User’s Guide

Version 7.1
This edition applies to Debug Tool for z/OS, Version 7.1 (Program Number 5655-R44) with the PTF for APAR PK34211 applied, 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 (Program Number 5694-A01)
- 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)
- High Level Assembler for MVS & VM & VSE Version 1 Release 4 and Version 1 Release 5 (Program Number 5696-234)
- 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 3.5 or earlier (Program Number 5655-H31)

Parts of this edition apply to Debug Tool Utilities and Advanced Functions for z/OS, Version 7.1 (Program Number 5655-R45).

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

You can order publications online at www.ibm.com/shop/publications/order, or by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. Eastern Standard Time (EST). The phone number is (800)879-2755. The fax number is (800)445-9269.

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

© Copyright International Business Machines Corporation 1992, 2006. All rights reserved.

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

About this document xvi
Who might use this document xi
Accessing z/OS licensed documents on the Internet xi
Using LookAt to look up message explanations xii
How this document is organized xii
Terms used in this document xv
How to read syntax diagrams xvi
Symbols xvi
Syntax items xvii
Syntax examples xvii
How to send your comments xviii

Summary of changes xxi
Changes introduced with PTF for APAR PK34211 xxi
Changes introduced with Debug Tool V7.1 xxii
Changes introduced with Debug Tool Utilities and
Advanced Functions V7.1 xxiii

Part 1. Getting started with Debug
Tool 1

Chapter 1. Debug Tool: overview 3
Debug Tool interfaces 4
Full-screen mode 4
Batch mode 5
Remote debug mode 5

Chapter 2. Debug Tool Utilities and
Advanced Functions: introduction 7
Debug Tool Utilities: creating and managing setup
files 8
Debug Tool Utilities: converting, compiling, and
linking 8
Debug Tool Utilities: preparing assembler programs 9
Debug Tool Utilities: conducting code coverage 9
Debug Tool Utilities: preparing IMS run-time
environment 9
Debug Tool Utilities: Load Module Analyzer 9
Debug Tool Utilities: Fault Analyzer Side File Create 9
Debug Tool Utilities: Fault Analyzer listing create 9
Starting Debug Tool Utilities 10

Chapter 3. Debugging a program in
full-screen mode: introduction 11
Compiling or assembling your program with the
proper compiler options 11
Starting Debug Tool 12
Stepping through a program 14
Running your program to a specific line 14
Setting a breakpoint 14
Displaying the value of a variable 15
Changing the value of a variable 16
Skipping a breakpoint 16

Part 2. Preparing your program for
debugging 19

Chapter 4. Planning your debug
session and collecting resources 21
Do you want your program to have hooks 21
Do you want to reference variables during your
debug session 22
Do you want full debug capability or smaller
application size and higher performance 22
When do you want to start Debug Tool and when
do you want it to gain control 23
Do you want to use Debug Tool in full-screen mode,
in batch mode, or in remote debug mode 24
Collecting your resources 24

Chapter 5. Preparing a COBOL
program 25
Compiling a COBOL program with the TEST
compiler option 25
Converting an OS/VS COBOL program to 1985
COBOL Standard 27

Chapter 6. Preparing an OS/VS COBOL
program 29
Compiling your OS/VS COBOL program 29
Creating the EQALANGX file 29
Link-editing your program 31

Chapter 7. Preparing a PL/I program 33
Compiling a PL/I program with the TEST compiler
option 33
Compiling a Enterprise PL/I program on an HFS
file system 34
Compiling a PL/I for MVS & VM or OS PL/I
program 35

Chapter 8. Preparing a C program 37
Choosing between TEST and DEBUG compiler
options 37
Compiling a C program with the TEST compiler
option 37
Compiling your C program with the
DEBUG(FORMAT(DWARF)) option 39
Compiling a C program on an HFS file system 40
Rules for the placement of hooks in functions and
nested blocks 40
Rules for placement of hooks in statements and
path points 41
Chapter 9. Preparing a C++ program .............. 43
| Choosing between TEST and DEBUG compiler options .................. 43
| Compiling a C++ program with the TEST compiler option ................. 43
| Compiling a C++ program with the DEBUG(FORMAT(DWARF)) option .......... 44
| Compiling a C++ program on an HFS file system .......................... 45
| Rules for the placement of hooks in functions and nested blocks ........ 45
| Rules for the placement of hooks in statements and path points ........... 45

Chapter 10. Preparing an assembler program ....................... 47
| Before you begin ........................................ 47
| Assembling your program .................................. 48
| Creating the EQALANGX file ................................ 48
| Assembling your program and creating EQALANGX ...................... 49
| Link-editing your program .................................. 50

Chapter 11. Preparing a DB2 program ................................. 51
| Processing SQL statements .................................. 51
| Linking DB2 programs for debugging ......................... 52
| Binding DB2 programs for debugging .......................... 53

Chapter 12. Preparing a DB2 stored procedures program ............. 55

Chapter 13. Preparing a CICS program ............................... 57
| Link-editing EQADCCXT into your program ......................... 57
| Creating and storing a DTCN profile ................................ 58
| Sharing DTCN repository profile items among CICS systems ........... 63
| Using CADP to manage debugging profiles ......................... 63
| Starting non-Language Environment Debug Tool under CICS ........... 63
| Passing run time parameters into the non-Language Environment debug session on startup ................................. 64

Chapter 14. Preparing an IMS program ............................... 65
| Preparing an IMS program with Debug Tool Utilities ................. 65
| Compiling and linking your IMS program .......................... 65
| Creating setup file for your IMS program by using Debug Tool Utilities ................................................ 66
| Setting up the run-time options for your IMS Version 8 TM program by using Debug Tool Utilities ......................... 66
| Linking IMS programs for debugging ................................ 67
| Preparing an IMS program without Debug Tool Utilities .............. 67
| Preparing an IMS MPP program that does not run in Language Environment .............................................. 68

Chapter 15. Preparing a program by using the LE exit routine ............ 69
| Editing the source code of CEEBXTA (optional) ..................... 69
| Modifying the naming pattern .................................. 70
| Modifying the message display level ............................. 70
| Linking the exit routine into the program ........................ 71
| Creating the TEST runtime option data set ......................... 71

Part 3. Starting Debug Tool ........................................... 73

Chapter 16. Starting Debug Tool from the Debug Tool Utilities ........ 75
| Creating the setup file ....................................... 75
| Editing an existing setup file .................................. 75
| Copying information into a setup file from an existing JCL .......... 76
| Entering file allocation statements, run-time options, and program parameters ........................................ 76
| Saving your setup file ........................................ 78
| Starting your program ........................................ 78

Chapter 17. Starting Debug Tool by using the TEST run-time option ...... 79
| Special considerations while using the TEST run-time option ........ 79
| Defining TEST suboptions in your program ......................... 79
| Suboptions and NOTEST ....................................... 80
| Implicit breakpoints ........................................... 80
| Primary commands file and USE file ................................ 80
| Running in batch mode ......................................... 80
| Starting Debug Tool at different points .......................... 80
| Session log ....................................................... 81
| Precedence of Language Environment run-time options ............... 81
| Example: TEST run-time options ................................ 82
| Specifying additional run-time options with COBOL II and PL/I programs ................................................ 83
| Specifying the STORAGE run-time option ........................ 83
| Specifying the TRAP(ON) run-time option ......................... 84
| Specifying TEST run-time option with #pragma runopts in C and C++ ..................................................... 84

Chapter 18. Starting Debug Tool from a program ....................... 85
| Starting Debug Tool with CEETEST ............................. 85
| Usage notes .................................................... 87
| Example: using CEETEST to start Debug Tool from C/C++ ......................... 88
| Example: using CEETEST to start Debug Tool from COBOL II and PL/I ......................... 89
| Example: using CEETEST to start Debug Tool from PL/I ......................... 89
| Starting Debug Tool with PLITEST ................................ 92
| Starting Debug Tool with the __ctest() function .................... 93

Chapter 19. Starting Debug Tool for batch or TSO programs ............ 95
| Programs that start in Language Environment ....................... 95
Halting before calling an invalid program in COBOL 164

Chapter 25. Debugging an OS/VS COBOL program in full-screen mode  167
Example: sample OS/VS COBOL program for debugging  167
Defining a compilation unit as OS/VS COBOL and loading debug information  169
Defining a compilation unit in a different load module as OS/VS COBOL  170
Halting when certain OS/VS COBOL programs are called  170
Displaying and modifying the value of OS/VS COBOL variables or storage  170
Halting on a line in OS/VS COBOL only if a condition is true  171
Debugging OS/VS COBOL when debug information is only available for a few parts  171
Getting an OS/VS COBOL program traceback  171
Finding unexpected storage overwrite errors in OS/VS COBOL  172

Chapter 26. Debugging a PL/I program in full-screen mode  173
Example: sample PL/I program for debugging  173
Halting when certain PL/I functions are called  176
Modifying the value of a PL/I variable  176
Halting on a PL/I line only if a condition is true  177
Debugging PL/I when only a few parts are compiled with TEST  177
Displaying raw storage in PL/I  178
Getting a PL/I function traceback  178
Tracing the run-time path for PL/I code compiled with TEST  179
Finding unexpected storage overwrite errors in PL/I  180
Halting before calling an undefined program in PL/I  180

Chapter 27. Debugging a C program in full-screen mode  181
Example: sample C program for debugging  181
Halting when certain functions are called in C  184
Modifying the value of a C variable  184
Halting on a line in C only if a condition is true  185
Debugging C when only a few parts are compiled with TEST  186
Capturing C output to stdout  186
Capturing C input to stdin  186
Calling a C function from Debug Tool  187
Displaying raw storage in C  187
Debugging a C DLL  187
Getting a function traceback in C  188
Tracing the run-time path for C code compiled with TEST  188
Finding unexpected storage overwrite errors in C  189
Finding uninitialized storage errors in C  189
Halting before calling a NULL C function  190

Chapter 28. Debugging a C++ program in full-screen mode  191
Example: sample C++ program for debugging  191
Halting when certain functions are called in C++  195
Modifying the value of a C++ variable  195
Halting on a line in C++ only if a condition is true  196
Viewing and modifying data members of the this pointer in C++  197
Debugging C++ when only a few parts are compiled with TEST  197
Capturing C++ output to stdout  198
Capturing C++ input to stdin  198
Calling a C++ function from Debug Tool  198
Displaying raw storage in C++  199
Debugging a C++ DLL  199
Getting a function traceback in C++  199
Tracing the run-time path for C++ code compiled with TEST  200
Finding unexpected storage overwrite errors in C++  200
Finding uninitialized storage errors in C++  201
Halting before calling a NULL C++ function  202

Chapter 29. Debugging an assembler program in full-screen mode  203
Example: sample assembler program for debugging  203
Defining a compilation unit as assembler and loading debug data  206
Deferred LDDs  207
Re-appearance of an assembler CU  207
Multiple compilation units in a single assembly  207
Loading debug data from multiple CSECTs in a single assembly using one LDD command  208
Loading debug data from multiple CSECTs in a single assembly using separate LDD commands  208
Debugging multiple CSECTs in a single assembly after the debug data is loaded  208
Halting when certain assembler routines are called  209
Displaying and modifying the value of assembler variables or storage  209
Converting a hexadecimal address to a symbolic address  210
Halting on a line in assembler only if a condition is true  210
Getting an assembler routine traceback  210
Finding unexpected storage overwrite errors in assembler  211

Chapter 30. Customizing your full-screen session  213
Defining PF keys  213
Defining a symbol for commands or other strings  214
Customizing the layout of windows on the session panel  214
Opening and closing session panel windows  215
Resizing session panel windows  215
Zooming a window to occupy the whole screen  216
Customizing session panel colors  216
Customizing profile settings  217
Saving customized settings in a preferences file  219
# Debugging

## Chapter 31. Entering Debug Tool commands

- **Using uppercase, lowercase, and DBCS in Debug Tool commands**
- **DBCS**
- **Character case and DBCS in C and C++**
- **Character case in COBOL and PL/I**
- **Abbreviating Debug Tool keywords**
- **Entering multiline commands in full-screen and line mode**
- **Entering multiline commands in a command file**
- **Entering multiline commands without continuation**
- **Using blanks in Debug Tool commands**
- **Using comments in Debug Tool commands**
- **Using constants in Debug Tool commands**
- **Getting online help for Debug Tool command syntax**

