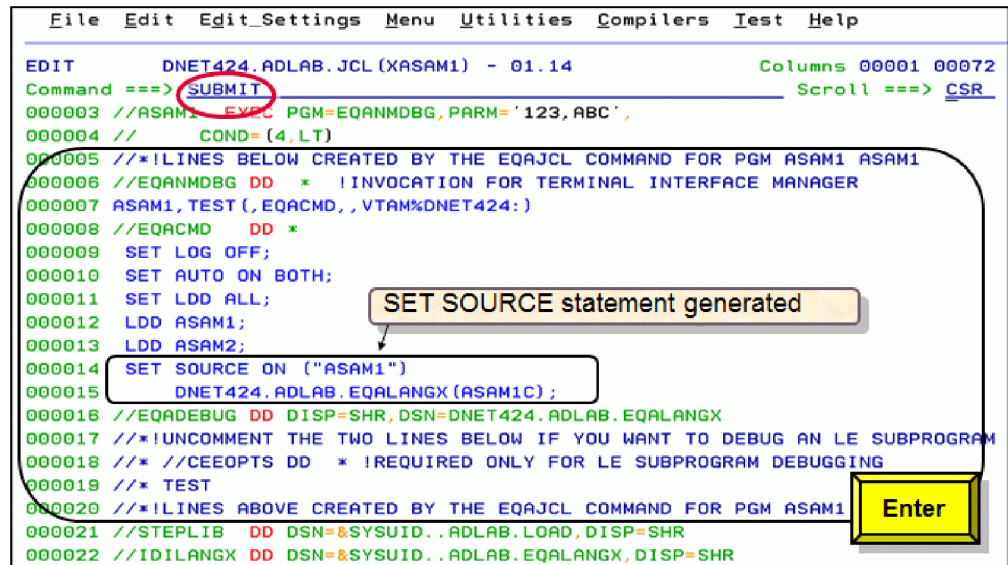
Select the step name in to debug

Enter

F1=HELP F2=SPLIT F3=END F4=IPT View F5=RFIND F6=
F7=UP F8=DOWN F9=SWAP F10=>BACK F11=RIGHT F12=RETRIEVE

9. The SET SOURCE statement is generated.

```
File Edit Edit_Settings Menu Utilities Compilers Test Help
EDIT          DNET424.ADLAB.JCL(XASAM1) - 01.14          Columns 00001 00072
Command ==> SUBMIT                               Scroll ==> CSR
000003 //ASAM1  EXPC PGM=EQANMDBG,PARM='123,ABC',
000004 //      COND=(4,LT)
000005 /*ILINES BELOW CREATED BY THE EQAJCL COMMAND FOR PGM ASAM1 ASAM1
000006 //EQANMDBG DD * !INVOCATION FOR TERMINAL INTERFACE MANAGER
000007 ASAM1,TEST(,EQACMD,,VTAM%DNET424:)
000008 //EQACMD  DD *
000009 SET LOG OFF;
000010 SET AUTO ON BOTH;
000011 SET LDD ALL;
000012 LDD ASAM1;
000013 LDD ASAM2;
000014 SET SOURCE ON ("ASAM1")
000015 DNET424.ADLAB.EQALANGX(ASAM1C);
000016 //EQADEBUG DD DISP=SHR,DSN=DNET424.ADLAB.EQALANGX
000017 /*!UNCOMMENT THE TWO LINES BELOW IF YOU WANT TO DEBUG AN LE SUBPROGRAM
000018 /* //CEEOPST DD * !REQUIRED ONLY FOR LE SUBPROGRAM DEBUGGING
000019 /* TEST
000020 /*ILINES ABOVE CREATED BY THE EQAJCL COMMAND FOR PGM ASAM1
000021 //STEPLIB DD DSN=&SYSUID..ADLAB.LOAD,DISP=SHR
000022 //IDILANGX DD DSN=&SYSUID..ADLAB.EQALANGX,DISP=SHR
```



The end.

Appendix E. Debug Tool Code Coverage

You can use IBM Debug Tool 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 Debug Tool.
2. Code coverage observations gathering: Using Debug Tool 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 Debug Tool Code Coverage

You can use Debug Tool to measure code coverage in your application testing. In this part, you can learn the basics of running the code coverage function of Debug Tool 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 Debug Tool Code Coverage

Debug Tool 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 Debug Tool 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, Debug Tool 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 Debug Tool, creating code coverage reports, and displaying the reports.

- Startup
- EQAOPTS
- Debug Tool Utilities Option E. Debug Tool Code Coverage
 - Observation Viewer. Option E.1: Browse code coverage observations.
 - Debug Tool Options. Option E.2: Create or modify Debug Tool 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 Debug Tool

The following figure shows the steps that are required for Debug Tool 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 Debug Tool Code Coverage session.
- An Options file that indicates what programs you want Debug Tool to monitor to get code coverage observations.
- EQAOPTS commands that indicate the location of the input and output data sets.

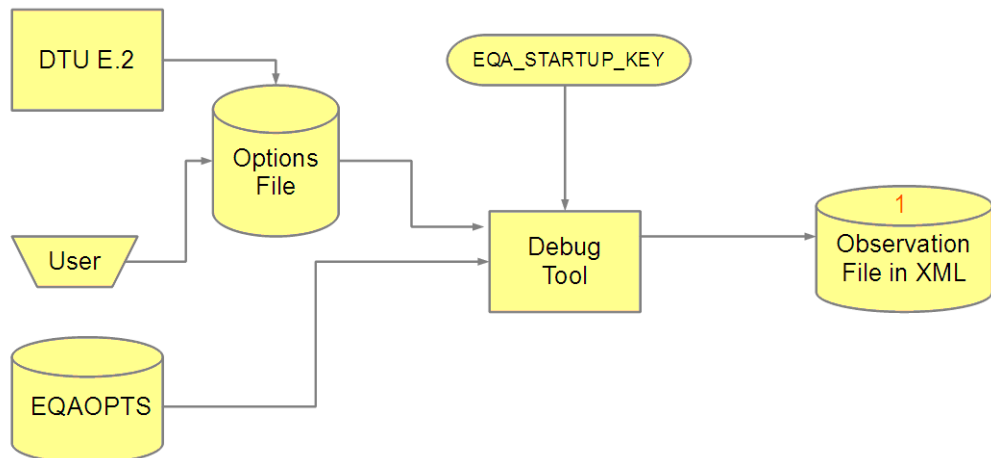


Figure 3. Step 1 Gathering code coverage observations with Debug Tool

The code coverage observations collection process is as follows:

1. When the environment variable `EQA_STARTUP_KEY` is specified during invocation of the debugger, Debug Tool collects code coverage observations.
2. Debug Tool gathers code coverage data based on input from the Options file.
3. The Options file can be created by using Debug Tool Utilities Option E.2. Alternatively, you can code an Options file by following the Options file XML DTD syntax.
4. Debug Tool retrieves the name of data set of the Options file from a value that is provided by an EQAOPTS command.
5. Debug Tool retrieves the name of data set of the Observation file 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 Debug Tool 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 Debug Tool Utilities Option E.3. Alternatively, you can code a Selection file by following the Selection file XML DTD syntax.

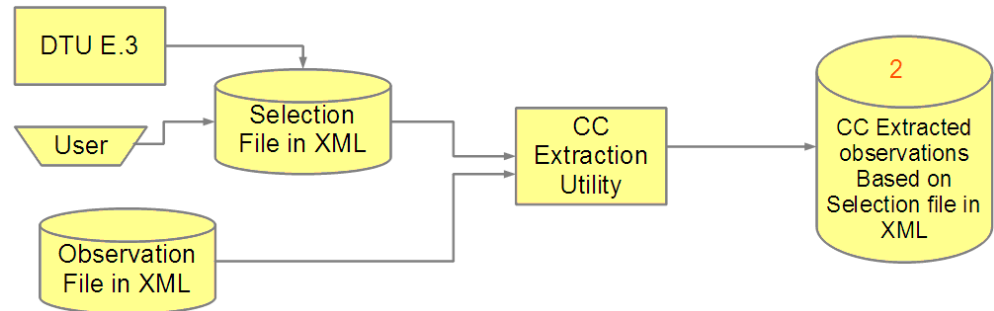


Figure 4. Step 2 Code coverage selection and extraction process

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 Debug Tool 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 EQAXCC2 REXX exec. You must specify the following DDNAMES:

EQAC SINP

Location of Observation file.

EQAC SSEL

Location of Selection file.

EQAC SOUT

Location of output code coverage extracted observations file.

An example of using EQAXCC2 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 Debug Tool 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.

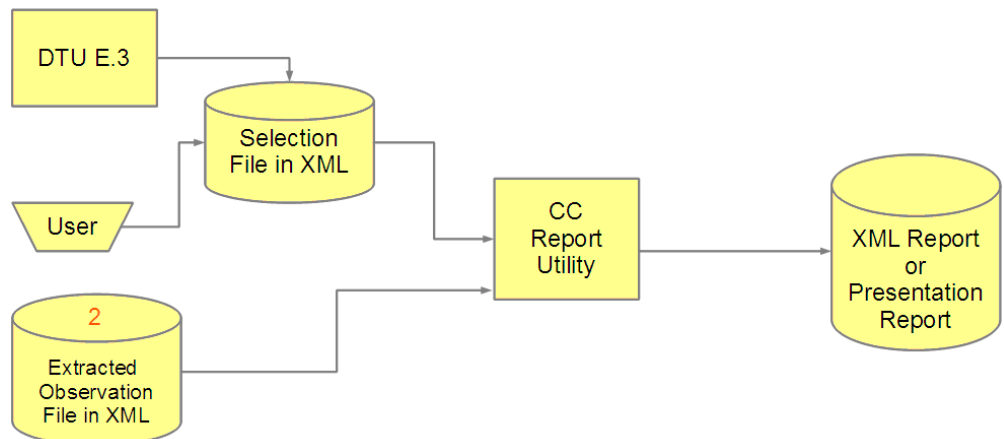


Figure 5. Step 3 Code coverage report process

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 Debug Tool Code Coverage session. It takes as input either the code coverage Observation files first created by Debug Tool 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.

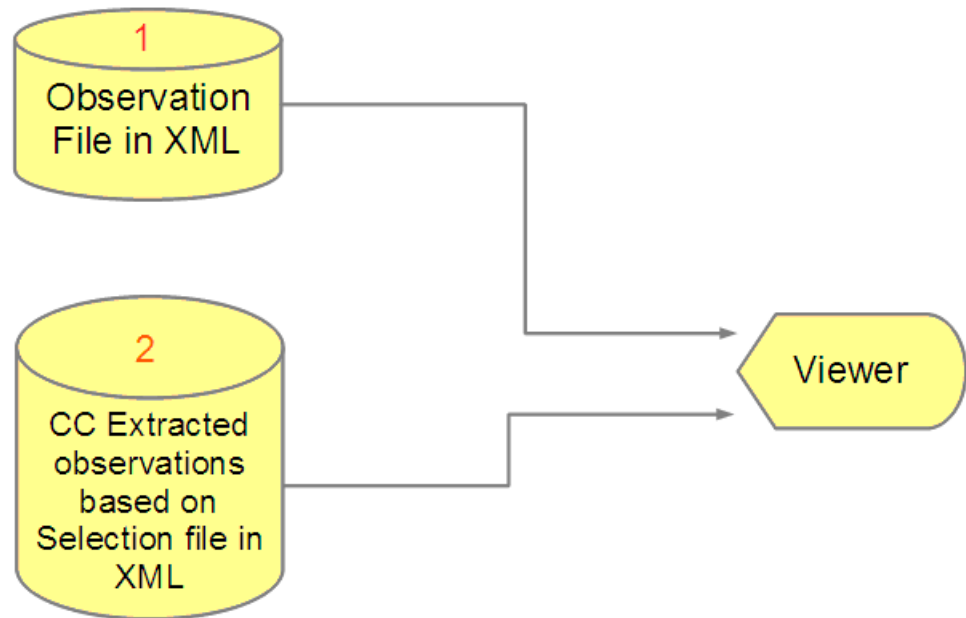


Figure 6. Step 4 The Viewer

- The Viewer is part of Debug Tool 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 Debug Tool (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 Debug Tool

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 Debug Tool. 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 and 5. 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 - TEST.

Code Coverage is supported for Enterprise PL/I for z/OS Version 4.2 and above. 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 Debug Tool 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 Debug Tool, by saving it in an 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 a HFS file.