## Chapter 32. Debugging COBOL programs

- **Debug Tool commands that resemble COBOL statements**
- **COBOL command format**
- **COBOL compiler options in effect for Debug Tool commands**
- **COBOL reserved keywords**
- **Using COBOL variables with Debug Tool**
- **Accessing COBOL variables**
- **Assigning values to COBOL variables**
- **Example: assigning values to COBOL variables**
- **Displaying values of COBOL variables**
- **Using DBCS characters in COBOL**
- **%PATHCODE values for COBOL**
- **Declaring session variables in COBOL**
- **Debug Tool evaluation of COBOL expressions**
- **Displaying the results of COBOL expression evaluation**
- **Using constants in COBOL expressions**
- **Using Debug Tool functions with COBOL**
- **Using %HEX with COBOL**
- **Using the %STORAGE function with COBOL**
- **Qualifying variables and changing the point of view in COBOL**
- **Qualifying variables in COBOL**
- **Changing the point of view in COBOL**
- **Considerations when debugging a COBOL class program**
- **Debugging VS COBOL II programs**
- **Finding the listing of a VS COBOL II program**

## Chapter 33. Debugging an OS/VS COBOL program

- **Loading an OS/VS COBOL program’s debug information**
- **Debug Tool session panel while debugging an OS/VS COBOL program**
- **Restrictions for debugging an OS/VS COBOL program**
- **%PATHCODE values for OS/VS COBOL programs**
- **Restrictions for debugging non-Language Environment programs**

## Chapter 34. Debugging PL/I programs

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

## Chapter 35. Debugging C and C++ programs

- **Debug Tool commands that resemble C and C++ commands**
- **Using C and C++ variables with Debug Tool**
- **Accessing C and C++ program variables**
- **Displaying values of C and C++ variables or expressions**
- **Assigning values to C and C++ variables**
- **%PATHCODE values for C and C++**
- **Declaring session variables with C and C++**
- **C and C++ expressions**
- **Calling C and C++ functions from Debug Tool**
- **C reserved keywords**
- **C operators and operands**
- **Language Environment conditions and their C and C++ equivalents**
- **Debug Tool evaluation of C and C++ expressions**
- **Intercepting files when debugging C and C++ programs**
- **Scope of objects in C and C++**
- **Storage classes in C and C++**
- **Blocks and block identifiers for C**
- **Blocks and block identifiers for C++**
- **Example: referencing variables and setting breakpoints in C and C++ blocks**
- **Scope and visibility of objects**
- **Blocks and block identifiers**
- **Displaying environmental information**
Qualifying variables and changing the point of view in C and C++ .......................... 272
  Qualifying variables in C and C++ ............................................. 273
  Changing the point of view in C and C++ ................................. 273
  Example: using qualification in C ......................................... 274
Stepping through C++ programs ................................................. 275
Setting breakpoints in C++ ...................................................... 275
  Setting breakpoints in C++ using AT ENTRY(EXIT) ......................... 275
  Setting breakpoints in C++ using AT CALL ................................ 276
Examining C++ objects ............................................................ 277
  Example: displaying attributes of C++ objects ........................... 277
Monitoring storage in C++ ........................................................ 278
  Example: monitoring and modifying registers and storage in C .......... 278

Chapter 36. Debugging an assembler program ............................................. 281
  The SET ASSEMBLER and SET DISASSEMBLY commands .............. 281
  Loading an assembler program’s debug information .................... 281
  Debug Tool session panel while debugging an assembler program .... 282
  %PATHCODE values for assembler programs .............................. 283
  Using the STANDARD and NOMACGEN view ............................... 285
Restrictions for debugging an assembler program ............................. 285
  Restrictions for debugging a Language Environment assembler MAIN program .................................................. 286
  Restrictions on setting breakpoints in the prologue of Language Environment assembler programs .................... 286
  Restrictions for debugging non-Language Environment programs ....... 287
  Restrictions for debugging code that uses instructions as data ....... 287
  Restrictions for debugging self-modifying code ......................... 288

Chapter 37. Debugging a disassembled program ............................................ 291
  The SET ASSEMBLER and SET DISASSEMBLY commands ................ 291
  Capabilities of the disassembly view ....................................... 291
  Starting the disassembly view ................................................. 292
  The disassembly view .......................................................... 292
  Performing single-step operations ......................................... 293
  Setting breakpoints .......................................................... 293
  Restrictions for debugging self-modifying code ......................... 293
  Displaying and modifying registers ....................................... 294
  Displaying and modifying storage ......................................... 294
  Changing the program displayed in the disassembly view ............... 294
  Restrictions for the disassembly view ..................................... 294

Part 6. Debugging in different environments .............................................. 297

Chapter 38. Debugging DB2 programs ....................................................... 299
  Debugging DB2 programs in batch mode .................................... 299

  Debugging DB2 programs in full-screen mode .............................. 300

Chapter 39. Debugging DB2 stored procedures ............................................. 303
  Resolving some common problems ........................................... 303

Chapter 40. Debugging IMS programs ...................................................... 305
  Debugging IMS programs interactively ..................................... 306
  Debugging IMS programs in batch mode ................................... 306
  Debugging non-Language Environment IMS MPP programs ............... 307
  Verifying configuration and starting a region ........................... 307
  Choosing an interface and gathering information ....................... 307
  Running the EQASE transaction ............................................. 308

Chapter 41. Debugging CICS programs ..................................................... 311
  Debug modes under CICS ....................................................... 311
  Preventing Debug Tool from stopping at EXEC CICS RETURN .......... 312
  Saving settings while debugging a pseudo-conversational program ... 312
  Saving and restoring breakpoints and monitor specifications .......... 312
  Restrictions when debugging under CICS ................................ 313
  Accessing CICS resources during a debugging session ................ 313

Chapter 42. Debugging ISPF applications ................................................. 315

Chapter 43. Debugging programs in a production environment ........................ 317
  Fine-tuning your programs with Debug Tool .............................. 317
    Removing hooks .............................................................. 317
    Removing statement and symbol tables ................................ 318
  Debugging without hooks, statement tables, and symbol tables .... 318
  Debugging optimized COBOL programs ................................... 319

Chapter 44. Debugging UNIX System Services programs .................................. 321
  Debugging MVS POSIX programs ............................................ 321

Chapter 45. Debugging non-Language Environment programs .......................... 323
  Debugging exclusively non-Language Environment programs .......... 323
  Debugging MVS batch or TSO non-Language Environment initial programs ...... 323
  Debugging CICS non-Language Environment assembler or OS/VS COBOL initial programs ......................... 323

Part 7. Debugging complex applications .................................................. 325
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.

The Debug Tool Utilities and Advanced Functions Coverage Utility is referred to throughout this document as the Debug Tool Coverage Utility or Coverage Utility.

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.

The following operating systems and subsystems are supported:

- z/OS
  - CICS®
  - DB2®
  - IMS™
  - JES batch
  - TSO
  - UNIX® System Services in remote debug mode or full-screen mode through a VTAM terminal only
  - WebSphere® in remote debug mode or full-screen mode through a VTAM terminal 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:


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, (GI10-0671), that includes this key code.

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


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.

---

**Using LookAt to look up message explanations**

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

You can use LookAt from the following locations to find IBM message explanations for z/OS elements and features, z/VM®, VSE/ESA®, and AIX® and Linux®:

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

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

---

**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 introduces Debug Tool Utilities and Advanced Functions and describes some of its features.
  - Chapter 3 provides instructions on how to use full-screen mode to debug a program.
- Part 2 groups together information about how to prepare programs for debugging. The following list describes each chapter:
Chapter 4 describes some of the preliminary planning and resource actions to take when you prepare your programs.

Chapter 5 describes how to prepare a COBOL program.

Chapter 6 describes how to prepare an OS/VS COBOL program.

Chapter 7 describes how to prepare a PL/I program.

Chapter 8 describes how to prepare a C program.

Chapter 9 describes how to prepare a C++ program.

Chapter 10 describes how to prepare an assembler program.

Chapter 11 describes how to prepare a DB2 program.

Chapter 12 describes how to prepare a DB2 stored procedures program.

Chapter 13 describes how to prepare a CICS program.

Chapter 14 describes how to prepare an IMS program.

Chapter 15 describes how to include a call to the TEST run-time 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 16 describes how to start Debug Tool from Debug Tool Utilities.

Chapter 17 describes how to start Debug Tool Utilities by using the TEST run-time option.

Chapter 18 describes how to start Debug Tool from a program.

Chapter 19 describes how to start Debug Tool for your batch or TSO programs.

Chapter 20 describes how to start Debug Tool from CICS programs.

Chapter 21 describes how to start Debug Tool in full-screen mode.

Chapter 22 describes how to start Debug Tool in full-screen mode through a VTAM® terminal. 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 23 provides overview information about full-screen mode.

Chapter 24 provides a sample COBOL program to describe how to debug it in full-screen mode.

Chapter 25 provides a sample OS/VS COBOL program to describe how to debug it in full-screen mode.

Chapter 26 provides a sample PL/I program to describe how to debug it in full-screen mode.

Chapter 27 provides a sample C program to describe how to debug it in full-screen mode.

Chapter 28 provides a sample C++ program to describe how to debug it in full-screen mode.

Chapter 29 provides a sample assembler program to describe how to debug it in full-screen mode.

Chapter 30 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 31 provides information about entering mixed case commands, using DBCS characters, abbreviating commands, entering multiline commands, and entering comments.

Chapter 32 describes how to use Debug Tool commands to debug COBOL programs.

Chapter 33 describes how to use Debug Tool commands to debug OS/VS COBOL programs.

Chapter 34 describes how to use Debug Tool commands to debug PL/I programs.

Chapter 35 describes how to use Debug Tool commands to debug C or C++ programs.

Chapter 36 describes how to use Debug Tool commands to debug assembler programs.

Chapter 37 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 38 describes how to debug a DB2 program.

Chapter 39 describes how to debug a DB2 stored procedure.

Chapter 40 describes how to debug an IMS program.

Chapter 41 describes how to debug a CICS program.

Chapter 42 describes how to debug an ISPF program.

Chapter 43 describes how to debug a production-level program.

Chapter 44 describes how to debug a program running in the UNIX System Services shell.

Chapter 45 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 46 describes how to debug a program written in multiple languages.

Chapter 47 describes the restrictions when you debug a multithreaded program.

Chapter 48 describes how to debug a program that runs across multiple processes and enclaves.

Chapter 49 describes how to debug a multiple-enclave interlanguage communication (ILC) application.

Chapter 50 describes how to instruct Debug Tool to handle problems created by program names or the large size of programs.

Part 8 groups together appendices. 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 provides information about debugging a program in batch mode.
- Appendix E provides information about debugging a program in remote debug mode.
- Appendix F describes the syntax of the TEST compiler option.
- Appendix G describes how to use Load Module Analyzer, a stand-alone program that is part of Debug Tool Utilities and Advanced Functions.
- Appendix H describes the features and tools available to people with physical disabilities that help them use Debug Tool and Debug Tool documents.

The last several chapters 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 has been established. The table below lists these terms and their equivalency in each language.

<table>
<thead>
<tr>
<th>Debug Tool term</th>
<th>C and C++ equivalent</th>
<th>COBOL equivalent</th>
<th>PL/I equivalent</th>
<th>assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile unit</td>
<td>C and C++ source file</td>
<td>Program or class</td>
<td>Program file</td>
<td>CSECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PL/I source</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>file for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Enterprise PL/I</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A package</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>statement or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the name of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the main</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>procedure for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Enterprise PL/I</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Function or compound</td>
<td>Program, nested</td>
<td>Block</td>
<td>CSECT</td>
</tr>
<tr>
<td></td>
<td>statement</td>
<td>program, method</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or PERFORM group</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td>Label</td>
<td>Paragraph name</td>
<td>Label</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or section name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. The PL/I program must be compiled with the Enterprise PL/I for z/OS, Version 3.5, compiler with the PTFs for APARs PK35230 and PK35489 applied and must be running in Language Environment Version 1.4 through 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, the 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 and dialects supported by Debug Tool except OS/VS 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.

**full-screen mode through a VTAM terminal**

Refers to the debugging mode that requires a second terminal, a VTAM terminal, be started and used to debug an application. After the VTAM terminal has been started, you can optionally use the Debug Tool Terminal Interface Manager to identify that terminal to Debug Tool by using a user ID instead of a LU name.

**OS/VS COBOL**

Refers to COBOL programs compiled using the IBM OS/VS COBOL compiler.

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. Please read the information regarding non-Language Environment programs for instructions on how to start Debug Tool and debug OS/VS COBOL programs, unless OS/VS COBOL-specific information 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.

**separate debug file**

Refers to the Enterprise COBOL for z/OS and OS/390 side file and the Enterprise PL/I for z/OS Version 3 Release 5 separate debug file.

---

### 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>➡️</td>
<td>Indicates the beginning of the syntax diagram.</td>
</tr>
<tr>
<td>➡️</td>
<td>Indicates that the syntax diagram is continued to the next line.</td>
</tr>
<tr>
<td>➡️</td>
<td>Indicates that the syntax is continued from the previous line.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Item type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>Required items are displayed on the main path of the horizontal line.</td>
</tr>
<tr>
<td>Optional</td>
<td>Optional items are displayed below the main path of the horizontal line.</td>
</tr>
<tr>
<td>Default</td>
<td>Default items are displayed above the main path of the horizontal line.</td>
</tr>
</tbody>
</table>

Syntax examples

The following table provides syntax examples.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required item.</td>
<td>➞KEYWORD—required_item—➞</td>
</tr>
<tr>
<td>Required items appear on the main path of the horizontal line. You must specify these items.</td>
<td>➞KEYWORD—required_choice1—required_choice2—➞</td>
</tr>
<tr>
<td>Required choice.</td>
<td>➞KEYWORD—optional_item—➞</td>
</tr>
<tr>
<td>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.</td>
<td>➞KEYWORD—optional_choice1—optional_choice2—➞</td>
</tr>
</tbody>
</table>
Table 1. Syntax examples (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default.</td>
<td>![Syntax diagram](default-choice1 optional-choice2 optional-choice3)</td>
</tr>
<tr>
<td>Variable.</td>
<td><img src="variable" alt="Syntax diagram" /></td>
</tr>
<tr>
<td>Repeatable item.</td>
<td><img src="repeatable_item" alt="Syntax diagram" /></td>
</tr>
<tr>
<td>Fragment.</td>
<td>![Syntax diagram](fragment: required-choice1 default-choice optional-choice)</td>
</tr>
</tbody>
</table>

### 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/](http://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.
- Fill out the Readers’ Comment Form at the back of this document, and return it by mail or give it to an IBM representative. If the form has been removed, address your comments to:
  
  IBM Corporation  
  H150/090  
  555 Bailey Avenue  
  San Jose, CA 95141-1003  
  USA  
- Fax your comments to this U.S. number: (800)426-7773.
When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.
Summary of changes

This section lists the key changes made to Debug Tool for z/OS and Debug Tool Utilities and Advanced Functions for z/OS that affect this document.

Changes introduced with PTF for APAR PK34211

The following changes were introduced with the PTF for APAR PK34211:

- Debug Tool can now debug C and C++ programs compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option. For a description of how to compile C and C++ programs with this option, see "Compiling your C program with the DEBUG(FORMAT(DWARF)) option" on page 39 or "Compiling a C++ program with the DEBUG(FORMAT(DWARF)) option" on page 44. For a description of the DEBUG compiler option, see "Syntax for the C and C++ DEBUG compiler option" on page 378.

- You can now specify a network identifier when you specify the VTAM logical unit (LU) identifier for a terminal used in full-screen mode through a VTAM terminal. See Debug Tool Customization Guide for a description of how to install this support. See the section on the TEST run time option in Debug Tool Reference and Messages for a description of the new suboption.

- If your PL/I program was compiled with the following compiler and it is running in the following environment, you can specify the package name or the name of the main procedure instead of a fully qualified data set name.
  - Enterprise PL/I for z/OS, Version 3.5, with the PTFs for APARs PK35230 and PK35489 applied
  - Language Environment Version 1.4 through 1.8 with the PTF for APAR PK33738 applied

- Typing over the values of variables to modify them has changed. Previously, you would do the following steps:
  1. Type in a new value for a variable in the Monitor value area.
  2. Press Enter. Debug Tool would display the corresponding command in the command line.
  3. You look at the command to verify it was correct.
  4. Press Enter to run the command. Debug Tool would change the value.

Now, you do the following steps:
  1. Type in a new value for a variable in the Monitor value area.
  2. Press Enter. Debug Tool changes the value and the command that it used to change the value is stored in the log file.

See "Modifying variables or storage by typing over an existing value" on page 148 for information about the new behavior.

- The size and display of windows are saved. By using the SET SAVE command, you can save the size of each window displayed in full screen mode, as well as which window is displayed in full screen mode. Then, you can use the SET RESTORE command to reinstate those settings. Changes were made to the SET RESTORE, SET SAVE, and QUERY commands to enable this feature.

- Miscellaneous updates.
Changes introduced with Debug Tool V7.1

Minor changes were made to the second edition that support changes introduced with Debug Tool Version 7.1.

- The following enhancements have been made to the monitoring functions:
  - For COBOL programs, Debug Tool does not prefix the program name to the output, allowing more data to be displayed on the same line.
  - You can now display the value of variables, including members of an array or structure, in a columnar format. Debug Tool provides a new command, SET MONITOR COLUMN, which you can use to indicate that you want the Monitor window to display information in columnar format.
  - Debug Tool now displays a ruler, which indicates the offset from the start of the display to current cursor position.
  - You can now update large variables directly in the Monitor window.
  - You can now use the HEX prefix command on only one member of an array or a sublevel of a structure. Previously, you could use the HEX prefix command only on the entire array or structure.
  - You can now update an array or a structure member without making the full name of the array or structure visible. Previously, you could update an array or structure member only if the full name of the array or structure was visible.
- You can now access source code (C, C++, Enterprise PL/I if a separate debug file is not used) stored in library systems that require data sets to be allocated as a DSORG DA or VSAM data set with the SUBSYS=ssss allocation parameter, where ssss is a subsystem provided by the library system vendor.
- Debug Tool provides a method (by using EQAOPTS) that instructs Debug Tool to use the SUBSYS=ssss allocation parameter when it allocates the data set.
- You can now indicate that you want COPE facilities to continue operating while Debug Tool is active. Standardware Corporation's COPE product is used in an IMS environment to deliver some additional capabilities for applications and systems administrators. You use a new option (through EQAOPTS) to enable this behavior.
- You can now indicate what Debug Tool should do if the terminal using the full-screen mode through a VTAM terminal facility or the remote debugger is not available. Use a new option (through EQAOPTS) to select the new behavior.
- Debug Tool has enhanced the LIST, CLEAR, ENABLE, and DISABLE commands to support suspended breakpoints.
- The LIST STORAGE and STORAGE commands have been enhanced so that you can provide a starting byte offset. Previously, Debug Tool used the start of the area of storage allocated to the variable as the starting byte.
- You can now set COBOL level 88 condition variables to TRUE.
- The Debug Tool CICS control utility (DTCN) has been enhanced so that you can temporarily inactivate a profile, then reactivate it at a later time.
- DTCN has been enhanced so you can select CICS tasks to debug based on the client IP name or address.
- A new option TASK has been added to the QUIT DEBUG command to help you terminate debugging sessions that involve pseudo-conversational applications. If you specify the TASK option, Debug Tool terminates immediately. It does not wait until the end of the current CICS pseudo-conversational task, which can be indicated by, for example, an EXEC CICS RETURN TRANSID. When a new task is started in the pseudo-conversation, Debug Tool resumes debugging.
Changes introduced with Debug Tool Utilities and Advanced Functions
V7.1

• The following enhancements have been made to the monitoring functions:
  – You can use a new command, SET MONITOR WRAP, to indicate how you want to
display the value of a variable, which is being monitored or automonitored,
in the Monitor window. Debug Tool can display the value of a variable in
either a wrapped format or on a scrollable line. In a wrapped format, if the
value exceeds the width of the display, Debug Tool continues the value on the
next line. On a scrollable line, if the value exceeds the width of the display,
you can scroll left or right to see the rest of the value.
  – You can use a new command, SET MONITOR DATATYPE, to indicate whether you
want to display the data type of a variable that is being monitored or
automonitored.

• You can display the source for a compile unit or CSECT (CU) before the load
module containing the CU has been loaded or run. You can work with
breakpoints (for example, examine existing breakpoints or set new breakpoints)
as you would for a CU that has been loaded or run. Debug Tool applies these
breakpoints when the CU becomes active. This feature is also available on
WebSphere Developer for zSeries® and WebSphere Developer Debugger for
zSeries.

• In Debug Tool Utilities, the layout of the panels and arrangement of parameters
in the Create Private Message Regions function of the Manage IMS Programs
section has been improved.

• You can specify which interface (Main Frame Interface (MFI), WebSphere
Developer for zSeries, or WebSphere Developer Debugger for zSeries) to start
when you want to debug a DB2 Stored Procedure, IMS Transaction Manager
(TM), or batch program. Through an user exit, you can specify a TEST run-time
option string that indicates which interface you want to start.

• You can use a utility, EQALANGP, to create a readable listing from a Fault
Analyzer side file (IDLANGX or EQALANGX) or a SYSDEBUG file, which is
generated by using the COBOL TEST(,SEPARATE) compiler option. If you do not
keep compiler listings in order to conserve DASD space, EQALANGP can help
you create a compiler listing that resembles the original compiler listing.

Note: EQALANGP, which is shipped as a component of Debug Tool Utilities
and Advanced Functions, is functionally equivalent to the IDILANGP
program shipped as a component of Fault Analyzer for z/OS.
- The Debug Tool Coverage Utility (DTCU) SVC installer has been updated to help you ensure that the SVC numbers that you choose to use for the DTCU breakpoint SVCs do not conflict with the SVC numbers chosen for another program.

- The Coverage Utility Annotated Listing report is updated for COBOL programs so that you can add an HTML version of the report. The HTML version contains colored lines that indicate statements that were not executed and recomputed statistics based on the annotations in the listing instead of the raw coverage data. In addition, a new HTML Targeted Coverage Report, which contains an Annotated Listing with lines that were changed between two versions of source files, is available for COBOL programs.

- You can use a new command, SET LIST TABULAR, to indicate how you want the output of the LIST command displayed. This helps you format the display so it matches the display of the MONITOR LIST command.

- You can use a new command, DESCRIBE LOADMODS, to indicate that you want to display information about all load modules or a specific load module, which are known to Debug Tool. Debug Tool displays information about where the load module or load modules are loaded from, and the size, the name, and the programs and CSECTs that are contained in a load module if information on a specific load module is requested.

- Enhancements have been added to better integrate Debug Tool Utilities and Advanced Functions with WebSphere Developer for zSeries and WebSphere Developer Debugger for zSeries.
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 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 that lets you 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

<table>
<thead>
<tr>
<th>Compiler or assembler</th>
<th>Batch mode</th>
<th>Full-screen mode</th>
<th>Remote debug mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS COBOL II Version 1 Release 3 and Version 1 Release 4 (with limitations)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AD/Cycle® COBOL/370™ Version 1 Release 1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS/VS COBOL, Version 1 Release 2.4 (with limitations)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL for MVS™ &amp; VM</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COBOL for OS/390 &amp; VM</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Enterprise COBOL for z/OS and OS/390</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PL/I for MVS &amp; VM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enterprise PL/I</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AD/Cycle C/370™ Version 1 Release 2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ for MVS/ESA™ Version 3 Release 2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ feature of OS/390 Version 1 Release 3 and earlier</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++ feature of OS/390 Version 2 Release 10 and later</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>C/C++ feature of z/OS</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM High Level Assembler (HLASM), Version 1 Release 4, and Version 1 Release 5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 on page 4 maps out the Debug Tool interfaces and subsystems each interface supports.
Table 3. Debug Tool interface type by subsystem

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Batch mode</th>
<th>Full-screen mode</th>
<th>Remote mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JES batch</td>
<td>X</td>
<td>X*</td>
<td>X</td>
</tr>
<tr>
<td>UNIX System Services</td>
<td>X*</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CICS</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DB2</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DB2 stored procedures</td>
<td>X*</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IMS (TM and DB) with BTS TSO</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IMS (TM and DB) with BTS batch</td>
<td>X</td>
<td>X*</td>
<td></td>
</tr>
<tr>
<td>IMS without BTS IMS DB batch</td>
<td>X</td>
<td>X*</td>
<td></td>
</tr>
<tr>
<td>IMS without BTS IMS TM</td>
<td>X*</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Support is for full-screen mode through a VTAM terminal only.