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 Debug Tool 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 Debug Tool. 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'
```

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 Debug Tool Code Coverage session is requested. In this case, an interactive debug session is not launched.
- DCC** A combined Debug Tool 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.

Debug Tool 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: Debug Tool code coverage session and RDz code coverage session. Debug Tool handles the Debug Tool code coverage session. RDz handles the RDz code coverage session. The code coverage data format and presentation are different in the two sessions.

EQA_STARTUP_KEY	Debug Tool session device	Debug Tool interactive debug session	Debug Tool code coverage session	RDz code coverage session
CC	MFI	No	Yes	No
CC	TCPIP - PDTools Studio	Yes	No	No

EQA_STARTUP_KEY	Debug Tool session device	Debug Tool interactive debug session	Debug Tool code coverage session	RDz code coverage session
CC	TCPIP - RDz	No	No	Yes
DCC	MFI	Yes	Yes	No
DCC	TCPIP - PDTools Studio	Yes	Yes	No
DCC	TCPIP - RDz	Yes	Yes	No

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 Debug Tool 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 Debug Tool TCPIP with the PDTools Studio or RDz, 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 Debug Tool 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 Debug Tool collects code coverage observations for programs C0B01A, C0B01B, C0B01C, and C0B01D. Debug Tool 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>COB01C</EXTNAME>  
<EXTNAME>COB01D</EXTNAME>
```

Generating code coverage extracted observations

Depending on the values that are provided in the Options file, Debug Tool 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, Debug Tool 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 Debug Tool selects such programs by providing a Selection file. You can create the Selection file by using Debug Tool 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 an 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 Debug Tool. 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	GYOUNG	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'mmm'*)

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 21. 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 Debug Tool Utilities Option E.3, or you can code the file manually by following the Selection file XML DTD syntax.

Debug Tool Utilities Option E

The Debug Tool Code Coverage option in Debug Tool 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 Debug Tool Utilities Option E suboptions.

```
----- Debug Tool Code Coverage -----
Option ==>

1 Observation viewer
  Browse code coverage observations.

2 Debug Tool options
  Create or modify the Debug Tool 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 Debug Tool Utilities. You can view the XML entries of the code coverage observations in a table format that facilitates analysis. The Viewer uses either the original file of observations that were created by Debug Tool 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.

```
----- Debug Tool - 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 Debug Tool code coverage session.  
  
Data Set Name:  
Data Set Name . . . 'GYOUNG.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.

----- Debug Tool - Code Coverage Observation View 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:  GYOUNG
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:  GYOUNG
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:  GYOUNG
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:  GYOUNG
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:  GYOUNG
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:  GYOUNG
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 510.

Option E.2 Code Coverage Options file

In this option, you can create the Options file that is used as an input to Debug Tool 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 520).

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.

```
----- Debug Tool - 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 . . . 'GYOUNG.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 Debug Tool 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.

```

----- Debug Tool - 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 520). 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.

```

----- Debug Tool - 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 . . . 'GYOUNG.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.

```

----- Debug Tool - 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      GYOUNG      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 520).

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	ELIN	LMD1	PRG1	2013/04/08	10:10:20	Y	100	80	80%
2	Pay1	Test2	ELIN	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 Debug Tool 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


```

----- Debug Tool - 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 . . . 'GYOUNG.DBGTOOL.CCOUTPUT'

Specify the name of a code coverage selection criteria data set.

Data Set Name . . . 'GYOUNG.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage observation output data set.

Data Set Name . . . 'GYOUNG.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.

```

----- Debug Tool - 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 . . . 'GYOUNG.DBGTOOL.CCOUTPUT'

Specify the name of a code coverage selection criteria data set.

Data Set Name . . . 'GYOUNG.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage observation output data set.

Data Set Name . . . 'GYOUNG.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:

```

----- Debug Tool - 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 . . . 'GYOUNG.DBGTOOL.CCOUTPUT.SELECTED'

Specify the name of a code coverage selection criteria data set.

Data Set Name . . . 'GYOUNG.DBGTOOL.CCOBSSEL'

Specify the name of a code coverage report data set.

Data Set Name . . . 'GYOUNG.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.

```
lRpt 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 Stmt: 17              Exec Stmt: 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 for z/OS *
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         ENVIRONMENT DIVISION.
19
20         DATA DIVISION.
21
22         WORKING-STORAGE SECTION.
23         01 TAPARM1 PIC 99 VALUE 5.
24         01 TAPARM2 PIC 99 VALUE 2.
25         01 COB01B PIC X(6) VALUE 'COB01B'.
26         01 P1PARM1 PIC 99 VALUE 0.
27
28         01 TASTRUCT.
29             05 LOC-ID.
30                 10 STATE PIC X(2).
31                 10 CITY PIC X(3).
32                 05 OP-SYS PIC X(3).
33
34         PROCEDURE DIVISION.
35
36         * THE FOLLOWING ALWAYS PERFORMED
37
38
```

```

39          * Defect456Begin
40
41          PROG.
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          .
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          .
72          LOOP1.
73 >          IF TAPARM1 > 0 THEN
74 >          SUBTRACT 1 FROM TAPARM1.
75 >          CALL 'COB01B'
76          .
77          LOOP2.
78 I>          IF TAPARM2 > 0 THEN                PMR12345
79 I>          SUBTRACT 1 FROM TAPARM2.            PMR12345
80
81          * Defect456End

```

Marker type	Selection	Start Column	End 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 for z/OS */
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;

```

Marker type	Selection	Start Column	End 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/Exc1	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: GYOUNG
Load Name: C01              Prog Name: C01A
Comp Date: 2013/05/07      Comp Time: 15:53:00    Debug override: N
Tot Stmt: 12                Exec Stmt: 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     {
14 I>     int EXPARM1 = 5;
15 I>     int EXPARM2 = 2;
16     extern void C01B(void);
17 I<     void PROCA(int);

```

```

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         }

```

Marker type	Selection	Start Column	End 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 22. The column layout for the source lines

Columns	Contents
1	Blank.
2 through 7	6 digit listing line number, right justified, leading zeros suppressed.
9	<p>If executable and not using E.1 (V)iew then:</p> <ul style="list-style-type: none"> • 'I' included • 'E' excluded • 'B' both included and excluded • '' neither included nor excluded <p>If not executable or using E.1 (V)iew then:</p> <ul style="list-style-type: none"> • ''
10 through 15	<p>1 column per statement on this line. For example, col 10 represents the 1st statement, column 11 represents the 2nd statement.</p> <p>These columns indicate whether the statement was executed (>), unexecuted (<), unspecified () or specified multiple times (M, which likely indicates an internal error).</p> <p>Column 15 may contain a plus sign (+) if an executed or unexecuted tag value indicates a statement number that exceeds 6 for this line.</p>
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.

EQACSSSEL

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)*.

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 Debug Tool 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                Debug Tool CICS Control - Menu 2                S07CICPB

Select Debug Tool options

Test Option    ==>  TEST                Test/Notest
Test Level    ==>  ERROR                All/Error/None
Commands File ==>  GYOUNG.CC.CICS.GOCMD
Prompt Level  ==>  PROMPT
Preference File ==>  *

EQAOPTS File   ==>  GYOUNG.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 Debug Tool while collecting observations, remove the Commands file from the Options panel shown above, and change the value of EQA_STARTUP_KEY to DCC.

XML tags for code coverage

This section contains a set of XML tags for code coverage. The set of tags is an enhanced version of the exported XML data definition in the Appendix F of Debug Tool Coverage Utility User's Guide and Messages.

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 23. 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>	Debug Tool 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>
<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 or 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>

Table 23. XML tags and the contents (continued)

XML tag	Description	Occurrence
<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>
<INCPCNT>xxx</INCPCNT>	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>
<PROGRAMDSOMPILEDATE> xxx </PROGRAMDSOMPILEDATE>	The compile date container of data set that contains program source	1, per <COMPILATIONUNIT>
<PROGRAMDSOMPILETIME> xxx </PROGRAMDSOMPILETIME>	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: <ul style="list-style-type: none"> • 1 - COBOLSYSDEBUG • 2 - PLISYSDEBUG 	1, per <COMPILATIONUNIT>
<RUNDATE>	Date that the code coverage data was saved	1, per <DTCCODECOVERAGEFILE> or <COVERAGEFILE >
<RUNTIME>	Time that the code coverage data was saved	1, per <DTCCODECOVERAGEFILE> or <COVERAGEFILE>

Table 23. XML tags and the contents (continued)

XML tag	Description	Occurrence
<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>
<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 24. 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: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> GROUPID1 GROUPID2 USERID DBGOV

Table 25. 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

Table 25. Description of XML tags used for source maker (continued)

Tag	Description	Occurrence
<MARKERTYPE>xxx</MARKERTYPE>	Marker type. Valid types are as follows: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> GROUPID1 GROUPID2 USERID DBGOV
<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: <ul style="list-style-type: none"> 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

Debug Tool can run in batch mode, creating a noninteractive session.

In batch mode, Debug Tool 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 Debug Tool 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 Debug Tool 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 15, "Starting Debug Tool in batch mode," on page 137

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

You can use predefined IMS message region templates to debug a specific transaction in a private message region by using Debug Tool 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 Debug Tool Utilities. For detailed information, see “Starting Debug Tool Utilities” on page 9.
2. In the Debug Tool 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 Debug Tool Setup Utility data set
 - Templates previously customized and stored in a private Debug Tool Setup Utility data set

If you use a member from a private Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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

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 *Debug Tool 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 532 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 a negative offset of D0 bytes from address 45CB00, enter the command `DTST 45CB00 - D0`.

The result of the calculation (45CB00-D0) is the base address. In this example, the base address is 45CA30.

To display program storage at a 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 *Debug Tool 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.

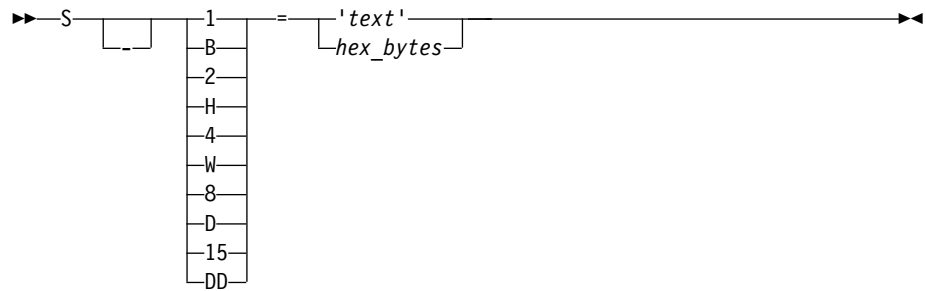
PF7

Display the previous screen.

PF8

Display the next screen.

- 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. Debug Tool Load Module Analyzer

The Debug Tool 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 Debug Tool 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 EQAZLMA 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 Debug Tool Utilities

To start the Load Module Analyzer by using Debug Tool Utilities, do the following steps:

1. Start Debug Tool 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)
- 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 NORES 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 CKVOLFPFRS:
 - If a Language Environment prologue was detected, information is included in a string identified by LEINF0=(. . . . 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.

EQASYSF

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). Debug Tool provides data for this

file in member EQALMPFX of the table library (SEQATLIB). See “Description of EQASYSFPF file format” on page 539 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). Debug Tool provides data for this file in member EQALMPGM of the table library (EQATLIB). See “Description of EQAPGMNM file format” on page 540 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:

CKVOLFRS

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 is 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 CKVOLFPERS 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.

Debug Tool 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 member in *hlq.SEQATLIB* through the SMP/E USERMOD in *hlq.SEQASAMP(EQAUMOD3)*.
- Create a data set containing the new entries. Then, concatenate this data set to the one that ships with Debug Tool.

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.

Debug Tool 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 EQALMPRM 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 Debug Tool.

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 EQASYSPPF file format" on page 539 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:

```
5655-Q10 Debug Tool Version 13 Release 1.2 Load Module Analyzer 2013/10/23 Page 15
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
@@INITE@ 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*
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
(Multiple program ID's)
```

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

DTNP is a CICS transaction, supplied by Debug Tool, 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 **Debug Tool - 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 **Debug Tool - 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*.

Appendix K. Installing the IBM Debug Tool plug-ins

The IBM Debug Tool 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 Problem Determination Tool Plug-ins (<http://www.ibm.com/software/awdtools/deployment/pdtplugins/>). 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.¹⁷
- The **DTSP Profile** view, which helps you create and manage the TEST runtime options data set (EQUAUOPTS) on your z/OS system.
- The **Instrument JCL for Debug Tool Debugging** view, which guides you through the process of filling out information that it uses to instrument JCL to start Debug Tool for batch jobs.
- The **Debug Tool Code Coverage** view, which guides you through the process that measures test coverage in application programs.
- The **Debug Tool 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 “IBM Problem Determination Tools Plug-ins V13 Combined Packages”(<http://www.ibm.com/support/docview.wss?uid=swg24033351>).

1. Go to IBM Problem Determination Tool Plug-ins (<http://www.ibm.com/software/awdtools/deployment/pdtplugins/>).
2. Click **Link to download page** that is next to “IBM Debug Tool Plug-in for Eclipse”.
3. Download the compressed file containing the DT plug-in from the IBM Debug Tool plug-in for Eclipse website.
4. Follow the instructions in “DT Plug-in Readme PDF” to install the Debug Tool plug-in.¹⁸
5. Verify that your system administrator has completed the following tasks described in the *Debug Tool Customization Guide*:
 - “Adding support for the DTCN Profiles view and APIs”
 - “Adding support for the DTSP Profile view”
6. Restart your Eclipse-based application.

Establishing a connection between the DTCN Profiles view for CICS and your z/OS system

17. You can use CICS debug configurations to manage DTCN profiles with Rational Developer for System z. To learn more about CICS debug configurations, see the topic “Debugging CICS applications” in the Developer for System z, Version 8, information center. If you choose to use CICS debug configurations, you do not need to follow the rest of the instructions in this topic.

18. As seen in the DT Plug-in Readme PDF instructions, Rational Developer for System z already contains the IBM Debug Tool plug-in. Download the DT plug-in which is contained in the compressed file only when you are not using Rational Developer for System z. However, you can still install the DTCN and DTSP plug-ins directly in Rational Developer for System z by following the instructions in the DT Plug-in Readme PDF.

Specify the settings needed to establish a connection between the **DTCN profiles** view and your z/OS system by taking the following steps:

1. Click **Window>Show view>Other**.
2. Type "DTCN" in the text box at the top of the window. Select **DTCN Local profiles, DTCN Server Profiles**, and click **OK**.
3. Click **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 TCPIP SERVICE resource" in the *Debug Tool Customization Guide*.

Port number

The port number of the z/OS system as described in "Defining the CICS TCPIP SERVICE resource" in the *Debug Tool 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 *Debug Tool 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 Debug Tool Debugging, Code Coverage, and Load Module Analyzer view" on page 555) 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. Click **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. Click **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 **Problem Determination Tools for z/OS** and click **Add** to create a connection to the Problem Determination Tools for z/OS Common Component 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 *Debug Tool 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 *Debug Tool Customization Guide*.

Default encoding

The default encoding is "cp037". If you use a different encoding scheme, specify it in this field.

7. Click **Window>Preferences**.
8. Click **Debug Tool>DTSP (non-CICS)** in the navigation pane.
9. In the Preferences window, select the Problem Determination Tools connection you created from the **Connection** list and click **Connect**.
10. If this is the first time you are connecting to the IBM Problem Determination Tools Common Component Server, click **Yes** in the Certificate Information window.
11. 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 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 Debug Tool Debugging, Code Coverage, and Load Module Analyzer view” on page 555) for diagnostic information that can help identify a mistake.

12. 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**.

Instrument JCL for Debug Tool Debugging Plug-in

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 Debug Tool for batch jobs.

You can access the Instrument JCL for Debug Tool Debugging Plug-in by taking the following steps:

1. Click **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 Debug Tool 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 Debug Tool 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 Debug Tool load module library and Language Environment CEEBINIT load module library.

Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool. <<<  
//*>>> The JCL lines above were inserted by Debug Tool. <<<
```

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 Debug Tool Language Environment user exit EQAD3CXT, see the *Specifying the TEST runtime options through the Language Environment user exit* chapter in the *Debug Tool User's Guide* or the *Debug Tool 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 RD/z, PD Tools Studio or Z/OS Explorer. 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" on page 139 for more information on using Terminal Interface Manager.

Debug Tool Code Coverage Plug-in

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:

- Click **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 debug tool 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 **Problem Determination Tools for z/OS** and click **Add** to create a connection to the Problem Determination Tools for z/OS Common Component 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 *Common Component Customization Guide and User's 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 *Common Component Customization Guide and User's 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 IBM Problem Determination Tools Common Component 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 in the statistics calculation for code coverage. It is based on the indicators in the source listing like a comment, and 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.

Debug Tool Load Module Analyzer Plug-in

Debug Tool 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 Debug Tool Load Module Analyzer Plug-in by taking the following steps:

- Click **Window > Show view > Other**.
- 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 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. 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 Debug Tool Debugging, Code Coverage, and Load Module Analyzer view

When you do actions in the **DTCN Profiles**, **DTSP Profile**, **Instrument JCL for Debug Tool 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 Debug Tool 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

The following example shows what the file might contain after you send a request to create a profile for Debug Tool for z/OS, Version 10:

```
15 Oct 2009 16:06:11 PDT
Request URI: http://tlba07me.torolab.ibm.com:33000/dtcn/smith02?clientversion=0102
Request method: PUT
<?xml version="1.0"?>
<profile>
<terminalid></terminalid>
<transactionid></transactionid>
<program>
<loadname></loadname>
```

```

<pgmname></pgmname>
</program>
<userid>smith02</userid>
<netname></netname>
<clientip></clientip>
<commareaoffset>0</commareaoffset>
<commareadata></commareadata>
<containername></containername>
<containeroffset>0</containeroffset>
<containerdata></containerdata>
<urmdeb>NO</urmdeb>
<activation>ACTIVE</activation>
<trigger>TEST</trigger>
<level>ALL</level>
<sesstype>TCP</sesstype>
<sessaddr>9.30.247.101</sessaddr>
<sessport>8001</sessport>
<commandfile>*</commandfile>
<preferencefile>*</preferencefile>
<otheropts></otheropts>
</profile>
Server response code = 201
Server response msg = Profile_Created_OK
Server response details = <?xml version="1.0"?><profile><profileversion>0102</profileversion><serviceid>DBGTPROF</serviceid><clientversion>0102</clientversion><serverversion>0102</serverversion></profile>

```

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
-- Populated 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
-- Modified 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. 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.

For a comprehensive multimedia overview of IBM software support resources, see the IBM Education Assistant presentation “IBM Software Support Resources for System z Enterprise Development Tools and Compilers products” at <http://publib.boulder.ibm.com/infocenter/ieduasst/stgv1r0/index.jsp?topic=/com.ibm.iea.debugt/debugt/6.1z/TrainingEducation/SupportInfoADTools/player.html>.

- “Searching knowledge bases”
- “Getting fixes” on page 561
- “Subscribing to support updates” on page 561
- “Contacting IBM Support” on page 562

Searching knowledge bases

You can search the available knowledge bases to determine whether your problem was already encountered and is already documented.

- Searching the information center
- Searching product support documents

Searching the information center

You can find this publication and documentation for many other products in the IBM System z Enterprise Development Tools & Compilers information center at <http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp>. Using the information center, you can search product documentation in a variety of ways. You can search across the documentation for multiple products, search across a subset of the product documentation that you specify, or search a specific set of topics that you specify within a document. Search terms can include exact words or phrases, wild cards, and Boolean operators.

To learn more about how to use the search facility provided in the IBM System z Enterprise Development Tools & Compilers information center, you can view the multimedia presentation at <http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=/com.ibm.help.doc/InfoCenterTour800600.htm>.

Searching product support documents

If you need to look beyond the information center to answer your question or resolve your problem, you can use one or more of the following approaches:

- Find the content that you need by using the IBM Support Portal at www.ibm.com/software/support or directly at www.ibm.com/support/entry/portal.

The IBM Support Portal is a unified, centralized view of all technical support tools and information for all IBM systems, software, and services. The IBM

Support Portal lets you access the IBM electronic support portfolio from one place. You can tailor the pages to focus on the information and resources that you need for problem prevention and faster problem resolution.

Familiarize yourself with the IBM Support Portal by viewing the demo videos at https://www.ibm.com/blogs/SPNA/entry/the_ibm_support_portal_videos?lang=en_us about this tool. These videos introduce you to the IBM Support Portal, explore troubleshooting and other resources, and demonstrate how you can tailor the page by moving, adding, and deleting portlets.

Access a specific IBM Software Support site:

- Application Performance Analyzer for z/OS Support
 - Debug Tool for z/OS Support
 - Enterprise COBOL for z/OS Support
 - Enterprise PL/I for z/OS Support
 - Fault Analyzer for z/OS Support
 - File Export for z/OS Support
 - File Manager for z/OS Support
 - WebSphere® Studio Asset Analyzer for Multiplatforms Support
 - Workload Simulator for z/OS and OS/390 Support
- Search for content by using the IBM masthead search. You can use the IBM masthead search by typing your search string into the Search field at the top of any [ibm.com](http://www.ibm.com)® page.
 - Search for content by using any external search engine, such as Google, Yahoo, or Bing. If you use an external search engine, your results are more likely to include information that is outside the [ibm.com](http://www.ibm.com) domain. However, sometimes you can find useful problem-solving information about IBM products in newsgroups, forums, and blogs that are not on [ibm.com](http://www.ibm.com). Include "IBM" and the name of the product in your search if you are looking for information about an IBM product.
 - The IBM Support Assistant (also referred to as ISA) is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. It provides quick access to support-related information. You can use the IBM Support Assistant to help you in the following ways:
 - Search through IBM and non-IBM knowledge and information sources across multiple IBM products to answer a question or solve a problem.
 - Find additional information through product and support pages, customer news groups and forums, skills and training resources and information about troubleshooting and commonly asked questions.

In addition, you can use the built in Updater facility in IBM Support Assistant to obtain IBM Support Assistant upgrades and new features to add support for additional software products and capabilities as they become available.

For more information, and to download and start using the IBM Support Assistant for IBM System z Enterprise Development Tools & Compilers products, please visit http://www.ibm.com/support/docview.wss?rs=2300&context=SSFMHB&dc=D600&uid=swg21242707&loc=en_US&cs=UTF-8&lang=en.

General information about the IBM Support Assistant can be found on the IBM Support Assistant home page at <http://www.ibm.com/software/support/isa>.

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 Application Performance Analyzer for z/OS
- Latest PTFs for Debug Tool for z/OS
- Latest PTFs for Fault Analyzer for z/OS
- Latest PTFs for File Export for z/OS
- Latest PTFs for File Manager for z/OS
- Latest PTFs for Optim™ Move for DB2
- Latest PTFs for WebSphere Studio Asset Analyzer for Multiplatforms
- Latest PTFs for Workload Simulator for z/OS and OS/390

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 stay informed of important information about the IBM products that you use, you can subscribe to updates. By subscribing to receive updates, you can receive important technical information and updates for specific Support tools and resources. You can subscribe to updates by using the following:

- RSS feeds and social media subscriptions
- My Notifications

RSS feeds and social media subscriptions

For general information about RSS, including steps for getting started and a list of RSS-enabled IBM web pages, visit the IBM Software Support RSS feeds site at <http://www.ibm.com/software/support/rss/other/index.html>. For information about the RSS feed for the IBM System z Enterprise Development Tools & Compilers information center, refer to the [Subscribe to information center updates](#) topic in the information center at http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/topic/com.ibm.help.doc/subscribe_info.html.

My Notifications

With My Notifications, you can subscribe to Support updates for any IBM product. You can specify that you want to receive daily or weekly email announcements. You can specify what type of information you want to receive (such as publications, hints and tips, product flashes (also known as alerts), downloads, and drivers). My Notifications enables you to customize and categorize the products about which you want to be informed and the delivery methods that best suit your needs.

To subscribe to Support updates, follow the steps below.

1. Click My notifications to get started. Click **Subscribe now!** on the page.
2. Sign in My notifications with your IBM ID. If you do not have an IBM ID, create one ID by following the instructions.

3. After you sign in My notifications, enter the name of the product that you want to subscribe in the **Product lookup** field. The look-ahead feature lists products matching what you typed. If the product does not appear, use the **Browse for a product** link.
4. Next to the product, click the **Subscribe** link. A green check mark is shown to indicate the subscription is created. The subscription is listed under Product subscriptions.
5. To indicate the type of notices for which you want to receive notifications, click the **Edit** link. To save your changes, click the **Submit** at the bottom of the page.
6. To indicate the frequency and format of the email message you receive, click **Delivery preferences**. Then, 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

IBM Support provides assistance with product defects, answering FAQs, and performing rediscovery.

After trying to find your answer or solution by using other self-help options such as technotes, you can contact IBM Support. Before contacting IBM Support, your company must have an active IBM maintenance contract, and you must be authorized to submit problems to IBM. For information about the types of available support, see the information below or refer to the Support portfolio topic in the Software Support Handbook at <http://www14.software.ibm.com/webapp/set2/sas/f/handbook/offerings.html>.

- For IBM distributed software products (including, but not limited to, Tivoli[®], Lotus[®], and Rational products, as well as DB2 and WebSphere products that run on Windows, or UNIX operating systems), enroll in Passport Advantage[®] in one of the following ways:

Online

Go to the Passport Advantage Web site at http://www.lotus.com/services/passport.nsf/WebDocs/Passport_Advantage_Home and click **How to Enroll**.

By phone

For the phone number to call in your country, go to the Contacts page of the *IBM Software Support Handbook* on the Web at <http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html> and click the name of your geographic region.

- For customers with Subscription and Support (S & S) contracts, go to the Software Service Request Web site at <http://www.ibm.com/support/servicerequest>.
- For customers with IBMLink, CATIA, Linux, S/390[®], iSeries, pSeries, zSeries, and other support agreements, go to the IBM Support Line Web site at <http://www.ibm.com/services/us/index.wss/so/its/a1000030/dt006>.
- For IBM eServer[™] software products (including, but not limited to, DB2 and WebSphere products that run in zSeries, pSeries, and iSeries environments), you can purchase a software maintenance agreement by working directly with an IBM sales representative or an IBM Business Partner. For more information about support for eServer software products, go to the IBM Technical Support Advantage Web site at <http://www.ibm.com/servers/eserver/techsupport.html>.

If you are not sure what type of software maintenance contract you need, call 1-800-IBMSERV (1-800-426-7378) in the United States. From other countries, go to the Contacts page of the *IBM Software Support Handbook* on the Web at <http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html> and click the name of your geographic region for phone numbers of people who provide support for your location.

Complete the following steps to contact IBM Support with a problem:

1. "Define the problem and determine the severity of the problem"
2. "Gather diagnostic information"
3. "Submit the problem to IBM Support" on page 564

To contact IBM Software support, follow these steps:

Define the problem and determine the severity of the problem

Define the problem and determine severity of the problem When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Support can help you solve the problem efficiently.

IBM Support needs you to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting. Use the following criteria:

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

The 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.

For more information, see the Getting IBM support topic in the Software Support Handbook at <http://www14.software.ibm.com/webapp/set2/sas/f/handbook/getsupport.html>.

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 Application Performance Analyzer for z/OS: http://www.ibm.com/support/docview.wss?rs=2300&context=SSFMHB&q1=mustgather&uid=swg21265542&loc=en_US&cs=utf-8&lang=en
- Mustgather: Read first for problems encountered with Debug Tool for z/OS: http://www.ibm.com/support/docview.wss?rs=615&context=SSGTSD&q1=mustgather&uid=swg21254711&loc=en_US&cs=utf-8&lang=en
- Mustgather: Read first for problems encountered with Fault Analyzer for z/OS: http://www.ibm.com/support/docview.wss?rs=273&context=SSXJAJ&q1=mustgather&uid=swg21255056&loc=en_US&cs=utf-8&lang=en
- Mustgather: Read first for problems encountered with File Manager for z/OS: http://www.ibm.com/support/docview.wss?rs=274&context=SSXJAV&q1=mustgather&uid=swg21255514&loc=en_US&cs=utf-8&lang=en
- Mustgather: Read first for problems encountered with Enterprise COBOL for z/OS: http://www.ibm.com/support/docview.wss?rs=2231&context=SS6SG3&q1=mustgather&uid=swg21249990&loc=en_US&cs=utf-8&lang=en
- Mustgather: Read first for problems encountered with Enterprise PL/I for z/OS: http://www.ibm.com/support/docview.wss?rs=619&context=SSY2V3&q1=mustgather&uid=swg21260496&loc=en_US&cs=utf-8&lang=en

If the product does not have a Mustgather document, please 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 one of three ways:

Online using the IBM Support Portal

Click **Service request** on the IBM Software Support site at <http://www.ibm.com/software/support>. On the right side of the Service request page, expand the Product related links section. Click Software support (general) and select ServiceLink/IBMLink to open an Electronic Technical Response (ETR). Enter your information into the appropriate problem submission form.

Online using the Service Request tool

The Service Request tool can be found at <http://www.ibm.com/software/support/servicerequest>.

By phone

Call 1-800-IBMSERV (1-800-426-7378) in the United States or, from other countries, go to the Contacts page of the *IBM Software Support Handbook* at <http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html> and click the name of your geographic region.

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.

After a Problem Management Record (PMR) is open, you can submit diagnostic MustGather data to IBM using one of the following methods:

- FTP diagnostic data to IBM. For more information, refer to <http://www.ibm.com/support/docview.wss?rs=615&tuid=swg21154524>.
- If FTP is not possible, e-mail diagnostic data to techsupport@mainz.ibm.com. You must add PMR xxxxx bbb ccc in the subject line of your e-mail. xxxxx is your PMR number, bbb is your branch office, and ccc is your IBM country code. Go to <http://itcenter.mainz.de.ibm.com/ecurep/mail/subject.html> for more details.

Always update your PMR to indicate that data has been sent. You can update your PMR online or by phone as described above.

Appendix M. 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 Debug Tool.

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

The *IBM System z Enterprise Development Tools & Compilers Information Center*, and its related publications, are accessibility-enabled. The accessibility features of the information center are described at http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/topic/com.ibm.help.doc/accessibility_info.html.

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 the *IBM System z Enterprise Development Tools & Compilers Information Center*

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.

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 illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or functions of these programs.

Programming interface information

This book 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 Debug Tool.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are 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.

Java and all Java-based trademarks and logos are trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

MasterCraft is a trademark of Tata Consultancy Services Ltd.

Glossary

This glossary defines technical terms and abbreviations used in *Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool.

C

CADP A CICS-supplied transaction used for managing debugging profiles from a 3270 terminal.

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 Debug Tool, the actions of Debug Tool 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 *nodate*.

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

Debug Tool Control utility, a CICS transaction that enables the user to identify which CICS programs to debug.

Debug Tool procedure

A sequence of Debug Tool commands delimited by a PROCEDURE and a corresponding END command.

Debug Tool variable

A predefined variable that provides information about the user's program that the user can use during a session. All of the Debug Tool variables begin with %, for example, %BLOCK 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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool, 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 Debug Tool.

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 Debug Tool 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 Debug Tool 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 Debug Tool 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 Debug Tool 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. Debug Tool 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 Debug Tool 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).

Bibliography

Debug tool publications

Using CODE/370 with VS COBOL II and OS PL/I, SC09-1862

Debug Tool for z/OS

You can access Debug Tool publications through the IBM System z Enterprise Development Tool and Compilers information center. You can receive RSS feeds about updates to the information center by following the instructions in the topic "Subscribe to information center updates", which is in the IBM System z Enterprise Development Tools and Compilers information center.

Debug Tool User's Guide, SC14-7600

Debug Tool Coverage Utility User's Guide and Messages, SC27-4651

Debug Tool Reference and Messages, SC27-4652

Debug Tool Reference Summary, SC14-7602

Debug Tool API User's Guide and Reference, SC27-4654

Debug Tool Customization Guide, SC14-7601

Debug Tool Program Directory, GI13-3004

COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM: User's Guide, SC26-9400-02

Program Directory for IBM COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM, GI10-5080-04

Japanese Program Directory for IBM COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM, GI10-6976-02

Problem Determination Tools Common Component Program Directory, GI10-8969

Problem Determination Tools for z/OS Common Component Customization Guide and User Guide, SC19-4159

High level language publications

z/OS C and C++

Compiler and Run-Time Migration Guide, GC09-4913

Curses, SA22-7820

Language Reference, SC09-4815

Programming Guide, SC09-4765

Run-Time Library Reference, SA22-7821

User's Guide, SC09-4767

Enterprise COBOL for z/OS, Version 5

Customization Guide, SC14-7380

Language Reference, SC14-7381

Programming Guide, SC14-7382

Migration Guide, GC14-7383

Program directory, GI11-9180

Licensed Program Specifications, GI11-9181

Enterprise COBOL for z/OS, Version 4

Compiler and Runtime Migration Guide, GC23-8527

Customization Guide, SC23-8526

Licensed Program Specifications, GI11-7871

Language Reference, SC23-8528

Programming Guide, SC23-8529

Enterprise COBOL for z/OS and OS/390, Version 3

Migration Guide, GC27-1409

Customization, GC27-1410

Licensed Program Specifications, GC27-1411

Language Reference, SC27-1408

Programming Guide, SC27-1412

COBOL for OS/390 & VM

Compiler and Run-Time Migration Guide, GC26-4764

Customization under OS/390, GC26-9045

Language Reference, SC26-9046

Programming Guide, SC26-9049

Enterprise PL/I for z/OS, Version 4

Language Reference, SC14-7285

Licensed Program Specifications, GC14-7283

Messages and Codes, GC14-7286

Compiler and Run-Time Migration Guide, GC14-7284

Programming Guide, GI11-9145

Enterprise PL/I for z/OS and OS/390, Version 3

Diagnosis, SC27-1459

Language Reference, SC27-1460

Licensed Program Specifications, GC27-1456

Messages and Codes, SC27-1461

Migration Guide, GC27-1458

Programming Guide, SC27-1457

VisualAge PL/I for OS/390

Compiler and Run-Time Migration Guide, SC26-9474

Diagnosis Guide, SC26-9475

Language Reference, SC26-9476

Licensed Program Specifications, GC26-9471

Messages and Codes, SC26-9478

Programming Guide, SC26-9473

PL/I for MVS & PM

Compile-Time Messages and Codes, SC26-3229

Compiler and Run-Time Migration Guide, SC26-3118

Diagnosis Guide, SC26-3149

Installation and Customization under MVS, SC26-3119
Language Reference, SC26-3114
Licensed Program Specifications, GC26-3116
Programming Guide, SC26-3113
Reference summary, SX26-3821

Related publications

CICS

Application Programming Guide, SC34-6231
Application Programming Primer, SC34-0674
Application Programming Reference, SC34-6232

DB2 Universal Database™ for z/OS

Administration Guide, SC18-7413
Application Programming and SQL Guide, SC18-7415
Command Reference, SC18-7416
Data Sharing: Planning and Administration, SC18-7417
Installation Guide, GC18-7418
Messages and Codes, GC18-7422
Reference for Remote RDRA Requesters and Servers*, SC18-7424
Release Planning Guide, SC18-7425
SQL Reference, SC18-7426
Utility Guide and Reference, SC18-7427

IMS

IMS Application Programming: Database Manager, SC27-1286
IMS Application Programming: EXEC DLI Commands for CICS & IMS, SC27-1288
IMS Application Programming: Transaction Manager, SC27-1289

TSO/E

Command Reference, SA22-7782
Programming Guide, SA22-7788
System Programming Command Reference, SA22-7793
User's Guide, SA22-7794

z/OS

MVS JCL Reference, SA22-7597
MVS JCL User's Guide, SA22-7598
MVS System commands, SA22-7627

z/OS Language Environment

Concepts Guide, SA22-7567
Customization, SA22-7564
Debugging Guide, GA22-7560
Programming Guide, SA22-7561
Programming Reference, SA22-7562
Run-Time Migration Guide, GA22-7565

Vendor Interfaces, SA22-7568

Writing Interlanguage Communication Applications, SA22-7563

Softcopy publications

Online publications are distributed on CD-ROMs and can be ordered through your IBM representative. *Debug Tool User's Guide*, *Debug Tool Customization Guide*, and *Debug Tool Reference and Messages* are distributed on the following collection kit:

SK5T-8871

Online publications can also be downloaded from the IBM website. Visit the IBM website for each product to find online publications for that product.

Index

Special characters

__ctest() function 135
./E, BTS Environment command 106
.mdbg
 how Debug Tool locates 449
.mdbg file 442
.mdbg file, how to create 40, 45
%CONDITION variable
 for PL/I 309
%PATHCODE variable
 for C and C++ 322
 for PL/I 308
 values for COBOL 293
&PGMNAME 105
&userid 547
&USERID 105
#pragma 43
 specifying TEST compiler option 43
 specifying TEST run-time option with 122

A

ABEND 4038 418
abnormal end of application, setting breakpoint at 409
accessing PL/I program variables 311
ALL suboption of TEST compiler option (PL/I), effect of 38
ALL, how Enterprise COBOL for z/OS, Version 4, handles 33
ALLOCATE command
 managing file allocations 207
allocating Debug Tool files
 example of 142
allocating Debug Tool load library data set
 example of 142
ALTER PROCEDURE statement, example of 85
applications 389
Applid 92
assembler
 debugging a program in full-screen mode
 displaying variable or storage 269
 finding storage overwrite errors 271
 getting a function traceback 270
 modifying variables or storage 269
 multiple CUs in single assembly 267
 stopping at assembler routine call 269
 stopping when condition is true 270
 debugging non-reentrant 347
 defining CU as 266
 how Debug Tool locates EQALANGX files 448
 loading debug data of 266
 QUERY LOCATION 269
 reappearing 267
 restrictions 348
 assembler code using instructions as data 351
 detectable self-modifying 352
 non-detectable self-modifying 352
 non-Language Environment 350
 self-modifying 351
 while debugging MAIN program 350
 with STORAGE run-time option 350
 sample program for debugging 263

assembler (*continued*)
 self-modifying code, restrictions 357
assembler program
 loading debug information 343
 locating EQALANGX 343
 making assembler CUs known to Debug Tool 344
assembler programs
 assembling, requirements 75
 requirements for debugging 75
 using Debug Tool Utilities to assemble and create 77
assembler, definition of xvii
assembling your program, requirements for 75
assigning values to variables 291, 321
AT commands
 AT CALL
 breakpoints, for C++ 339
 AT ENTRY
 breakpoints, for C++ 338
 AT EXIT
 breakpoints, for C++ 338
attention interrupt
 effect of during Dynamic Debug 210
 effect of during interactive sessions 210
 how to initiate 210
 required Language Environment run-time options 210
attributes of variables 412
automatic saving and restoring of settings, breakpoints, and
 monitor specifications 193
automatic saving and restoring of settings, breakpoints, and
 monitor specifications; disabling 194
available only with programs compiled with
 L prefix command 16
 M prefix command 16