Refer to the following sections for more information related to the material discussed in this section.

- **Related concepts**
  - “Debug Tool interfaces”
- **Related tasks**
  - Chapter 4, “Planning your debug session and collecting resources,” on page 21
  - Chapter 23, “Using full-screen mode: overview,” on page 117
- **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.

#### 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.

You can debug non-TSO and non-CICS programs in full-screen mode by using the full-screen mode through a VTAM terminal facility. For example, you can debug a COBOL batch job running in MVS/JES, a DB2 Stored Procedure, an IMS transaction running on an IMS MPP region, or an application running in UNIX System Services. Contact your system administrator to determine how to access a terminal capable of using the full-screen mode through a VTAM terminal facility on your system.
You can debug CICS programs in several different modes, which are described in “Debug modes under CICS” on page 311.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

[Debug Tool Customization Guide](#)

**Batch mode**

You can use Debug Tool command files to predefine series of Debug Tool commands to be performed on a running batch application. Neither terminal input nor user interaction is available for batch debugging of a batch application. The results of the debugging session are saved to a log, which you can review at a later time.

**Remote debug mode**

In remote debug mode, the host application starts Debug Tool, which uses a TCP/IP connection to communicate with a remote debugger on your Windows workstation.

Debug Tool, in conjunction with a remote debugger, provides users with the ability to debug host programs, including batch programs, through a graphical user interface (GUI) on the workstation. The following remote debuggers are available:

- WebSphere Developer Debugger for zSeries
- Compiled Language Debugger component of WebSphere Developer for zSeries
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- IBM Distributed Debugger
  
  This remote debugger is available through the following products:
  - OS/390 C/C++ Productivity Tools
  - VisualAge COBOL for Windows
  - VisualAge PL/I for Windows

The remote debuggers available with the WebSphere products are the recommended choice because they offer more functionality than the IBM Distributed Debugger. For more information about the commands that Debug Tool supports with each remote debugger, see Appendix E, “Notes on debugging in remote debug mode,” on page 371. For more information about the software requirements for each remote debugger, see the Program Directory for Debug Tool for z/OS.

See each product’s documentation for a list of its prerequisites and capabilities.
Chapter 2. Debug Tool Utilities and Advanced Functions: introduction

Debug Tool Utilities and Advanced Functions enhances Debug Tool, and the combined strength of these products can help you debug and examine your programs.

Debug Tool provides a tool called Debug Tool Setup Utility (DTSU) to help you create and manage setup files which can help you run your programs. Debug Tool Utilities and Advanced Functions adds tools to help you do the following tasks:

- Prepare your high-level language programs for debugging by helping you convert, compile, and link.
- Prepare your assembler programs for debugging by helping you assemble, create debug information, and link.
- Conduct analysis on your test cases to determine how thoroughly your test cases test your programs (also called code coverage).
- For IMS Version 8, browse and edit the Language Environment run-time parameters table.
- Create a batch job for private IMS message region with customized load libraries and region attributes.
- Analyze load modules and program objects to identify the language translator used to generate the object for each CSECT.
- Edit a TEST runtime option data set that the Debug Tool Language Environment user exit uses to start a debug session.
- Invoke IBM File Manager for z/OS.
- Convert old COBOL source code and copybooks to new versions of COBOL by using COBOL and CICS Command Level Conversion Aid (CCCA).

The combination of DTSU and these tools is called Debug Tool Utilities.

Debug Tool provides a rich set of commands to debug your programs. Debug Tool Utilities and Advanced Functions enhances this set of commands by adding the following commands:

- ALLOCATE
- CALL %CEBR
- CALL %CECI
- CALL %FA
- CALL %HOGAN
- CLEAR LOAD
- DESCRIBE ALLOCATIONS
- DESCRIBE LOADMODS
- FREE
- JUMPTO
- LOAD
- LOADDEBUGDATA or LDD
- PLAYBACK BACKWARD
- PLAYBACK DISABLE
- PLAYBACK ENABLE
- PLAYBACK FORWARD
- PLAYBACK START
- PLAYBACK STOP
- QUERY ASSEMBLER
- QUERY AUTOMONITOR
- QUERY CURRENT VIEW
- QUERY DEFAULT VIEW
- QUERY PLAYBACK
- QUERY PLAYBACK LOCATION
- SET ASSEMBLER
- SET AUTOMONITOR
- SET DEFAULT VIEW
- SET LIST TABULAR
- SET MONITOR DATATYPE
- SET MONITOR WRAP
- SET PROGRAMMING LANGUAGE ASSEMBLER
- SET PROGRAMMING LANGUAGE OSVSCOBOL
- SET QUALIFY CU
- SET QUALIFY LOAD
- WHEN option of AT CHANGE
- FS, WSS, LS, LOS option of LIST TITLED

**Debug Tool Utilities: creating and managing setup files**

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 run your program and start Debug Tool. 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 files, use Debug Tool Setup Utility (DTSU), which is part of Debug Tool. You do not need Debug Tool Utilities and Advanced Functions to use this tool.

**Debug Tool Utilities: converting, compiling, and linking**

Debug Tool Utilities helps you do the following tasks:

- Run the DB2 precompiler or the CICS translator.
  
  If the program source is a sequential data set and the DB2 precompiler is selected, make sure the DBRMLIB data set field in EQAPPCnB panel, where \( n = 1,2, \ldots, 5 \), is a partitioned data set with a member name. For example, `DEBUG.TEST.DBRMLIB(PROG1)`.

- 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.

1. See [Debug Tool Reference and Messages](#) for description of the specific features of this command that require Debug Tool Utilities and Advanced Functions.
- Generate IDILANGX or EQALANGX side files.
- Generate a listing from an IDILANGX, EQALANGX or COBOL SYSDEBUG side file.
- Prepare the following COBOL programs for debugging:
  - Programs written for OS/VS 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 OS/VS 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 OS/VS COBOL source.

### Debug Tool Utilities: preparing assembler programs

By using High-Level Assembler (HLASM), Debug Tool Utilities can help you assemble and create debug information for your assembler programs. After your assembler program is assembled and the debug information is created, you can start Debug Tool and begin debugging your program.

### Debug Tool Utilities: conducting 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.

### Debug Tool Utilities: preparing IMS run-time environment

You can create private IMS message regions that you can use to debug test applications and, therefore, not interfere with other regions. For IMS Version 8, you can modify the Language Environment run-time parameters table without relinking the applications.

### 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.

### Debug Tool Utilities: Fault Analyzer Side File Create

Use this option to create Fault Analyzer IDILANGX or Debug Tool EQALANGX side files.

### Debug Tool Utilities: Fault Analyzer listing create

Use this option to create a listing file from a Fault Analyzer IDILANGX, Debug Tool EQALANGX, or COBOL SYSDEBUG side file.
Starting Debug Tool Utilities

Debug Tool Utilities can be started in two ways. To determine which method to use on your system, contact your system administrator.

To start Debug Tool Utilities, do one of the following:
- 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'

The common_parameters are optional and specify any of the parameters described in Appendix E of [Debug Tool Coverage Utility Users Guide](#). 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 3. 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. To debug a program in full-screen mode, you must compile or assemble your program with the proper options. ["Compiling or assembling your program with the proper compiler options"] gives an overview of the basic compiler or assembler tasks and directs you to other sections of this document that provide more detail.

This chapter gives an overview of basic debugging tasks and directs you to other sections of the document that provide more detail.

Compiling or assembling your program with the proper compiler options

Each programming language has a comprehensive set of compiler options. It’s important to use the correct compiler options to debug your program.

Compiler options to use with C programs
The TEST and DEBUG compiler options provide suboptions to refine debugging capabilities. Use the defaults to gain maximum debugging capability.

Compiler options to use with C++ programs
The TEST and DEBUG compiler options provide suboptions to refine debugging capabilities. Use the defaults to gain maximum debugging capability.

Compiler options to 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 to use with OS/VS 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.

Compiler options to use with PL/I programs
All PL/I programs must use the TEST compiler option and suboptions with the following stipulations:
• Programs compiled with the PL/I for MVS or OS PL/I compilers must also specify the SOURCE suboption.
• The syntax for the TEST compiler option of the Enterprise PL/I compilers is slightly different. Refer to the documentation that corresponds to the version of the compiler you are using for a description of the TEST compiler option.

Assembler options to 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.
Refer to the following sections for more information related to the material discussed in this section.

Related tasks
- “Compiling a C program with the TEST compiler option” on page 37
- “Compiling a C program on an HFS file system” on page 40
- “Compiling your C program with the #pragma statement” on page 41
- “Compiling a C++ program with the TEST compiler option” on page 43
- “Compiling a C++ program on an HFS file system” on page 45
- “Compiling a COBOL program with the TEST compiler option” on page 25
- “Compiling a PL/I program with the TEST compiler option” on page 33
- “Compiling an Enterprise PL/I program on an HFS file system” on page 34
- “Compiling a PL/I for MVS & VM or OS PL/I program” on page 35
- Chapter 10, “Preparing an assembler program,” on page 47

Starting Debug Tool

You have a choice of several ways to start Debug Tool in full-screen mode in TSO:

- By using Debug Tool Utilities, which helps you compile or assemble your program and then start Debug Tool.
- When you start your Language Environment program, include the TEST run-time option as part of your command. For example:

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

  Place the slash (/) before or after the TEST parameter, depending on the programming language you are debugging.
  This is the simplest and most direct way to start Debug Tool. It also starts Debug Tool at the earliest point possible in your program.

- From within your Language Environment program, insert calls to _ctest(), plittest, or ceetest to start Debug Tool. This method is useful when you are familiar with your program and you know where you want debugging to begin.

- To start Debug Tool for a program that does not run in the Language Environment, you must start program EQANMDBG in MVS batch or TSO and pass the name of your program and any required Debug Tool run-time options as part of the PARM string.

When you start your program with the TEST run-time option, the Debug Tool screen appears:
The screen is divided into four sections:

**Session panel header**

The top two lines of the screen are 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.

**Monitor window**

The second part of the screen is the Monitor window. It displays the results of the AUTOMONITOR and MONITOR commands.

**Source window**

The third part of the screen is the Source window. The Source window displays the source or listing file.

**Log window**

The fourth part of the screen is the Log window. It records your interactions with Debug Tool and the results of those interactions.

Now that you are familiar with the Debug Tool interface, you can begin debugging programs.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

Chapter 17, “Starting Debug Tool by using the TEST run-time option,” on page 79

Chapter 18, “Starting Debug Tool from a program,” on page 85

“Starting Debug Tool with CEETEST” on page 85

“Starting Debug Tool with PLTEST” on page 92

“Starting Debug Tool with the _ctest() function” on page 93

Chapter 19, “Starting Debug Tool for batch or TSO programs,” on page 95

Chapter 20, “Starting Debug Tool under CICS,” on page 103

“Entering commands on the session panel” on page 125

“Navigating through Debug Tool session panel windows” on page 130

“Recording and replaying statements” on page 17


**Related references**

"Debug Tool session panel" on page 117

---

**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 sections for more information related to the material discussed in this section.

**Related tasks**

"Stepping through or running your program" on page 137

---

**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 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 sections for more information related to the material discussed in this section.

**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 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 sections for more information related to the material discussed in this section.