B

base address, how to specify for MEMORY command 181
base address, using in Memory window 181
batch mode 118
 debugging DB2 programs in 363
 debugging IMS programs in 372
 description of 5
 for non-Language Environment programs 143
 starting Debug Tool in 137
 using Debug Tool in 523
binder APIs 427
blanks, significance of 286
BLOCK suboption of TEST compiler option (PL/I), effect
 of 38
BLOCK, how Enterprise COBOL for z/OS, Version 4,
 handles 33
blocks and block identifiers
 using, for C 332
boundaries, setting for searches 179
breakpoint
 clearing 18
 implicit 118
 setting, introduction to 14
 skipping 18
 using DISABLE and ENABLE 18

- breakpoints
 - before calling a NULL function
 - in C 250
 - in C++ 262
 - before calling an invalid program, in COBOL 222
 - before calling an undefined program, in PL/I 238
 - halting if a condition is true
 - in C 245
 - in C++ 257
 - in COBOL 218
 - in LangX COBOL 229
 - in PL/I 235
 - halting when certain COBOL routines are called 216
 - halting when certain functions are called
 - in C 244
 - in C++ 255
 - in PL/I 234
 - halting when certain LangX COBOL routines are called 228
 - placing in IMS programs 379
 - recording, using SET AUTOMONITOR 186
 - setting a line 186
 - setting, in C++ 338
- breakpoints, setting in load modules that are not loaded 186
- breakpoints, setting in programs that are not active 186
- browse mode
 - enabling and disabling 54
 - introduction to 52
 - list of commands not permitted 52, 53
 - remote debug mode
 - list of actions not permitted 53
- BTS Environment command (./E), when to use 106

C

- compiling with c89 or c++ 64
- DEBUG compiler option, what it controls 39
- debugging a program in full-screen mode
 - calling a C function from Debug Tool 247
 - capturing output to stdout 246
 - debugging a DLL 248
 - displaying raw storage 247
 - displaying strings 247
 - finding storage overwrite errors 249
 - finding uninitialized storage errors 249
 - getting a function traceback 248
 - halting on line if condition true 245
 - halting when certain functions are called 244
 - modifying value of variable 245
 - setting breakpoint to halt 250
 - tracing run-time path for code compiled with TEST 248
 - when not all parts compiled with TEST 246
- GONUMBER compiler option 41
- LP64 versus ILP32 40
- OPT(1) or OPT(2) compiler options 42
- OPTIMIZE 40
- possible prerequisites 40, 41
- preparing, programs to debug 39
- sample program for debugging 241
- TEST compiler option, what it controls 41
- user defined functions 41
- when to Dynamic Debug facility with 40, 41
- C and C++
 - AT ENTRY/EXIT breakpoints 338
 - blocks and block identifiers 332

- C and C++ (*continued*)
 - choosing between TEST and DEBUG compiler option 39, 44
 - commands
 - summary 319
 - equivalents for Language Environment conditions 326
 - function calls for 324
 - notes on using 284
 - reserved keywords 325
 - when to use FORMAT(DWARF) 39, 44
- C/C++ file produced by DEBUG(FORMAT(DWARF)), how Debug Tool locates 448
- C/C++ source files, how Debug Tool locates 448
- C++
 - AT CALL breakpoints 339
 - DEBUG compiler option, what it controls 44
 - debugging a program in full-screen mode
 - calling a C++ function from Debug Tool 259
 - capturing output to stdout 258
 - debugging a DLL 259
 - displaying raw storage 259
 - displaying strings 259
 - finding storage overwrite errors 261
 - finding uninitialized storage errors 261
 - getting a function traceback 260
 - halting on a line if condition true 257
 - modifying value of variable 256
 - setting a breakpoint to halt 255, 262
 - tracing the run-time path 260
 - viewing and modifying data members 257
 - when not all parts compiled with TEST 257
 - examining objects 339
 - GONUMBER compiler option 46
 - LP64 versus ILP32 46
 - OPT(1) or OPT(2) compiler options 47
 - OPTIMIZE 46
 - overloaded operator 338
 - possible prerequisites 45
 - preparing, programs to debug 44
 - sample program for debugging 251
 - setting breakpoints 338
 - stepping through C++ programs 338
 - template in C++ 338
 - TEST compiler option, what it controls 46
 - user defined functions 46
 - using slashes to enter comments 287
 - when to Dynamic Debug facility with 45
- CADP
 - how to start Debug Tool with 149
 - how to use 99
- CAF (call access facility), using to start DB2 program 364
- call access facility (CAF), using to start DB2 program 364
- call_sub function, how to debug DB2 stored procedures invoked by 57
- capturing output to stdout
 - in C 246
 - in C++ 258
- CC...CC, Monitor prefix command 172
- CCCA 59
- CEE3CBTS 425
- CEEBXITA 87
 - description of how it works 105
- CEEBXITA, comparing two methods of linking 109
- CEEBXITA, specifying message display level in 108
- CEEBXITA, specifying naming pattern in 107
- CEEROPT, using
 - for IMS programs 101

- CEETEST
 - description 128
 - examples, for C 130
 - examples, for COBOL 131
 - examples, for PL/I 132
 - Starting Debug Tool with 127
 - using 372
- CEEUOPT runtime options module 81
- CEEUOPT to start Debug Tool under CICS, using 150
- CEEUOPT, using
 - for IMS programs 101
- changing how Monitor window displays values 196
- changing physical window layout in the session panel 274
- changing the value of a variable, introduction to 17
- character set 283
- characters, searching 178
- CICS
 - breakpoints, pattern-match 383
 - CADP, how to use 99
 - choosing a debugging mode for 50
 - DPL 51
 - DTCN profile, creating a 88
 - DTCN profiles, displaying list of 91
 - DTCN, fields on Advanced Options 98
 - DTCN, fields on Menu 2 97
 - DTCN, fields on Primary Menu 92
 - DTST transaction, description of storage window 530
 - DTST transaction, navigating through DTST storage window 529
 - DTST transaction, starting the 527
 - DTST transaction, syntax of the 532
 - DTST transaction, using to modify storage 529
 - list of general tasks to complete for 87
 - non-Language Environment programs, passing runtime parameters to 100
 - non-Language Environment programs, starting Debug Tool for 99
 - pseudo-conversational program 386
 - region, reloading programs into an active 543
 - requirements for using Debug Tool in 381
 - restoring breakpoints 386
 - restrictions for debugging 386
 - saving breakpoints 386
 - starting Debug Tool under 147
 - starting the log file 387
 - WAIT option 51
- CICS debugging
 - RLIM processing 387
- closing automonitor section of Monitor window 200
- closing Debug Tool physical windows 275
- COBOL 213
 - CCCA 59
 - command format 289
 - debugging a program in full-screen mode
 - capturing I/O to system console 219
 - displaying raw storage 219
 - finding storage overwrite errors 222
 - generating a run-time paragraph trace 221
 - modifying the value of a variable 217
 - setting a breakpoint to halt 216
 - setting breakpoint to halt 222
 - stopping on line if condition true 218
 - tracing the run-time path 220
 - when not all parts compiled with TEST 218, 229
 - debugging COBOL classes 299
 - debugging VS COBOL II programs 300
 - finding listing 300
- COBOL (*continued*)
 - EJPD suboption 27
 - Enterprise, L prefix command only available with 16
 - Enterprise, M prefix command only available with 16
 - FACTORY 299
 - how Debug Tool locates separate debug file 447
 - list of effect of ALL compiler option 33
 - list of effect of BLOCK compiler option 33
 - list of effect of NOSYM compiler option 32
 - list of effect of NOTEST compiler option 30
 - list of effect of PATH compiler option 32
 - list of effect of STMT compiler option 32
 - Load Module Analyzer 58
 - non-Language Environment, QUERY LOCATION 228
 - NONE and NOHOOK with optimized programs 31
 - note on using H constant 287
 - notes on using 284
 - OBJECT 299
 - OPT compiler option 27
 - optimized programs, debugging 396
 - paragraph names, finding 180
 - paragraph trace, generating a COBOL run-time 221
 - possible prerequisites 29
 - QUERY LOCATION 217
 - reserved keywords 290
 - RESIDENT compiler option 29
 - restrictions on accessing, data 191
 - run-time options 121
 - sample program for debugging 213, 225
 - SOURCE compiler option 29
 - TEST compiler option, what suboptions to specify 27
 - variables, using with Debug Tool 291
 - when to Dynamic Debug facility with 27
 - why you need to specify SYM 29
 - Working-Storage Section, displaying 198
- COBOL listing, data set 439
- COBOL, reusable runtime environments 401
- coexistence of Debug Tool with other debuggers 414
- coexistence with unsupported HLL modules 414
- colors
 - changing in session panel 276
- columnar format, displaying value in Monitor window in 204
- command
 - syntax diagrams xviii
- command format
 - for COBOL 289
- command line, Debug Tool 169
- Command pop-up window, changing size of 164
- command sequencing, full-screen mode 170
- commands
 - abbreviating 284
 - DTSU, using to debug DB2 program 364
 - for C and C++, Debug Tool subset 319
 - for PL/I, Debug Tool subset 307
 - getting online help for 288
 - interpretive subset
 - description 406
 - multiline 285
 - PLAYBACK 18
 - prefix, using in Debug Tool 171
 - truncating 284
 - TSO, using to debug DB2 program 364
- commands (system), entering in Debug Tool 171
- commands file 118, 442
 - example of specifying 137
 - using log file as 184

- commands file (*continued*)
 - using session log as 119
- Commands File
 - in DTCN, description of 97
- commands file, how to create a 182
- commands, Debug Tool
 - COBOL compiler options in effect 290
 - entering on the session panel 167
 - entering using program function keys 173
 - order of processing 170
 - retrieving with RETRIEVE command 174
 - that resemble COBOL statements 289
- COMMANDSDSN, EQAOPTS command 183, 442
- Commarea data 98
- Commarea offset 98
- comments, inserting into command stream 287
- Common pop-up window, how to enter commands in 175
- common_parameters, when to use 10
- compile unit 169
 - general description 407
 - name area, Debug Tool 169
 - qualification of, for C and C++ 335
- compile units known to Debug Tool, displaying list of 209
- compiler options
 - COBOL 27
 - how to choose, for PL/I 34
 - suggested 25
 - which options to use for COBOL 27
- compiling
 - a C program on an HFS file system 65
 - a C++ program on an HFS file system 66
 - an OS/VS COBOL program 58
 - Enterprise PL/I program on HFS file system 64
 - programs, introduction to 11
- condition
 - handling of 309, 410
 - Language Environment, C and C++ equivalents 326
- considerations
 - when using the TEST run-time option 117
- constants
 - Debug Tool interpretation of HLL 406
 - entering 287
 - HLL 406
 - PL/I 313
 - using in expressions, for COBOL 296
- constructor, stepping through 338
- Container data 98
- Container name 98
- Container offset 98
- continuation character 170
 - for COBOL 289
 - using in full-screen 285
- continuing lines 285
- continuous display 197
- copying
 - JCL into a setup file using DTSU 124
- CREATE PROCEDURE statement, example of 84
- creating
 - setup file using Debug Tool Utilities 123
- CRTE 51
- CSECT, debugging multiple, in one assembly 268
- CSECT, loading multiple, in one assembly 268
- CU(s) 92
- CURSORS command
 - using 175, 176
- cursor commands
 - CLOSE 275

- cursor commands (*continued*)
 - CURSORS 176
 - FIND 178
 - OPEN 275
 - SCROLL 160, 176
 - SIZE 275
 - using in Debug Tool 173
 - WINDOW ZOOM 276
- customer support 562
- customizing
 - PF keys 273
 - Profile panel 117
 - profile settings 277
 - session settings 273
- CWI, Language Environment 425

D

- data only modules, debugging 428
- DATA parameter
 - restrictions on accessing COBOL data 191
- data sets
 - COBOL listing 439
 - PL/I listing 441
 - PL/I source 440
 - separate debug file 441
 - specifying 184
 - used by Debug Tool 439
- data type of variable, displaying in Monitor window the 199
- DB2
 - assembling with assembler programs 80
 - compiling with C or C++ programs 80
 - compiling with COBOL programs 79
 - compiling with PL/I programs 79
 - DB2 programs for debugging 79
 - linking programs 81
 - using Debug Tool with 363
- DB2 programs
 - what files to keep 79
- DB2 programs, binding 82
- DB2 stored procedures
 - compiling or assembling options to use 84
 - debugging modes supported 83
 - NUMTCB 83
 - restrictions 127
 - specifying TEST runtime options through EQADDCXT 84
 - starting Debug Tool from 153
 - using Debug Tool with 367
 - what to do before debugging 83
- DBCS
 - using with C 284
 - using with COBOL 293
 - using with Debug Tool commands 283
- DEBUG and TEST compiler option, choosing between 39, 44
- DEBUG compiler options 39, 45
- debug mode
 - delay 431, 434
- debug session
 - ending 210
 - recording 162
 - starting 151
 - starting your program 151
- Debug Tool
 - C and C++ commands, interpretive subset 319
 - COBOL commands, interpretive subset 289
 - commands, subset 406
 - condition handling 410

- Debug Tool (*continued*)
 - data sets 439
 - enhancing performance of 80
 - evaluation of HLL expressions 405
 - exception handling, for C and C++ and PL/I 411
 - interfaces 4
 - interpretation of HLL variables 406
 - list of supported compilers 3
 - list of supported subsystems 4
 - multilanguage programs, using 411
 - PL/I commands, interpretive subset 307
 - starting at different points 118
 - starting under CICS 147
 - starting under MVS in TSO 141
 - starting your program with 151
 - starting, by using Debug Tool Utilities 123
 - stopping, session 19
 - terminology xvi
 - using in batch mode 523
- Debug Tool Setup Utility 123
- Debug Tool Utilities
 - brief description of Load Module Analyzer 8
 - brief description on preparing assembler 6
 - creating and managing setup files 7
 - creating private message region for IMS program 376
 - creating setup file for IMS program 376, 377, 525
 - Deferred Breakpoints 9
 - Delay Debug Profile 8
 - how to start 9
 - how to use, to link-edit 77
 - IMS Transaction and User ID Cross Reference Table 9
 - instructions for compiling or assembling 452
 - instructions for modifying and using a setup file 455
 - instructions for running a program in batch 456
 - JCL Wizard 9
 - list of all utilities in 6
 - managing debugging profiles 8
 - Non-CICS Debug Session Start and Stop Message Viewer 9
 - overview of code coverage tasks 7
 - overview of IMS BTS Debugging 8
 - overview of IMS program preparation tasks 7
 - overview of JCL file conversion 8
 - overview of JCL for Batch Debugging 8
 - overview of Job Cards 6
 - overview of program preparation tasks 6
 - specifying TEST runtime options for IMS program 102
 - starting your program 126
 - using to assemble and create 77
- Debug Tool Utilities, general instructions on how to use 63
- debuggers, coexistence with other 414
- debugging
 - CICS programs 381
 - CICS programs, choosing mode 50
 - COBOL classes 299
 - DB2 programs 363
 - DB2 stored procedures 367
 - DLL
 - in C 248
 - in C++ 259
 - IMS programs, choosing mode 51
 - in full-screen mode 157
 - ISPF applications 389
 - multithreading programs 415
 - non-Language Environment programs 401
 - UNIX System Services programs 399
- debugging profiles
 - how to create one with DTCN 89
- declared data type, displaying characters in their 203
- declared data type, modifying characters that cannot be displayed in their 203
- declaring session variables
 - for C 322
 - for COBOL 295
- deferred, description of 267
- deferring an LDD command 228
- DESCRIBE ALLOCATIONS command
 - managing file allocations 207
- DESCRIBE command
 - using 334
- description of how Debug Tool locates CICS tasks to debug 148
- destructor, stepping through 338
- diagnostics, expression, for C and C++ 327
- DISABLE command 384
- disassembly
 - changing program in disassembly view 358
 - differences between SET ASSEMBLER and SET DISASSEMBLY 343, 355
 - displaying registers 358
 - displaying storage 358
 - modifying registers 358
 - modifying storage 358
 - performing single-step operations 357
 - restrictions on what you can debug 358
 - self-modifying code, restrictions 357
 - setting breakpoints 357
 - what you can do is disassembly view 355
- disassembly view, description of 356
- disassembly view, how to start 356
- Display Id 92
 - in DTCN, description of 96
- displaying
 - environment information 334
 - halted location 180
 - lines at top of window, Debug Tool 178
 - raw storage
 - in C 247
 - in C++ 259
 - in COBOL 219
 - in PL/I 236
 - source or listing file in full-screen mode 165
 - strings
 - in C 247
 - in C++ 259
 - value of variable one time 196
 - values of COBOL variables 292
 - variable value 196
 - variables or storage
 - in LangX COBOL 229
- displaying list of known compile units 209
- displaying prefixes 428
- displaying the value of a variable, introduction to 15
- displaying variable value 196
- displaying Working-Storage Section 198
- DLL debugging
 - in C 248
 - in C++ 259
- documents, licensed xiii
- DOWN, SCROLL command 176
- DTCN
 - creating a profile 88
 - data entry verification 91

- DTCN (*continued*)
 - defining COMMAREA 90
 - description of 147
 - description of columns 92
 - description of Session Type 96
 - do not link to EQADCCXT with particular COBOL compilers 87
 - do not link to EQADCCXT with particular PL/I compilers 87
 - migrating from versions earlier than V10 94
 - modifying Language Environment options 98
 - using repository profile items 149
- DTCN Profiles 545, 548
- DTCNFORCEFORCEIP, how Transaction Id in DTCN works with 96
- DTCNFORCELOADMODID, how Transaction Id in DTCN works with 95
- DTCNFORCENETNAME, how Transaction Id in DTCN works with 96
- DTCNFORCETERMID, how Terminal Id in DTCN works with 93
- DTCNFORCETRANID, how Transaction Id in DTCN works with 93
- DTCNFORCEUSERID, how Transaction Id in DTCN works with 95
- DTNP 543
- DTSC 51
- DTSP Profile 545, 548
- DTST
 - syntax of 532
- DTST transaction
 - description of storage window 530
 - modifying storage after starting 529
 - navigating through storage window 529
 - starting the 527
 - syntax of the 532
- DWARF suboption of FORMAT compiler option, when to use 39, 44
- Dynamic Debug
 - attention interrupts, support for 210
- Dynamic Debug facility, how it works 48

E

- editing
 - setup file using Debug Tool Setup Utility 123
- elements, unsupported, for PL/I 316
- ENABLE command 384
- enclave
 - multiple, debugging interlanguage communication application in 423
 - non-Language Environment 127
 - starting 417
- ending
 - debug session 210
 - Debug Tool within multiple enclaves 418
- entering
 - commands on session panel 167
 - file allocation statements into setup file 124
 - program parameters into setup file 124
 - runtime option into setup file 124
- entering long command with Command pop-up window 175
- entering multiline commands without continuation 286
- entering PL/I statements, freeform 310
- Enterprise COBOL
 - compiler options to use 72
- Enterprise PL/I
 - restrictions 317
- Enterprise PL/I, definition of xviii
- EQADCCXT 87
- EQADCCXT user exit 119
- EQADDCXT
 - comparing DB2 RUNOPTS to 106
- EQADEBUG DD statement 167
- EQALANGX
 - creating for LangX COBOL 72
- EQALANGX file 440
 - how to create 76
- EQALANGX files, how Debug Tool locates 445, 448
- EQALMPFX 539
- EQALMPRM 540
- EQALOAD 427
- EQANMDBG
 - example 146
 - methods for starting Debug Tool with 143
 - passing parameters to 143, 145
 - using only EQANMDBG DD statement 145
 - using only PARM 144
- EQA_OPTS file, format options 442
- EQA_OPTS file, where to specify, in DTCN 98
- EQASET 373
 - when to run 106
- EQASTART, entering command 9
- EQASYSPF 539
- EQAUEDAT user exit 167
- EQAUOPT
 - how to create with Debug Tool Utilities 113
 - how to create with TIM 111
- EQAWLCEE 110
- EQAZLMAe 535
- EQUATE, SET command
 - description 273
- error numbers in Log window 209
- evaluating expressions
 - COBOL 295
 - HLL 405
- evaluation of expressions
 - C and C++ 327
- examining C++ objects 339
- examples
 - assembler
 - sample program for debugging 263
 - C
 - sample program for debugging 241
 - C and C++
 - assigning values to variables 321
 - blocks and block identifiers 334
 - expression evaluation 324
 - monitoring and modifying registers and storage 341
 - referencing variables and setting breakpoints 333
 - scope and visibility of objects 333
 - C++
 - displaying attributes 339
 - sample program for debugging 251
 - setting breakpoints 339
- CEETEST calls, for PL/I 132
- CEETEST function calls, for C 130
- CEETEST function calls, for COBOL 131
- changing point of view, general 409
- COBOL
 - %HEX function 297
 - %STORAGE function 297
 - assigning values to COBOL variables 291

- examples (*continued*)
 - COBOL (*continued*)
 - changing point of view 299
 - displaying results of expression evaluation 296
 - displaying values of COBOL variables 292
 - qualifying variables 298
 - sample program for debugging 213
 - using constants in expressions 296
 - declaring variables, for COBOL 295
 - displaying program variables 321
 - modifying setup files by using Debug Tool Utilities 451
 - OS/VS COBOL
 - sample program for debugging 225
 - PL/I
 - in PL/I 234
 - sample program for debugging 231
 - PLITEST calls for PL/I 134
 - preparing programs by using Debug Tool Utilities 451
 - remote debug mode 121
 - specifying TEST run-time option with #pragma 122
 - TEST run-time option 120
 - using #pragma for TEST compiler option 43
 - using constants 287
 - using continuation characters 285
 - using qualification 335
- exception handling for C and C++ and PL/I 411
- excluding programs 430
- EXEC CICS RETURN
 - under CICS 385
- explicit debug mode 429
- expressions
 - diagnostics, for C and C++ 327
 - displaying values, for C and C++ 320
 - displaying values, for COBOL 296
 - evaluation for C and C++ 323, 327
 - evaluation for COBOL 295
 - evaluation of HLL 405
 - evaluation, operators and operands for C 326
 - for PL/I 313
 - using constants in, for COBOL 296

F

- feedback codes, when to use 130
- FIND command
 - using with windows 178
- FIND command, setting boundaries with 179
- finding
 - characters or strings 178
 - storage overwrite errors
 - in assembler 271
 - in C 249
 - in C++ 261
 - in COBOL 222
 - in LangX COBOL 230
 - in PL/I 238
 - uninitialized storage errors
 - in C 249
 - in C++ 261
- finding COBOL paragraph names, example of 180
- fixes, getting 561
- FREE command
 - managing file allocations 207
- freeform input, PL/I statements 310
- full-screen mode
 - CICS, additional terminals 50
 - continuation character, using in 285

- full-screen mode (*continued*)
 - CURSOR 173
 - CURSOR command 176
 - debugging in 157
 - description of 5
 - example screen 13
 - introduction to 11
 - PANEL COLORS 276
 - PANEL LAYOUT 274
 - PANEL PROFILE 277
 - SCROLL 176
 - which why type of programs to use 50
 - WINDOW CLOSE 275
 - WINDOW OPEN 275
 - WINDOW SIZE 275
 - WINDOW ZOOM 276
- full-screen mode using the Terminal Interface Manager
 - description of 5
 - starting a debugging session 139
- function calls, for C and C++ 324
- function, calling C and C++ from Debug Tool
 - C 247
 - C++ 259
- function, unsupported for PL/I 316
- functions
 - PL/I 314
- functions, Debug Tool
 - %HEX
 - using with COBOL 297
 - %STORAGE
 - using with COBOL 297
 - using with COBOL 297

G

- global data 340
- global preferences file 442
- global scope operator 340
- GPFDSN, EQAOPTS command 442

H

- H constant (COBOL) 287
- halted location, displaying 180
- header fields, Debug Tool session panel 158
- help, online
 - for command syntax 288
- hexadecimal format, displaying values in 204
- hexadecimal format, how to display value of variable 204
- hexadecimal format, how to monitor value of variable 205
- hexadecimal format, monitoring values in 205
- HFS, compiling a C program on 65
- HFS, compiling a C++ program on 66
- HFS, compiling Enterprise PL/I program on 64
- highlighting, changing in Debug Tool session panel 276
- history area of Memory window 181
- history, Debug Tool command 174
 - retrieving previous commands 174
- hooks
 - compiling with 48
 - compiling with, PL/I 33
 - compiling without, COBOL 48
 - removing from application 393, 395
 - rules for placing in C programs 43
 - rules for placing in C++ programs 47, 48
- how to choose 39, 45

I

- I/O, COBOL
 - capturing to system console 219
- IBM Support Assistant, searching for problem resolution 559
- ignoring programs 429
- improving Debug Tool performance 393
- improving performance in multi-enclave environments 195
- IMS
 - choosing a debugging mode for 51
 - choosing method to specify TEST runtime options 101
 - JCL, sample doing replace link edit of CEEBXITA into CEEBINIT 103
 - making a user exit application-specific 102
 - making a user exit available installation-wide 102
 - making a user exit available region-wide 102
 - programs, debugging interactively 372
 - transaction isolation 369
- IMS MPP
 - debugging 372
 - preparing to debug 373
- INCLUDE files, how to automonitor variables in, while in remote debug mode 34
- INCLUDE files, how to debug PL/I 34
- information centers, searching for problem resolution 559
- information, displaying environmental 334
- initial programs, non-Language Environment 401
 - CICS assembler 402
 - non-Language Environment COBOL 402
- input areas, order of processing, Debug Tool 170
- INSPLOG
 - creating the log file 184
 - example of using 142
- INSPREF
 - example of using 142
- INSPSAFE
 - example of using 142
- instructions on how to compile a program with Debug Tool Utilities 63
- interfaces
 - batch mode 5
 - full-screen mode 5
 - full-screen mode using the Terminal Interface Manager 5
 - remote debug mode 6
- interfaces, description of 4
- interLanguage communication (ILC) application, debugging 423
- interlanguage programs, using with Debug Tool 411
- Internet
 - searching for problem resolution 559
- interpretive subset
 - general description 406
 - of C and C++ commands 319
 - of COBOL statements 289
 - of PL/I commands 307
- INTERRUPT, Language Environment run-time option 210
- IP Name/Addr 92
- IP Name/Address
 - in DTCN, description of 96
- IPv6 format (TCP/IP) 375
- ISPF
 - starting 171