**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.
- Continuous display (in the Monitor window) is useful for observing the value of a variable over time.
- 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 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
```

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.

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. You must purchase and install Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45) before using the SET AUTOMONITOR command. The SET AUTOMONITOR command can be used only with specific programming languages, as described in [Debug Tool Reference and Messages](#).

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- “Displaying values of C and C++ variables or expressions” on page 258
- “Displaying values of COBOL variables” on page 232
- “Displaying and monitoring the value of a variable” on page 144

**Related references**
- “Monitor window” on page 121
- *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:

```c
var = 1;
```

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- “Assigning values to C and C++ variables” on page 259
- “Assigning values to COBOL variables” on page 231

---

### 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 sections for more information related to the material discussed in this section.

**Related references**
- *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 sections for more information related to the material discussed in this section.

**Related references**
- [Debug Tool Reference and Messages](#)

## Recording and replaying statements

The commands described in this section are available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

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.**

<table>
<thead>
<tr>
<th>PLAYBACK FORWARD sequence</th>
<th>PLAYBACK BACKWARD sequence</th>
<th>COBOL Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>DISPLAY &quot;CALC Begins.&quot;</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>MOVE 1 TO BUFFER-PTR.</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>PERFORM ACCEPT-INPUT 2 TIMES.</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>DISPLAY &quot;CALC Ends.&quot;</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>GOBACK.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACCEPT-INPUT.</td>
</tr>
<tr>
<td>4, 6</td>
<td>4, 6</td>
<td>ACCEPT INPUT-RECORD FROM A-INPUT-FILE</td>
</tr>
<tr>
<td>5, 7</td>
<td>3, 5</td>
<td>MOVE RECORD-HEADER TO REPROR-HEADER.</td>
</tr>
</tbody>
</table>

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 on page 17.

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.

### 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 on the QQUIT, QUIT ABEND and QUIT DEBUG commands which can stop your debug session.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

[Debug Tool Reference and Messages](#)
Part 2. Preparing your program for debugging
Chapter 4. Planning your debug session and collecting resources

Before you begin debugging, you should plan how you want to conduct your debug session and then collect the resources you need to implement your plan. To help you create your plan, consider following questions:

- Do you want to compile your program with hooks?
- Do you want to reference variables during your debug session?
- Do you want full debug capability or smaller application size and higher performance?
- When do you want to start Debug Tool and when do you want it to gain control?
- Do you want to use Debug Tool in full-screen mode, in batch mode, or in remote debug mode?

The sections in this chapter provide answers to these questions which can help you determine how you want to conduct your debug session.

Refer to the following sections for more information related to the material discussed in this section.

- **Related concepts**
  - “Debug Tool interfaces” on page 4

- **Related tasks**
  - Chapter 5, “Preparing a COBOL program,” on page 25
  - Chapter 7, “Preparing a PL/I program,” on page 33
  - Chapter 8, “Preparing a C program,” on page 37
  - Chapter 9, “Preparing a C++ program,” on page 43
  - Chapter 10, “Preparing an assembler program,” on page 47
  - Chapter 11, “Preparing a DB2 program,” on page 51
  - Chapter 12, “Preparing a DB2 stored procedures program,” on page 55
  - Chapter 13, “Preparing a CICS program,” on page 57
  - Chapter 14, “Preparing an IMS program,” on page 65
  - Chapter 43, “Debugging programs in a production environment,” on page 317

---

**Do you want your program to have hooks?**

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, the compiler inserts hooks into your program.

Assembler, disassembly, and OS/VS COBOL programs are always created without hooks.

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 Release 4, compiler, you can compile your programs without hooks by using the TEST(NOHOOK) compiler option.

To debug these programs or to debug assembler, disassembled, or OS/VS COBOL programs, you must use the Dynamic Debug facility. The Dynamic Debug facility enables you to set breakpoints in a program that does not have hooks. The hooks are inserted at run time whenever you set a breakpoint or enter the step command.

When you compile with one the following compilers and have the compiler insert hooks, you can enhance the program’s performance while you debug it by using the Dynamic Debug facility:
• any COBOL compiler supported by Debug Tool
• any PL/I compiler supported by Debug Tool
• any C/C++ compiler supported by Debug Tool

Some path breakpoints might be unavailable when you use the Dynamic Debug facility.

Do you want to reference variables during your debug session?

If yes, you need to instruct the compiler or assembler to create a symbol table. 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 assembler:
• 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
• High Level Assembler for MVS & VM & VSE, Release 4

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 file by specifying the FORMAT(DWARF) suboption of the DEBUG compiler option.

Debug Tool supports the DEBUG compiler option with z/OS C/C++ Version 1.6 or later.

Do you want full debug capability or smaller application size and higher performance?

Removing hooks, statement tables, or symbol tables can increase the performance of your application and decrease its size. However, debug capabilities are diminished.
To decrease the size of your application, increase performance, and retain most debug capabilities, you need to do the following steps:

1. Compile your program with the appropriate compiler options.
   - For COBOL programs, use the TEST(NONE,SYM,SEPARATE) compiler option, which is available with one of the following compilers:
     - 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
   - For PL/I programs, use the TEST(NOHOOK,SYM,SEPARATE) compiler option, which is available with the Enterprise PL/I for z/OS, Version 3 Release 5, compiler.
   - For C and C++ programs, use the DEBUG(FORMAT(DWARF)) compiler option with z/OS C/C++ Version 1.6 or later.

2. Activate the Dynamic Debug facility. To determine if the Dynamic Debug facility is active, enter the QUERY DYNDDEBUG command after your debug session has started or contact your system administrator.

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 do you want to start Debug Tool and when do you want it to gain control?**

You have a choice of ways to start Debug Tool, as well as ways to let it gain control of your program.

If you are debugging TSO or batch programs, you can start Debug Tool in one of the following ways:
   - You can use Debug Tool Setup Utility, which you can access by using Debug Tool Utilities to start your program. Debug Tool Setup Utility can help you prepare your programs and start Debug Tool.
   - You can use the TEST run-time option to start your Language Environment program. This option gives you the choice of starting Debug Tool at any of these times:
     - Before you run your program
     - When a high-level language (HLL) condition occurs while your program is running
     - When an attention interrupt occurs (except for batch programs)
   - Language Environment provides a run-time service (called CEETEST) that you can call while your program is executing, at the location of your choice. PL/I and C/C++ provide similar services. The PL/I service is called PLITEST and the C/C++ service is called _ctest().
   - You can start Debug Tool by using EQANMDBG to start your MVS batch or TSO program that does not start in Language Environment including OS/VS COBOL programs.

If you are debugging DB2, IMS, or CICS programs, you can start Debug Tool in the way described in the following sections:
   - "Debugging DB2 programs in batch mode" on page 299
After Debug Tool starts, it gains control of your program and suspends execution so that you can take actions such as checking the value of a variable or examining the contents of storage.

**Do you want to use Debug Tool in full-screen mode, in batch mode, or in remote debug mode?**

Decide which interface you want to use when debugging your program. The interface that you choose affects how you start your program, how you start Debug Tool, and how you interact with Debug Tool.

**Collecting your resources**

After you determine how you want to conduct your debug session, collect the resources you need to prepare and debug your programs. The minimum requirement is the source file to compile or assemble your program and access to the compiler or assembler to prepare your program. The following list describes additional resources you might need, depending on the type of program you are debugging:

**To debug a program that is optimized**

If you are debugging a COBOL program that is optimized, you need to save the debug information in a separate debug file and you need to use the Dynamic Debug facility. You also need to use the compilers that support the SEPARATE suboption of the TEST compiler option.

**To debug a program without hooks**

If you are debugging a program without hooks, you need the Dynamic Debug facility to debug your program.

**To debug an assembler or OS/VS COBOL program**

If you are debugging an assembler or OS/VS COBOL program, you need to create the EQALANGX file.

**To debug a program without debug information**

If you are debugging a program without debug information, you need to use the disassembly view and have a hardcopy of the listing available.
Chapter 5. Preparing a COBOL program

Before you debug a program, ensure that the program complies with the requirements described in this book:

- All the data sets required to debug your program comply with the guidelines described in this book.
- All the libraries that your program needs are available.
- Your program is compiled with the appropriate compiler options.

When a program is under development, compile the program with the TEST(ALL) compiler option to get all of Debug Tool’s capabilities. If you use one of the following compilers, you can compile COBOL programs with the TEST(NONE,SYM,SEPARATE) compiler option and retain most of Debug Tool’s capabilities:
- 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

Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 363 describes how to prepare a sample COBOL program and start Debug Tool by using Debug Tool Utilities.

Depending on the compiler and compiler options you select, you need to save the source, listing, or separate debug file, which Debug Tool needs to display your source. Refer to Appendix A, “Data sets used by Debug Tool,” on page 355 for a description of which file you need to save.

Compiling a COBOL program with the TEST compiler option

The suboptions you specify when you compile your COBOL program with the TEST compiler option affect the size and performance of your program and the debugging capabilities available. Before debugging your COBOL program with Debug Tool, you must compile it with the COBOL TEST compiler option. Depending on the suboptions you specify, the compiler does the following:

- Creates the symbol tables.
- Creates debugging information.
- Inserts hooks at selected points in your program.

Debug Tool uses the symbol tables to obtain information about program variables. Programs compiled with one of the following compilers and with the SEPARATE suboption store debugging information and symbol tables in a separate file.
- 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

The file, called a separate debug file, must be a non-temporary file and must also be available during the debug session. If you move or rename the separate debug file, specify the new location by using one of the following methods:

- Entering the SET SOURCE command with the name of the new location
- Entering the SET DEFAULT LISTINGS command with the name of the new location
• Specifying the EQDEBUG DD statement with the name of the new location
• Coding the EQAUEDAT user exit with the new location

Debug Tool uses hooks to gain control of your program at selected points during its execution. The hooks do not modify your source. The hooks are inserted into your program during one of the following times:
• At compile time, when you specify the TEST compiler option with any suboption except NONE. Hooks can be inserted 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), such as before and after a CALL statement.
• At run time, if the Dynamic Debug facility is activated (which is the default). The hooks are inserted when you set a breakpoint or enter the STEP command. If your program has compiled-in hooks and you choose to use the Dynamic Debug facility to take advantage of improved performance, the hooks that are inserted at run time are used instead of the compiled-in hooks.

To debug programs that do not have compiled-in hooks, use the Dynamic Debug facility while you debug and compile your programs with the TEST(NONE) compiler option with one of the following compilers:
  – 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 the Dynamic Debug facility is used to place hooks in programs that reside in read-only storage, the Authorized Debug facility must be installed and you must be authorized to use it. Contact your system administrator to get authorized.

When you use the COBOL TEST compiler option:
• If you specify NUMBER with TEST, make sure the sequence fields in your source code all contain numeric characters.
• Usually, when you specify TEST in combination with any other suboptions (except NONE), the compiler options NOOPTIMIZE and OBJECT automatically go into effect, preventing you from debugging optimized programs. However, if you specify TEST(NONE,SYM) and compile with one of the following compilers, you can specify OPT, allowing you to debug optimized programs:
  – Enterprise COBOL for z/OS and OS/390, Version 3 Release 2
  – 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
• The TEST compiler option and the DEBUG run-time 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.
• If you want to use the DATA suboption of the PLAYBACK ENABLE command, you must specify the SYM suboption of the TEST compiler option when you compile your program.
• For VS COBOL II programs, in addition to the TEST compiler option, you must specify:
  – the SOURCE compiler option. This option is required to generate a listing file.
Converting an OS/VS COBOL program to 1985 COBOL Standard

Programs compiled with the OS/VS COBOL compiler can be debugged by converting them to the 1985 COBOL Standard level and compiling 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 use Load Module Analyzer or CCCA, you must purchase and install Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

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 G, “Debug Tool Load Module Analyzer,” on page 387.

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.
Chapter 6. Preparing an OS/VS COBOL program

This chapter describes how to prepare an OS/VS COBOL program that you can debug with Debug Tool. You must purchase and install Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45), in order to debug an OS/VS COBOL program.

To prepare an OS/VS COBOL program, you must do the following steps:

1. Compile your program with the IBM OS/VS 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. Please read the information regarding non-Language Environment programs for instructions on how to start Debug Tool and debug OS/VS COBOL programs, unless OS/VS COBOL-specific information 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
- NOSYMMDMP
- NOCOUNT

Creating the EQALANGX file

To create the EQALANGX file, you use the EQALANGX program. The EQALANGX program shipped as a component of Debug Tool is functionally equivalent to the IDILANGX program shipped as a component of IBM Fault Analyzer. If you have IBM Fault Analyzer installed, you can use the IDILANGX program to create the EQALANGX file, as long as the version of the IDILANGX program is the same as or newer than the EQALANGX program shipped with Debug Tool. To identify the version of the program, do the following steps:

1. Create the EQALANGX file as described in the IBM Fault Analyzer documentation.
2. Look at the first record of the generated EQALANGX file and make a note of the version.
3. Create the EQALANGX file as described in this section.
4. Look at the first record of the generated EQALANGX file.

If you choose to use IDILANGX to create the EQALANGX file, you can skip these instructions. See the IBM Fault Analyzer documentation for instructions on creating the EQALANGX file.

To create the EQALANGX file, do the following steps:

1. Create JCL similar to the following:
   ```plaintext
   //XTRACT EXEC PGM=EQALANGX,REGION=32M,
   // PARM='(COBOL ERROR LOUD'
   //STPLIB DD DISP=SHR,DSN=hlq.SEQAMOD
   //LISTING DD DISP=SHR,DSN=yourid.osvscompiler.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 an OS/VS COBOL module is being processed.
   