J

- Java 425
- JCL sample, linking CEEBXITA into your program 109

- JCL sample, runs Debug Tool in batch mode 137
- JCL to create EQALANGX file 76
- JCL, list of changes to make to 61
- JNI 425

K

- keywords, abbreviating 284
- knowledge bases, searching for problem resolution 559

L

- Language Environment
 - conditions, C and C++ equivalents 326
 - EQADCCXT user exit 119
 - runtime options, precedence 119
 - user exit, link, into private copy of Language Environment runtime module 110
 - user exit, link, into your program 109
 - user exits, how to prepare 106
 - user exits, methods to modify sample assembler 106
- Language Environment user exit, create and manage data set used by 111
- LangX COBOL
 - %PATHCODE values 306
 - debugging a program in full-screen mode
 - displaying raw storage 229
 - finding storage overwrite errors 230
 - setting a breakpoint to halt 228
 - stopping on line if condition true 229
 - when not all parts compiled with TEST 229
 - how to prepare a 71
 - loading debug information for 303
 - session panel's appearance 304
- LDD command, example 343
- LEFT, SCROLL command 176
- licensed documents xiii
- line breakpoint, setting 186
- line continuation
 - for C 285
 - for COBOL 286
- link-edit assembler program
 - how to, by using Debug Tool Utilities 77
- linking
 - DB2 programs 81
 - EQADCCXT 87
- LIST %HEX command 204
- LIST command
 - use to display value of variable one time 197
- LIST commands
 - LIST STORAGE
 - using with PL/I 310
- List pop-up window, description of 164
- listing
 - find, OS PL/I 316
 - find, VS COBOL II 300
- listing files, how Debug Tool locates 445, 447
- literal constants, entering 287
- LLA 427
- Load Module Analyzer 58, 535
- LoadMod::>CU(s)
 - in DTCN, description of 93
- LoadMod(s) 92
- LOCATION, description of 159
- log file 183, 443
 - creating 184

- log file (*continued*)
 - using 183
 - using as a commands file 184
- log file, saving automonitor section to 201
- Log window
 - description 162
 - error numbers in 209
 - retrieving lines from 174
- log, session 119
- LOGDSN, EQAOPTS command 184, 443
- LOGDSNALLOC, EQAOPTS command 184
- low-level debugging 340

M

- MAIN DB2 stored procedures 83
- managing file allocations 207
- manual restoring of settings, breakpoints, and monitor specifications 194
- mdbg
 - how Debug Tool locates 449
- mdbg file 442
- MDBG, EQAOPTS command 40, 45
- memory
 - displaying, introduction to 17
- MEMORY command, using 206
- Memory window
 - description of 163
 - displaying with base address 206
 - history area, navigating with 181
 - opening an empty 181
- Memory window, addresses that span two columns 182
- Memory window, entering multiple commands in 172
- message display level, how to specify, in Language Environment user exit 108
- modifying
 - value of variable by typing over 206
 - value of variable by using command 205
- modifying value of a C variable 245
- MONITOR command
 - viewing output from, Debug Tool 161
- MONITOR LIST command, using to monitor variables 197
- MONITOR LIST TITLED WSS 198
- Monitor window
 - description 161
 - opening and closing 206, 275
- Monitor window, adding variables to 200
- Monitor window, replacing variables in 199
- monitoring 197
- monitoring storage in C++ 340
- more than one language, debugging programs with 411
- moving around windows in Debug Tool 175
- moving the cursor, Debug Tool 176
- moving to new level of Language Environment 110
- multilanguage programs, using with Debug Tool 411
- multiline commands
 - continuation character, using in 285
 - without continuation character 286
- multiple commands, entering in Memory window 172
- multiple enclaves
 - ending Debug Tool 418
 - interlanguage communication application, debugging 423
 - starting 417
- multithreading 415
 - restrictions 415
- MVS
 - starting Debug Tool using TEST run-time option 151

- MVS POSIX programs, debugging 399
- MVS, starting Debug Tool under 141

N

- name (default) of data set that saves settings, breakpoints, and monitors specifications 193
- NAMES 427
- NAMES command 431
 - using EQAOPTS 431
- NAMES EXCLUDE 430
- naming conflicts 427
- naming pattern, how to specify, in Language Environment user exit 107
- navigating session panel windows 175
- Netname 92
- NetName
 - in DTCN, description of 95
- NOHOOK suboption of TEST compiler option (PL/I), effect of 37
- NOMACGEN 347
- non-Language Environment
 - CICS
 - passing runtime parameters 100
 - Starting Debug Tool 99
 - defining as 227
 - how Debug Tool locates EQALANGX files 448
 - loading debug information 227
 - restrictions 306
- non-Language Environment initial programs 401
 - CICS assembler 402
 - non-Language Environment COBOL 402
- non-Language Environment programs
 - debugging 401
 - starting Debug Tool 143
- non-reentrant
 - breakpoints 348
 - debugging, assembler 347
 - variables 348
- NONE suboption of TEST compiler option (PL/I), effect of 37
- NOSYM suboption of TEST compiler option (C), effect of 42
- NOSYM suboption of TEST compiler option (PL/I), effect of 37
- NOTEST compiler option (C), effect of 42
- NOTEST compiler option (C++), effect of 47
- NOTEST compiler option (PL/I), effect of 36
- NOTEST suboption of TEST run-time option 117
- NUMTCB 83

O

- objects
 - C and C++, scope of 330
- opening Debug Tool physical windows 275
- opening Memory window with base address 206
- operators and operands for C 326
- OPT
 - C compiler option 42
 - C++ compiler option 47
 - COBOL compiler option 27
- OPTIMIZE, C compiler option 40
- OPTIMIZE, C++ compiler option 46
- optimized applications, debugging large 429
- optimized COBOL programs, modifying variables in 206, 292, 293

- optimized programs, compiling COBOL with NONE and NOHOOK 31
- optimized programs, debugging COBOL 396
- options module, CEEUOPT runtime 81
- OS PL/I programs, debugging 316
- OS PL/I, compiling 36
- OS PL/I, finding list for 316
- OS/VS COBOL 225
 - compiler options to use 71
 - restrictions 304
- output
 - C, capturing to stdout 246
 - C++, capturing to stdout 258
- overloaded operator 338
- overwrite errors, finding storage
 - in assembler 271
 - in C 249
 - in C++ 261
 - in COBOL 222
 - in LangX COBOL 230
 - in PL/I 238

P

- panel
 - header fields, session 158
 - Profile 277
- PANEL command (full-screen mode)
 - changing session panel colors and highlighting 276
- PANEL PROFILE command 167
- paragraph trace, generating a COBOL run-time 221
- PATH, how Enterprise COBOL for z/OS, Version 4, handles 32
- performance
 - enhancing Debug Tool's 80
- performance, improving Debug Tool 393
- PF keys
 - defining 273
 - using 173
- PF4 key, using 197
- PHASEIN 543
- physical
 - opening and closing windows 275
- physical window, enlarging 177
- PL/I 231
 - AFTERALL 34
 - AFTERCICS 34
 - AFTERMACRO 34
 - AFTERSQL 34
 - built-in functions 314
 - compiler options to use to automonitor variables in INCLUDE files while in remote debug mode 34
 - compiler options to use when you want to debug INCLUDE files 34
 - condition handling 309
 - constants 313
 - debugging a program in full-screen mode
 - displaying raw storage 236
 - finding storage overwrite errors 238
 - getting a function traceback 236
 - halting on line if condition is true 235
 - modifying value of variable 235
 - setting a breakpoint to halt 234
 - setting breakpoint to halt 238
 - tracing run-time path for code compiled with TEST 237
 - when not all parts compiled with TEST 236

- PL/I (*continued*)
 - debugging OS PL/I programs 316
 - finding listing 316
 - Enterprise, L prefix command only available with 16
 - Enterprise, M prefix command only available with 16
 - Enterprise, restrictions 317
 - expressions 313
 - how Debug Tool locates separate debug file 447
 - how to choose compiler options for 34
 - notes on using 284
 - PLIBASE 36
 - possible prerequisites 35
 - preparing a program for debugging 33
 - QUERY LOCATION 234
 - run-time options 121
 - sample program for debugging 231
 - session variables 310
 - SIBMBASE 36
 - statements 307
 - structures, accessing 311
 - TEST compiler option, what it controls 33
 - when to Dynamic Debug facility with 35
- PL/I for MVS & VM, compiling 36
- PL/I listing, data set 441
- PL/I source, data set 440
- PL/I, definition of xviii
- PLAYBACK commands
 - introduction to 18
 - PLAYBACK BACKWARD
 - using 190
 - PLAYBACK DISABLE
 - using 191
 - PLAYBACK ENABLE
 - using 189
 - PLAYBACK FORWARD
 - using 190
 - PLAYBACK START
 - using 190
 - PLAYBACK STOP
 - using 191
- PLIBASE 36
- PLITEST 134
- plug-ins
 - how to install 545
 - list of available 6
- plug-ins for remote debugger 545, 548
- plugins 6
- point of view, changing
 - description 409
 - for C and C++ 336
 - with COBOL 299
- POPUP command 164
- positioning lines at top of windows 178
- precompiling DB2 programs 79
- preference file 97, 117
- preferences file 442
 - customizing Debug Tool with 279
- Preferences File
 - in DTCN, description of 97
- preferences files, how to create a 165
- prefix area
 - Debug Tool 169
- Prefix area, description of 160
- prefix commands
 - prefix area on session panel 169
 - using in Debug Tool 171
- prepare an assembler program, steps to 75

- preparing
 - a PL/I program for debugging 33
 - C programs for debugging 39
 - C++ programs for debugging 44
 - to replay recorded statements using PLAYBACK START command 190
- prerequisites
 - for COBOL, possible 29
- previous commands, retrieving 174
- problem determination
 - describing problems 563
 - determining business impact 563
 - submitting problems 564
- Profile name pattern 547
- profile settings, changing in Debug Tool 277
- program
 - CICS, choosing debugging mode for 50
 - CICS, debugging 381
 - DB2, debugging 363
 - hook
 - compiling with, PL/I 33
 - removing 393, 395
 - rules for placing in C 43, 48
 - rules for placing in C++ 47
 - IMS, choosing debugging mode for 51
 - loaded from LLA 427
 - multithreading, debugging 415
 - preparation
 - considerations, size and performance 393, 394, 395
 - TEST compiler option, for PL/I 33
 - TEST compiler option, for VS COBOL II 29
 - reducing size 393
 - source, displaying with Debug Tool 160
 - stepping through 188
 - that Debug Tool ignores when explicit debug mode is active 429
 - UNIX System Services, debugging 399
 - variables
 - accessing for C and C++ 320
 - variables, accessing for COBOL 291
- Program IDs, specifying correct for C/C++ and Enterprise PL/I programs 94
- programming language neutral, how to write commands that are 183
- pseudo-conversational program, saving settings 386
- PX constant (PL/I) 287

Q

- qualification
 - description, for C and C++ 335
 - general description 407
- qualifying variables
 - with COBOL 297
- QUERY LOCATION
 - assembler 269
 - COBOL 217
 - LangX COBOL 228
 - PL/I 234

R

- RACF access, combinations of EQAOPTS BROWSE command and 54
- recording
 - breakpoints using SET AUTOMONITOR 186