   - **ERROR**
     
     The ERROR parameter is suggested, but optional. If you specify it, additional information is displayed when an error is detected.
   
   - **LOUD**
     
     The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.
   
   - **64K CREF**
     
     The 64K and CREF parameters are optional. Previously, these options were required.

   The messages displayed by specifying the ERROR and LOUD parameters are Write To Operator or Write To Programmer (WTO or WTP) messages. See the *IBM Fault Analyzer for z/OS User's Guide and Reference* for detailed information about the messages and return codes displayed by the IDILANGX program. The EQALANGX program uses the same messages and return codes.

   - **hlq.SEQAMOD**
     
     The name of the data set containing the Debug Tool load modules. If the Debug Tool load modules are in a system linklib data set, you can omit the following line:
     ```plaintext
     //STPLIB DD DISP=SHR,DSN=hlq.SEQAMOD
     ```

   - **yourid.osvscompiler.listing**
     
     The name of the listing data set generated by the IBM OS/VS 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 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 OS/VS COBOL program. If you want the member name of
the EQALANGX debug file to match the name of the OS/VS COBOL
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 7. Preparing a PL/I program

Before you debug a program, ensure that the program complies with the requirements described in this book:

- All the data sets required to debug your program comply with the guidelines described in this book.
- All the libraries that your program needs are available.
- Your program is compiled with the appropriate compiler options.

When a program is under development, you should compile the program with the TEST(ALL) compiler option to get all of Debug Tool's capabilities.

Refer to Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 363 describes how to prepare a sample PL/I program and start Debug Tool by using Debug Tool Utilities.

Depending on the compiler and compiler options you select, you need to save the source, listing, or separate debug file, which Debug Tool needs to display your source. Refer to Appendix A, “Data sets used by Debug Tool,” on page 355 for a description of which file you need to save.

Compiling a PL/I program with the TEST compiler option

The PL/I compiler provides the TEST compiler option and its suboptions to control the placement of hooks and symbol tables. The suboptions BLOCK, STMT, PATH, ALL, and NONE regulate the points at which the compiler inserts hooks. Debug Tool uses these hooks to gain control of the program at select points while it is running.

If all of the following tasks are completed, Debug Tool uses the hooks inserted by the Dynamic Debug facility instead of the hooks inserted by the compiler:

- The Dynamic Debug facility is installed on your system.
- You have not deactivated the Dynamic Debug facility by entering the SET DYNDEBUG OFF command.

Using the hooks inserted by the Dynamic Debug facility provides better performance when you are debugging your program. However, some path breakpoints become unavailable. If you need to use those breakpoints, deactivate the Dynamic Debug facility by entering the SET DYNDEBUG OFF command.

If the Dynamic Debug facility is used to place hooks in programs that reside in read-only storage, the Authorized Debug facility must be installed and you must be authorized to use it. Contact your system administrator to get authorized.

If you compile your program with Enterprise PL/I for z/OS, Version 3 Release 5, you can use the SEPARATE suboption of the TEST compiler option to save debugging information in a separate debug file. The separate debug file must be a non-temporary file and must also be available during the debug session. If you move or rename the separate debug file, specify the new location by using one of the following methods:

- Entering the SET SOURCE command with the name of the new location
- Entering the SET DEFAULT LISTINGS command with the name of the new location
• Specifying the EQADDEBUG DD statement with the name of the new location
• Coding the EQAUEDAT user exit with the new location

You also need to install the Language Environment PTF for APAR PK12833 for z/OS Version 1.4 through Version 1.7 on the system where you are debugging this program.

If you are debugging an Enterprise PL/I program that was compiled without the SEPARATE sub-option of TEST, and the source code is being managed by a library system that requires the SUBSYS=ssss parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=ssss allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

In Enterprise PL/I for z/OS, Version 3 Release 4, a new pair of suboptions was added: HOOK and NOHOOK. These suboptions control the insertion of hook-related information into the program’s object file. If you compile your program with the NOHOOK suboption of the TEST compiler option, you must activate the Dynamic Debug facility.

If you are debugging an Enterprise PL/I program, do not associate SYSIN with an in-stream data set (for example /SYSIN DD *).

Refer to the following sections for more information related to the material discussed in this section.

Related references
- Debug Tool Reference and Messages

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 a PL/I for MVS & VM or OS PL/I program

For OS PL/I programs, you must specify the SOURCE compiler option in addition to the TEST compiler option. The SOURCE compiler option is required to generate a listing file. In addition, you must link your program with the Language Environment SCEELKED library; do not use the OS PL/I PLIBASE or SIBMBASE library.

To be able to view your listing while debugging in full-screen mode, PL/I for MVS & VM and OS PL/I programs must be compiled using the SOURCE compiler option. You must also 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.pgmmname.list in the Source window. If Debug Tool cannot find the listing at this location, use one of the following methods to associate your source listing with the program you are debugging:

• Entering the SET SOURCE command with the name of the location or file.
• Entering the SET DEFAULT LISTINGS command with the name of the location or file.
• Specifying the EQADEBUG DD statement with the name of the location or file.
• Coding the EQAUEDAT user exit with the new location.
Chapter 8. Preparing a C program

Before you debug a program, ensure that the program complies with the requirements described in this book:

- All the data sets required to debug your program comply with the guidelines described in this book. The source must be in a single file and not a concatenation of files.
- All the libraries that your program needs are available.
- Your program is compiled with the appropriate compiler options.

When a program is under development, you can get the full capability of Debug Tool by compiling your program with the TEST(ALL) compiler option.

Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 363 describes how to prepare a sample C program and start Debug Tool by using Debug Tool Utilities.

Depending on the compiler and compiler options you select, you need to save the source, listing, or separate debug file, which Debug Tool needs to display your source. Refer to Appendix A, “Data sets used by Debug Tool,” on page 355 for a description of which file you need to save.

Choosing between TEST and DEBUG compiler options

The C/C++ compiler option DEBUG(FORMAT(DWARF)) 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.

Compile your program with the DEBUG(FORMAT(DWARF)) compiler option to place the debug information in a separate file, which reduces the size of your load module. During a debug session, Debug Tool obtains information from this file and the source file.

You can debug programs that are a combination of compile units compiled with the TEST and DEBUG compiler options.

Refer to the z/OS XL C/C++ User’s Guide for more information about the TEST and DEBUG compiler options.

Related tasks

- “Compiling your C program with the DEBUG(FORMAT(DWARF)) option” on page 39
- “Compiling a C++ program with the DEBUG(FORMAT(DWARF)) option” on page 44

Compiling a C program with the TEST compiler option

Before you test your C program with Debug Tool, you must compile it with the C TEST compiler option:
The suboptions BLOCK, LINE, and PATH regulate the points where the compiler inserts program hooks. When you set breakpoints, they are associated with the hooks that are used to instruct Debug Tool where to gain control of your program.

The suboption SYM regulates the inclusion of symbol tables into the object output of the compiler. Debug Tool uses the symbol tables to obtain information about the variables in the program.

When you use the C TEST compiler option, be aware that:

- The C TEST compiler option generates hooks at entry and exit points for functions.
- 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.

If you are debugging a C program, do not associate SYSIN with an in-stream data set (for example //SYSIN DD *).

You can specify any combination of the C TEST suboptions in any order. The default suboptions are BLOCK, LINE, PATH, and SYM.

If all of the following tasks are completed, Debug Tool uses the hooks inserted by the Dynamic Debug facility instead of the hooks inserted by the compiler:

- The Dynamic Debug facility is installed on your system.
- You have not deactivated the Dynamic Debug facility by entering the SET DYNDDEBUG OFF command.

Using the hooks inserted by the Dynamic Debug facility provides better performance when you are debugging your program. However, some path breakpoints become unavailable. If you need to use those breakpoints, deactivate the Dynamic Debug facility by entering the SET DYNDDEBUG OFF command.

If the Dynamic Debug facility is used to place hooks in programs that reside in read-only storage, the Authorized Debug facility must be installed and you must be authorized to use it. Contact your system administrator to get authorized.

If you are debugging a C program, and the source code is being managed by a library system that requires the SUBSYS=SSSS parameter when the data set is
allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=SSSS allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

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:

   \texttt{cd /u/mike/app}
   \texttt{c89 \-g \-o "//PROJ.LOAD(FRED)" fred.c}

   \textbf{Note:} The double quotes in the command line above are required.

2. Set up your TSO environment, as described above.

3. Debug the program under TSO by entering the following:

   \texttt{FRED TEST ENVAR('PWD=/u/mike/app') / asis}

   \textbf{Note:} The single quotes in the command line above are required.

   \texttt{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 debugging your application in the UNIX System Services Shell, you must debug in remote debug mode or in full-screen mode through a VTAM terminal.

Refer to the following sections for more information related to the material discussed in this section.

\textbf{Related references}

\textit{Debug Tool Reference and Messages}

\section*{Compiling your C program with the DEBUG(FORMAT(DWARF)) option}

The \textit{z/OS XL C/C++ User’s Guide} describes all of the suboptions available with the DEBUG compiler option. If you want to use the FORMAT(DWARF) suboption, consider the following factors:

- 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.

To compile your C or C++ program with the FORMAT(DWARF) suboption so that Debug Tool can debug your program, do the following tasks:

1. Specify the DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL) compiler option or verify that this compiler option with these suboptions is the default. If you specify the FILE suboption, verify that the file specified is a non-temporary file and is available during the debug session.

2. Verify that the program source file is available during the debug session.
It is not recommended that you move the file specified in the FILE suboption (see step 1 on page 39). However, if you move or rename this file, specify the new name or location with the EQADEBUG DD statement. If you are using UNIX System Service’s HFS file system, you can not move the file specified in the FILE suboption.

If you move or rename the program source file, you must let Debug Tool know the new name or location of the file by using one of the following methods:
- Use the SET SOURCE command with the new name or location
- Use the SET DEFAULT LISTINGS command with the new name or location
- Code the EQAEDAT user exit with the new name or location

## 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.

## 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.
**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 sections for more information related to the material discussed in this section.

**Related tasks**
- “Compiling your C program with the #pragma statement”

**Related references**
- 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 run-time options.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 8, “Preparing a C program,” on page 37
- Chapter 9, “Preparing a C++ program,” on page 43
- z/OS XL C/C++ Language Reference
Chapter 9. Preparing a C++ program

Before you debug a program, ensure that the program complies with the requirements described in this book:

- All the data sets required to debug your program comply with the guidelines described in this book. This includes that the source must be in a single file and not a concatenation of files.
- All the libraries that your program needs are available.
- Your program is compiled with the appropriate compiler options.

When a program is under development, you should compile the program with the TEST(ALL) compiler option to get all of Debug Tool’s capabilities.

Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 363 describes how to prepare a sample C++ program and start Debug Tool by using Debug Tool Utilities.

Depending on the compiler and compiler options you select, you need to save the source, listing, or separate debug file, which Debug Tool needs to display your source. Refer to Appendix A, “Data sets used by Debug Tool,” on page 355 for a description of which file you need to save.

Choosing between TEST and DEBUG compiler options

The C/C++ compiler option DEBUG(FORMAT(DWARF)) 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.

Compile your program with the DEBUG(FORMAT(DWARF)) compiler option to place the debug information in a separate file, which reduces the size of your load module. During a debug session, Debug Tool obtains information from this file and the source file.

You can debug programs that are a combination of compile units compiled with the TEST and DEBUG compiler options.