- recording (*continued*)
 - number of times each source line runs 185
 - restrictions on, statements 191
 - session with the log file 183
 - statements, introduction to 18
 - statements, using PLAYBACK ENABLE command 189
 - stopping, using PLAYBACK DISABLE command 191
- recording a debug session 162
- referencing variables, implications of 49
- reloading programs into an active CICS region 543
- remote debug mode
 - commands not allowed while browse mode is active 53
 - description of 6
 - examples of 121
 - plug-ins for 545, 548
 - where to find list of Debug Tool commands supported by 6
- remote debug mode, PL/I, debugging INCLUDE files 34
- removing statement and symbol tables 394, 395
- replacing variables in Monitor window 199
- replaying
 - statements, introduction to 18
- replaying recorded statements 190
- replaying statements
 - changing direction of 190
 - direction of 190
 - restrictions on 191
 - stopping using PLAYBACK STOP command 191
 - using PLAYBACK commands 189
 - using PLAYBACK START command 190
- requirements
 - for debugging CICS programs 381
- reserved keywords
 - for C 325
 - for COBOL 290
- RESLIB 29
- restoring, manually; of settings, breakpoints, and monitor specifications 194
- restrictions 290
 - accessing COBOL data, for 191
 - arithmetic expressions, for COBOL 295
 - debugging OS PL/I programs 316
 - debugging VS COBOL II programs 300
 - expression evaluation, for COBOL 295
 - location of source on HFS 64, 65, 66
 - modifying variables in Monitor window 206
 - recording and replaying statements, for 191
 - string constants in COBOL 296
 - when debugging multilanguage applications 415
 - when debugging under CICS 386
 - when using a continuation character 290
 - while debugging assembler programs 348
 - while debugging Enterprise PL/I 317
- RETRIEVE command
 - using 174
- retrieving commands
 - with RETRIEVE command 174
- retrieving lines from Log or Source windows 174
- RIGHT, SCROLL command 176
- RLIM processing, CICS 387
- RUN subcommand 364
- run time
 - environment, displaying attributes of 334
 - option, TEST(ERROR, ...), for PL/I 310
 - options module, CEEUOPT 81
- run-time options
 - specifying the STORAGE option 121

- run-time options (*continued*)
 - specifying the TRAP(ON) option 121
 - specifying with COBOL and PL/I 121
- running a program 188
- running in batch mode
 - considerations, TEST run-time option 118
- running your program, introduction to 14
- RUNOPTS (DB2)
 - comparing EQADDCXT to 106
- RUNTO command
 - using, to replay recorded statements 190

S

- save breakpoints file 443
- save monitor specifications file 443
- save settings file 443
- SAVEBPDNSALLOC, EQAOPTS command 444
- SAVEBPDSN, EQAOPTS command 444
- SAVEBPS 443
- SAVESETDSN, EQAOPTS command 443
- SAVESETDSNALLOC, EQAOPTS command 443
- SAVESETS 443
- saving
 - breakpoints 191
 - monitor specifications 191
 - settings 191
 - setup file using Debug Tool Utilities 126
- saving (automatically) settings, breakpoints, and monitor specifications 193
- saving and restoring customizations 279
- saving and restoring settings, how to improve performance in environment with multiple enclaves 195
- saving, disabling automatic of settings, breakpoints, and monitor specifications 194
- scenarios
 - list of C, debugging 39, 41, 46
 - list of C++, debugging 45
 - list of COBOL, debugging 27
 - list of PL/I, debugging 34
- scope of objects in C and C++ 330
- screen control mode, what is 50
- scroll area, Debug Tool 169
- SCROLL command
 - using 175
- search string, syntax of 179
- searching for characters or strings 178
- searching, how Debug Tool searches for 178
- SELECT statement, example of 85
- self-modifying code, restrictions for debugging 357
- separate debug file
 - COBOL and PL/I, how Debug Tool locates the 447
 - separate debug file files, how Debug Tool locates 445
 - separate debug file, attributes to use for 80
 - separate debug file, data set 441
 - separate terminal mode, what is 50
- service, when you apply to Language Environment 110
- session
 - variables, for PL/I 310
- session panel
 - changing colors and highlighting in 276
 - changing physical window layout 274
 - command line 169
 - description 157
 - header fields 158
 - navigating 175
 - order in which Debug Tool accepts commands from 170

- session panel (*continued*)
 - PF keys
 - initial settings 173
 - using 173
 - while debugging LangX COBOL 304
 - windows
 - scrolling 176
- session panel, while debugging assembler 344
- session settings
 - changing in Debug Tool 273
- session variables
 - declaring, for COBOL 295
- SET AUTOMONITOR ON BOTH command, how it works 202
- SET AUTOMONITOR ON command, example 202
- SET AUTOMONITOR ON command, how it works 201
- SET AUTOMONITOR ON PREVIOUS command, how it works 201
- SET commands
 - SET AUTOMONITOR
 - using to record breakpoints 186
 - viewing output from 161
 - SET AUTOMONITOR ON
 - monitoring values of variables 200
 - SET DEFAULT SCROLL
 - using 160
 - SET EQUATE
 - using 273
 - SET INTERCEPT
 - using with C and C++ programs 328
 - SET PFKEY
 - using in Debug Tool 173
 - SET QUALIFY
 - using with COBOL 299
 - using, for C and C++ 336
 - SET REFRESH
 - using 389
 - SET SCROLL DISPLAY OFF
 - using 160
 - SET WARNING
 - using with PL/I 316
- SET DEFAULT LISTINGS command 167
- SET EXPLICITDEBUG 429
- SET QUALIFY
 - with multiple enclaves 417
- SET SOURCE command 166
- set up
 - overall steps to, debugging session 23
- SET WARNING OFF, how to use 187
- setting
 - line breakpoint 186
 - setting breakpoints, in C++ 338
 - setting breakpoints, introduction to 14
- settings
 - changing Debug Tool profile 277
 - changing Debug Tool session 273
- setup file
 - copying JCL into, using DTSU 124
 - creating, using Debug Tool Utilities 123
 - editing, using DTSU 123
 - saving, using Debug Tool Utilities 126
- setup files
 - overview of 7
- SIBMBASE 36
- single terminal mode, what is 50
- size, reducing program 393
- sizing physical windows 275

- skipping programs 429
- Software Support
 - contacting 562
 - describing problems 563
 - determining business impact 563
 - receiving updates 561
 - submitting problems 564
- Source display area, description of 161
- source file in window, changing 166
- source files, how Debug Tool locates 445, 447
- Source window
 - changing source files 166
 - description 160
 - displaying halted location 180
 - retrieving lines from 174
- SOURCE, PL/I compiler option 36
- source, program
 - displaying with Debug Tool 160
- SQLCODE 367
- Sta 92
- STANDARD 347
- starting
 - a debugging session in full-screen mode using the Terminal Interface Manager 139
 - Debug Tool from DB2 stored procedures 153
 - Debug Tool in full-screen mode, introduction to 12
 - Debug Tool Utilities 9
 - your program from Debug Tool Utilities 126
- starting a full-screen debug session 151
- starting Debug Tool
 - __ctest(), using 135
 - batch mode 137
 - DB2 program with TSO 364
 - from a Language Environment program 127
 - under CICS 147, 150
 - under CICS, using CEEUOPT 150
 - under MVS in TSO 141
 - using the TEST run-time option 117
 - with PLITEST 134
 - with the CEETEST function call 127
 - within an enclave 417
- Starting Debug Tool
 - at different points 118
- starting interactive function calls
 - in C 247
- starting your program 151
- statement tables, removing 394, 395
- statements
 - PL/I 307, 310
 - recording and replaying, introduction to 18
- stdout, capturing output to
 - in C 246
 - in C++ 258
- STEP command
 - using, to replay recorded statements 190
- stepping
 - through a program 188
 - through C++ programs 338
- 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 32
- stopping
 - Debug Tool session 19
- storage
 - classes, for C 331
 - displaying, introduction to 17
- storage (*continued*)
 - LangX COBOL, displaying 229
- storage errors, finding
 - overwrite
 - in assembler 271
 - in C 249
 - in C++ 261
 - in COBOL 222
 - in LangX COBOL 230
 - in PL/I 238
 - uninitialized
 - in C 249
 - in C++ 261
- STORAGE run-time option, specifying 121
- storage, raw
 - C, displaying 247
 - C++, displaying 259
 - COBOL, displaying 219
 - PL/I, displaying 236
- stored procedures
 - DB2, debugging 367
- string
 - syntax for searching 179
- string substitution, using 273
- strings
 - C, displaying 247
 - C++, displaying 259
 - searching for in a window 178
- SUB DB2 stored procedures 83
- substitution, using string 273
- SUBSYS
 - with C programs, what to do about 67
- Suffix area, description of 161
- suppressing the display of warning messages 187
- SWAP command compared to scroll commands 176
- SWAP command, when to use 176
- SYM suboption of TEST compiler option (PL/I), effect of 37
- symbol tables, removing 394, 395
- syntax diagrams
 - how to read xviii
- SYSCDBG 441
- SYSDEBUG 441
- system commands, issuing, Debug Tool 171

T

- TCP/IP, specifying for IMS programs (IPv4 or IPv6 formats) 375
- template in C++ 338
- temporary storage queue
 - how Debug Tool uses 88
- temporary storage queue, comparing VSAM with 88
- Term 92
- Terminal Id
 - in DTCN, description of 93
- Terminal Interface Manager
 - example of 138
 - how to start 139
- terminal mode, selecting correct Display ID for each type of 97
- terminology, Debug Tool xvi
- TEST compiler option
 - C, how to choose 41, 46
 - COBOL, how to choose 27
 - debugging C when only a few parts are compiled with 246

- TEST compiler option (*continued*)
 - debugging C++ when only a few parts are compiled with 257
 - debugging COBOL when only a few parts are compiled with 218
 - debugging LangX COBOL when only a few parts are compiled with 229
 - debugging PL/I when only a few parts are compiled with 236
 - for PL/I 33
 - PL/I, how to choose 34
 - specifying NUMBER option with 29
 - using #pragma statement to specify 43
 - versus DEBUG runtime option (COBOL) 29
- TEST compiler option (C), effect of 42
- TEST compiler option (C++), effect of 47
- TEST run-time option
 - as parameter on RUN subcommand 364
 - different ways to specify 54
 - example of 120
 - for CICS programs, how to specify 56
 - for DB2 programs, how to specify 56
 - for DB2 stored procedures, how to specify 57
 - for IMS programs, how to specify 57
 - for JES batch programs, how to specify 56
 - for PL/I 310
 - for TSO programs, how to specify 56
 - for UNIX System Services programs, how to specify 56
 - specifying with #pragma 122
 - suboption processing order 117
- TEST suboptions, redefining at runtime 117
- this pointer, in C++ 257
- TIM
 - use to create TEST runtime options data set 111
- trace file for DTCN Profiles or DTSP Profile 555
- trace, generating a COBOL run-time paragraph 221
- traceback, COBOL routine 220
- traceback, function
 - in assembler 270
 - in C 248
 - in C++ 260
 - in PL/I 236
- traceback, LangX COBOL routine 230
- tracing run-time path
 - in C 248
 - in C++ 260
 - in COBOL 220
 - in PL/I 237
- Tran 92
- Transaction Id
 - in DTCN, description of 93
- TRAP, Language Environment run-time option 210, 409
- TRAP(ON) run-time option, specifying 121
- trigraph 284
- trigraphs
 - using with C 284
- TSO
 - starting Debug Tool using TEST run-time option 151
- TSO command
 - using to debug DB2 program 364
- TSO, starting Debug Tool under 141
- TSQ 88

U

- uninitialized storage errors, finding
 - in C 249

- uninitialized storage errors, finding (*continued*)
 - in C++ 261
- UNIX System Services
 - compiling a C program on 65
 - compiling a C++ program 66
 - compiling a Enterprise PL/I program on 64
 - using Debug Tool with 399
- unsupported
 - HLL modules, coexistence with 414
 - PL/I language elements 316
- UP, SCROLL command 176
- URM debugging 99
- USE file 118
- User Id
 - in DTCN, description of 95

V

- values
 - assigning to C and C++ variables 321
 - assigning to COBOL variables 291
- variable
 - automonitor 16
 - changing value of 17
 - continuous display 16
 - displaying value of 15
 - modifying value
 - in C 245
 - in C++ 256
 - in COBOL 217
 - in PL/I 235
 - one-time and continuous display 16
 - one-time display 15
 - using SET AUTOMONITOR ON command to monitor value of 200
 - value, displaying 196
- variable, displaying data type of 199
- variables
 - accessing program, for C and C++ 320
 - accessing program, for COBOL 291
 - assigning values to, for C and C++ 321
 - assigning values to, for COBOL 291
 - compatible attributes in multiple languages 412
 - displaying, for C and C++ 320
 - displaying, for COBOL 292
 - HLL 406
 - qualifying 407
 - session
 - declaring, for C and C++ 322
 - session, for PL/I 310
- viewing and modifying data members in C++ 257
- VS COBOL II
 - compiler options to use 72
- VS COBOL II programs, additional preparation steps for 29
- VS COBOL II programs, debugging 300
- VS COBOL II, finding list for 300
- VSAM, comparing CICS temporary storage queue with 88
- VTAM
 - starting a debugging session through a, terminal 139

W

- warning, for PL/I 316
- window
 - description of Memory 163
- window id area, Debug Tool 169

- window, error numbers in 209
- windows, Debug Tool physical
 - changing configuration 274
 - opening and closing 275
 - resizing 275
- windows, Debug Tool session panel
 - opening and closing 275
 - zooming 276
- Working-Storage Section, displaying 198

X

- XPLINK
 - restriction on applications that use 359

Z

- ZOOM command, how and where to use 177
- zooming a window, Debug Tool 276



Product Number: 5655-Q10

Printed in USA

SC14-7600-04