Refer to the z/OS XL C/C++ User’s Guide for more information about the TEST and DEBUG compiler options.

Related tasks

- “Compiling your C program with the DEBUG(FORMAT(DWARF)) option” on page 39
- “Compiling a C++ program with the DEBUG(FORMAT(DWARF)) option” on page 44

Compiling a C++ program with the TEST compiler option

Before testing your C++ program with Debug Tool, you must compile it with the C++ TEST compiler option. This causes the compiler to generate information about your program that Debug Tool needs to help you debug your program.
If you are debugging a C++ program, do not associate SYSIN with an in-stream data set (for example //SYSIN DD *).

If all of the following tasks are completed, Debug Tool uses the hooks inserted by the Dynamic Debug facility instead of the hooks inserted by the compiler:

- The Dynamic Debug facility is installed on your system.
- You have not deactivated the Dynamic Debug facility by entering the SET DYNDEBUG OFF command.

Using the hooks inserted by the Dynamic Debug facility provides better performance when you are debugging your program. However, some path breakpoints become unavailable. If you need to use those breakpoints, deactivate the Dynamic Debug facility by entering the SET DYNDEBUG OFF command.

If the Dynamic Debug facility is used to place hooks in programs that reside in read-only storage, the Authorized Debug facility must be installed and you must be authorized to use it. Contact your system administrator to get authorized.

If you are debugging a C++ program, and the source code is being managed by a library system that requires the SUBSYS= parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS= allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

[Debug Tool Reference and Messages](#)

## Compiling a C++ program with the DEBUG(FORMAT(DWARF)) option

The z/OS XL C/C++ User’s Guide describes all of the suboptions available with the DEBUG compiler option. If you want to use the FORMAT(DWARF) suboption, consider the following factors:

- 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.

To compile your C or C++ program with the FORMAT(DWARF) suboption so that Debug Tool can debug your program, do the following tasks:

1. Specify the DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL) compiler option or verify that this compiler option with these suboptions is the default. If you specify the FILE suboption, verify that the file specified is a non-temporary file and is available during the debug session.

2. Verify that the program source file is available during the debug session.

It is not recommended that you move the file specified in the FILE suboption (see step 1). However, if you move or rename this file, specify the new name or
location with the EQADebug DD statement. If you are using UNIX System
Service's HFS file system, you can not move the file specified in the FILE
suboption.

If you move or rename the program source file, you must let Debug Tool know the
new name or location of the file by using one of the following methods:
• Use the SET SOURCE command with the new name or location
• Use the SET DEFAULT LISTINGS command with the new name or location
• Code the EQAuedat user exit with the new name or location

### 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.

### 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.

### 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 sections for more information related to the material discussed in this section.

- **Related tasks**
  - Chapter 8, “Preparing a C program,” on page 37

- **Related references**
  - z/OS XL C/C++ User’s Guide
Chapter 10. 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.

You must have Debug Tool Utilities and Advanced Functions (5655-R45) installed on your system to prepare and debug assembler programs.

You can debug an assembler program in remote debug mode only with the following products:
• WebSphere Developer Debugger for zSeries
• Compiled Language Debugger component of WebSphere Studio Enterprise Developer
• Compiled Language Debugger component of WebSphere Developer for zSeries

Before you begin

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” on page 48
2. “Creating the EQALANGX file” on page 48

“Assembling your program and creating EQALANGX” on page 49 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.

Creating the EQALANGX file

To create the EQALANGX file, you use the EQALANGX program. The EQALANGX program shipped as a component of Debug Tool is functionally equivalent to the IDILANGX program shipped as a component of IBM Fault Analyzer. If you have IBM Fault Analyzer installed, you can use the IDILANGX program to create the EQALANGX file, as long as the version of the IDILANGX program is the same as or newer than the EQALANGX program shipped with Debug Tool. To identify the version of the program, do the following steps:

1. Create the EQALANGX file as described in the IBM Fault Analyzer documentation.
2. Look at the first record of the generated EQALANGX file and make a note of the version.
3. Create the EQALANGX file as described in this section.
4. Look at the first record of the generated EQALANGX file.

If you choose to use IDILANGX to create the EQALANGX file, you can skip these instructions. See the IBM Fault Analyzer documentation for instructions on creating the EQALANGX file. 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=hlq.SEQAMOD
//SYSADATA DD DISP=SHR,DSN=yourid.sysadata
//IDILANGX DD DISP=OLD,DSN=yourid.EQALANGX
```

The following list describes the variables used in this example the parameters you can use with the EQALANGX program:

**PARM=**

- **ASM**
  - Indicates that an assembler module is being processed.

- **ERROR**
  - This parameter is suggested but optional. If you specify it, additional information is displayed when an error is detected.

- **LOUD**
  - The LOUD parameter is suggested, but optional. If you specify it, additional informational and statistical messages are displayed.

The messages displayed by specifying the ERROR and LOUD parameters are Write To Operator or Write To Programmer (WTO or WTP) messages. See the IBM Fault Analyzer for z/OS User's Guide and Reference for detailed information about the messages and return codes displayed by the IDILANGX program. The EQALANGX program uses the same messages and return codes.
**hlq.SEQAMOD**  
The name of the data set containing the Debug Tool load modules. If the Debug Tool load modules are in a system linklib data set, you can omit the following line:
```
//STEPLIB DD DISP=SHR,DSN=hlq.SEQAMOD
```

**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 the [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. 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 existed, you are asked to verify the options used to create them.
6. If you specified that the processing be completed by batch, the JCL created to run the batch job is displayed. Verify that the JCL is correct, type Submit in the command line, press Enter and then press PF3.
7. After the processing is completed, the High Level Assembler - View Outputs panel is displayed. This panel displays the return code of each process completed and enables you to view, edit, or browse the input and output data sets.

To read more information about a field in any panel, place the cursor in the input field and press PF1. To read more information about a panel, place the cursor anywhere on the panel that is not an input field and press PF1.
After you assemble your program and create the EQALANGX file, you can link-edit your program.

**Link-editing your program**

You can link-edit your program by using your normal link-edit procedures or you can use 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.
Chapter 11. 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 52.  
3. “Binding DB2 programs for debugging” on page 53.

Refer to the following sections for more information related to the material discussed in this section.

Related references
DB2 UDB for z/OS Application Programming and SQL Guide

Processing SQL statements

Your must run your program through the DB2 preprocessor or coprocessor, which processes SQL statements, either prior to 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. Then link your program as described in “Linking DB2 programs for debugging” on page 52.

• 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.

• If you are preparing a PL/I program using Enterprise PL/I for z/OS and OS/390 Version 3 Release 1 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.
  – Use the PP(SQL("option,...")) compiler option so that the SQL statements are processed by the DB2 coprocessor during compilation. Save the program
source file that you used as input to the compiler. Then link your program as described in "Linking DB2 programs for debugging."

If you are preparing a program using Enterprise PL/I for z/OS Version 3 Release 5 and you specify the SEPARATE suboption of the TEST compiler option, you must also save the separate debug file.

- 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. Then link your program as described in "Linking DB2 programs for debugging."

- If you are using an assembler program, assemble or compile your program using the output of the DB2 precompiler. Run EQALANGX on the output of that 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 sections for more information related to the material discussed in this section.

**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 run-time options module, CEEUOPT, by doing the following:

1. Find the user run-time options program CEEUOPT in the Language Environment SCEESAMP library.

2. Change the NOTEST parameter into the desired TEST parameter. For example:
   ```
   old: NOTEST=(ALL,*,PROMPT,INSPPREF),
   new: TEST=(*,*;,*),
   ```
   If you are using remote debug mode, specify the TCPIP suboption, as in the following example:
TEST=(,,,TCPIP&9.24.104.79%8001:*)

Note: Double ampersand is required.

If you are use a full-screen mode through a VTAM terminal session without the Debug Tool Terminal Interface Manager, specify the MFI suboption with a VTAM LU name, as in the following example:
Test=(,,,MFI%TRMLU001)

If you are use a full-screen mode through a VTAM terminal session with the Debug Tool 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.

```assembly
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
    CEEOPT TEST=(,*,;,*)
END
```

The user run-time options program can be assembled with predefined TEST
run-time 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
(DSNxxx).

Refer to the following sections for more information related to the material
discussed in this section.

**Related tasks**

Chapter 18, “Starting Debug Tool from a program,” on page 85

**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 12. Preparing a DB2 stored procedures program

The DB2 administrator must define the address space where the stored procedure runs. This can be a DB2 Address Space or a Workload Manager (WLM) Address Space. This address space is assigned a name which is used to define the stored procedure to DB2. In the JCL for the DB2 or WLM address space, verify that the following data sets are defined in the STEPLIB concatenation and have the appropriate RACF® Read authorization for programs to access them:

- LOADLIB for the stored procedure
- SEQAMOD for Debug Tool
- SCEERUN for Language Environment

After updating the JCL, the DB2 administrator must recycle the DB2 or WLM address space so that these updates take effect.

To debug a stored procedure, you must compile it with the TEST compiler option and run with the TEST run-time option. If the stored procedure is defined as Type=SUB, the following considerations apply:

- You cannot use the Dynamic Debug facility. If you have the facility installed, use the SET DYNDEBUG OFF command as the first command at the start of your debug session.
- CEEUOPTS is ignored by Language Environment.

If you are going to debug the stored procedure while another user runs it, do the following tasks:

- Compile the stored procedure with the TEST(ALL,SYM) compiler option. Do not compile it with the TEST(NONE,SYM) compiler option.
- You cannot use the Dynamic Debug facility. If you have the facility installed, enter the SET DYNDEBUG OFF command as the first command at the start of your debug session.

- See [DB2 Application Programming and SQL Guide](#) for information on running a stored procedure concurrently.

The TEST runtime options must be defined to the RUNOPTS field of the DB2 catalog by using the appropriate DB2 SQL commands. In the following example, the stored procedure is a COBOL program called SPROC1 of Type SUB that runs in a WLM address space called WLMENV1. The stored procedures are debugged in remote mode.

```
create procedure sproc1
  language cobol
  external name sproc1
  parameter style general
  wlm environment wlmenv1
  run options 'TEST(,,,TCP/IP&9.112.27.9980001:*)'
  program type sub;
```

To verify that the stored procedure is defined correctly, use the following SQL SELECT command on the appropriate DB2 table:

```
select * from sysibm.sysroutines;
```
If the definition is not correct, use the following SQL command to modify the stored procedure:

```
alter procedure sproc1 run options 'TEST(,,,TCPPIP&9.112.27.21%8001:*)';
```

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

*DB2 UDB for z/OS Application Programming and SQL Guide*
Chapter 13. Preparing a CICS program

To prepare a CICS program for debugging, you must do the following tasks:

1. Complete the program preparation tasks for assembler, C, C++, COBOL, or PL/I, as described in the following sections:
   - Chapter 5, “Preparing a COBOL program,” on page 25
   - Chapter 7, “Preparing a PL/I program,” on page 33
   - Chapter 8, “Preparing a C program,” on page 37
   - Chapter 9, “Preparing a C++ program,” on page 43
   - Chapter 10, “Preparing an assembler program,” on page 47

If you are using versions of CICS earlier than CICS Transaction Server for z/OS Version 3 Release 1, you can prepare OS/VS COBOL programs for debugging, as described in Chapter 6, “Preparing an OS/VS COBOL program,” on page 29.

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 58.
   - 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 “Using CADP to manage debugging profiles” on page 63 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 either 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 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
  - Running with Language Environment Version 1 Release 3 or later, with the PTF for APAR PK03093 applied

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 resource IDs that you want to debug. See “Creating and storing a DTCN profile.”

Creating and storing a DTCN profile

The DTCN transaction stores one profile for each DTCN terminal in a repository. 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.

Profiles are either active or inactive. If profiles are active, they are used for pattern matching. Inactive profiles are skipped. You can change the status of a profile by using the Debug Tool CICS Control - Primary Menu panel.

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.

```
<table>
<thead>
<tr>
<th>DTCN</th>
<th>Debug Tool CICS Control - Primary Menu</th>
<th>S07CICPD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Select the combination of resources to debug (see Help for more information)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Terminal Id =&gt; 0090</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction Id =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program Id(s) =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User Id =&gt; CICSUSER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NetName =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP Name/Address =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Select type and ID of debug display device</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Session Type =&gt; MFI, TCP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Port Number =&gt; TCP Port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Display Id =&gt; 0090</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Generated String: TEST(ALL,'<em>',PROMPT,'MFI%0090:</em>')</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Repository String: No string currently saved in repository</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Profile Status: No Profile Saved. Press PF4 to save current settings.</td>
<td></td>
</tr>
<tr>
<td>PF1=HELP 2=HELP 3=EXIT 4=SAVE 5=ACT/INACT 6=DELETE 7=SHOW 9=OPTION</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 next two steps and proceed to step 1 on page 61. If you want to change the settings on this panel, continue to the next step.

2. Specify the combination of resource IDs that you want to debug.

**Terminal Id**

Specify the CICS terminal to debug. By default, this ID is set to the terminal that is currently running DTCN. If the DTCNFORCETERMID option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.
**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 the DTCNFORCETRANID option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.

**Program Id(s)**

Specify the program or programs that you want to debug. You can specify any of the following programs:

- Any CICS program if it is invoked as an Language Environment enclave or over a CICS Link Level. This includes the following types of programs:
  - The initial program in a transaction
  - A program invoked by CICS LINK or XCTL
- Any COBOL program, even if it is a nested program 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 program (with the PTF for APAR PK03264 applied) running with Language Environment Version 1 Release 3 or later (with the PTF for APAR PK03093 applied), even if it is a nested program or a subprogram within a composite load module, invoked as a static or dynamic CALL.

When you specify a program ID 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 program ID 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.ENTPLSOURCE(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.ENTPLSOURCE.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.ENTPLSOURCE and DEV1.TEST.C.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.
- Language Environment Version 1.4 through 1.8 with the PTF for APAR PK33738 applied.

Specifying a CICS program in the Program Id(s) field is similar to setting a breakpoint by using the AT ENTRY command and Debug Tool stops each time you enter this program.

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 Program Id(s) field. Use the AT APPEARANCE or AT ENTRY command to stop at this CICS program.

If the DTCNFORCEPROGID option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.

**User Id**
Specify the CICS user ID to debug. All programs that are run by this user will start Debug Tool. If the DTCNFORCEUSERID option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.

**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 the DTCNFORCENCETNAME option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.

**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 the DTCNFORCETIP option in EQAOPTS is set to YES, this field must be specified. See [Debug Tool Customization Guide](#) for a description of these options.

For more information about these fields, place the cursor next to the field and press PF1 to display the online help.

3. Specify the type of debugging and the ID of the display device.

**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, IBM Distributed Debugger uses port 8000. By default, the following products use port 8001:

- WebSphere Developer Debugger for zSeries
• Compiled Language Debugger component of WebSphere Studio Enterprise Developer
• Compiled Language Debugger component of WebSphere Developer for zSeries

Display Id
Identifies the target destination for Debug Tool information. Depending on the session type that you’ve selected, the display ID is one of the following:

• If you selected MFI, the display ID is a CICS 3270 terminal ID. This ID is set by default to the terminal ID that is currently running DTCN, but you can change this to direct MFI screens to a different CICS terminal.
• If you selected TCP, enter either the IP address or host name of the workstation that will display the debug screens. For the debug session to start, the appropriate software must be running on that workstation.

4. Specify the debugging options by pressing PF9 to display the secondary options menu, shown below.

<table>
<thead>
<tr>
<th>DTCN</th>
<th>Debug Tool CICS Control - Menu 2</th>
<th>S07CICPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Debug Tool options</td>
<td>Test Option ==&gt; TEST</td>
<td>Test/Notest</td>
</tr>
<tr>
<td>Test Option ==&gt; TEST</td>
<td>Test Level ==&gt; ALL</td>
<td>All/Error/None</td>
</tr>
<tr>
<td>Test Level ==&gt; ALL</td>
<td>Commands File ==&gt; *</td>
<td></td>
</tr>
<tr>
<td>Commands File ==&gt; *</td>
<td>Prompt Level ==&gt; PROMPT</td>
<td></td>
</tr>
<tr>
<td>Prompt Level ==&gt; PROMPT</td>
<td>Preference File ==&gt; *</td>
<td></td>
</tr>
</tbody>
</table>
| Preference File ==> * | Any other valid Language Environment options | ==>

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 on page 62. If you want to change the settings on this panel, continue with this step.

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.

Command File
A valid fully qualified data set name that specifies the primary commands file for this run.

Note: Do not enclose the name of the data set in single or double quotes.

Prompt Level
Specifies whether Debug Tool is started at Language Environment initialization.
Preference File

A valid fully qualified data set name that specifies the preference file to be used. Do not enclose the name of the data set in single or double quotes.

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](https://www.ibm.com) or contact your CICS system programmer.

5. Press PF3 to return to the main DTCN panel.

6. 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.

7. 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.

8. 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 resource IDs that you specified in the previous steps will start Debug Tool.

To display all of the active DTCN profiles in the CICS region, press PF7. The Debug Tool CICS Control - All Sessions screen displays, shown below.

```
<table>
<thead>
<tr>
<th>Owner</th>
<th>Sta</th>
<th>Term</th>
<th>Tran</th>
<th>User Id</th>
<th>NetName</th>
<th>Applid</th>
<th>Display Id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0090</td>
<td>ACT</td>
<td>0090</td>
<td>TRN1</td>
<td>USER1</td>
<td>0072</td>
<td>S07CICPD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>Name/Address</td>
<td>______</td>
<td>______</td>
<td>______</td>
<td>______</td>
<td>_______</td>
<td></td>
</tr>
</tbody>
</table>
```

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.

Program(s)
The values that were entered on the main DTCN screen in the Program Ids field.

IP Name/Address
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 preference file, with the NOTEST option to supply additional defaults to DTCN.

Sharing DTCN repository profile items among CICS systems
The DTCN repository is a CICS temporary storage queue, named EQADTCN2. If you want to share the repository among CICS systems, define the queue as REMOTE in your CICS temporary storage tables (TST). This setting stores a profile item in one CICS system, and makes it readable to another system.

Using CADP to manage debugging profiles
CICS Transaction Server for z/OS Version 2 Release 3 introduces the following two features:
• A new utility transaction, called CADP, to manage debugging profiles. If you use CADP, you cannot use DTCN.
• A new system initialization parameter called DEBUGTOOL. To use the CADP transaction, the DEBUGTOOL system initialization parameter option must be set to YES.


Starting non-Language Environment Debug Tool 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 OS/VS 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,
XEIIN, 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 sections for more information related to the material discussed in this section.

Related tasks
[Debug Tool Customization Guide]

**Passing run time parameters into the non-Language Environment debug session on startup**

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 run-time options that will take effect in your non-Language Environment debugging session. These run-time options and their settings include the following:

- **TEST/NOTEST**: must be TEST
- **TEST LEVEL**: must be ALL
- Commands file
- Prompt Level: must be PROMPT
- Preference 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 sections for more information related to the material discussed in this section.

Related references
[Debug Tool Reference and Messages]
Chapter 14. Preparing an IMS program

This section describes the following two ways to prepare an IMS program:

- By using the Program Preparation, Manage and Use Debug Tool Setup Files, and Manage IMS Programs functions in Debug Tool Utilities. To use Program Preparation and Manage IMS Programs functions, you must purchase and install Debug Tool Utilities and Advanced Functions for z/OS, Version 7 Release 1 (5655-R45).
- Without using Debug Tool Utilities.

This section also describes how to prepare an IMS MPP program that does not run in Language Environment.

Preparing an IMS program with Debug Tool Utilities

If you use Debug Tool Utilities to help you prepare and start your IMS programs, do the following tasks:

1. Compile and link your IMS program by using the Program Preparation tool in Debug Tool Utilities. See "Compiling and linking your IMS program" for more information.

2. Create and customize a setup file. See "Creating setup file for your IMS program by using Debug Tool Utilities" on page 66 for more information.

3. For IMS Version 8 programs, set up the run-time options using Debug Tool Utilities. See "Setting up the run-time options for your IMS Version 8 TM program by using Debug Tool Utilities" on page 66 for more information.

Compiling and linking your IMS program

Appendix C, “Examples: Preparing programs and modifying setup files with Debug Tool Utilities,” on page 363 describes how to compile and link sample programs provided by Debug Tool. You can study these examples to guide you through the compilation and linking process of your program. This section describes specific options you must use.

In step 6 on page 365 verify that you are compiling your program with the TEST compiler option.

In step 4 on page 366 you might have to link in the CEEUOPT module, depending on the following conditions:

- For IMS Transaction Manager (TM) programs running in IMS Version 8, you do not have to link in the CEEUOPT module to obtain information about the TEST run-time options. You can specify the transaction settings and TEST run-time options to use to run your program by using the Manage IMS Programs tool in Debug Tool Utilities. See "Setting up the run-time options for your IMS Version 8 TM program by using Debug Tool Utilities" on page 66 for more information on how to do this task.

- For IMS batch programs, you must link in the CEEUOPT module to obtain information about the TEST run-time options. See "Linking IMS programs for debugging” on page 67 for more information on how to do this task.
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) program. 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 10.
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.

Setting up the run-time options for your IMS Version 8 TM program by using Debug Tool Utilities

To specify the transaction settings and TEST run-time options for your IMS program, do the following:

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 LE run-time 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 LE run-time 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).

**Linking IMS programs for debugging**

When you link your IMS TM Version 8 program, you can manage the TEST
run-time options without linking in a copy of CEEUOPT. For instructions on
managing the TEST run-time options, see “Setting up the run-time options for your
IMS Version 8 TM program by using Debug Tool Utilities” on page 66.

When you link your IMS batch program or IMS TM Version 7 or earlier program,
you must include a run-time options module in your program link. They must be
coded and assembled in a user-defined run-time option module. For instructions on
how to create the CEEUOPT run-time options module using the CEEUOPT macro,
follow the steps in “Linking DB2 programs for debugging” on page 52.

For COBOL programs using IMS, include the IMS interface module DFSLI000 from
the IMS RESLIB library.

**Preparing an IMS program without Debug Tool Utilities**

If you do not have Debug Tool Utilities and Advanced Functions (5655-R45)
installed on your system, you can continue to compile, link, and start your IMS
program by using your site procedures. Remember the following points:
   - Your program must be compiled with the TEST compiler option. Use the default
     options to gain maximum debugging facilities.
   - Ensure that your source, listing, or separate debug file is stored in a permanent
data set that is available to Debug Tool.
   - Ensure that you do the tasks described in “Linking IMS programs for
debugging.”

For IMS TM programs running in IMS Version 8, you do not have to link in the
CEEUOPT module to obtain information about the TEST run-time options. You
can specify the transaction settings and TEST run-time options to use to run your
program by using IMS Version 8 commands. See IMS Version 8 Command
Reference for more information on how to do this task.
Preparing an IMS MPP program that does not run in Language Environment

You can debug an IMS MPP program that does not run in Language Environment. Prepare your program as you normally do, then see "Debugging non-Language Environment IMS MPP programs" on page 307 for instructions on how to start the debugging session.
Chapter 15. Preparing a program by using the LE exit routine

Debug Tool Utilities and Advanced Functions (5655-R45) provides a customized version of the Language Environment user exit routine (CEEBXITA) to link into the application load module. The routine returns a TEST runtime option when called by the Language Environment initialization logic.

The routine extracts the TEST runtime option from a data set with a name that is constructed dynamically from a naming pattern, which includes the user ID token &USERID. This token is replaced with the user ID of the current user during name construction. Each user can specify an individual TEST runtime option when debugging an application.

Debug Tool provides the user exit routine in two forms:

- A load module that the user includes in the link-edit step of his or her application build job. 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.DBGT00L.EQA0OPTS and the default message display level is X'00'.

- Sample assembler routine that you can edit. The assembler routines for the three environments are in the hlq.SEQASAMP data set. You can also merge this source with an existing version of CEEBXITA. Use this source code if you want naming patterns or message display levels that are different than the default values.

Three different exit routines are provided. The load module form of these routines are in the hlq.SEQAMOD data set. The sample assembler routines are in the hlq.SEQASAMP data set.

Table 5. LE exit routines for various environments

<table>
<thead>
<tr>
<th>Environment</th>
<th>Exit routine name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 stored procedures of type MAIN that run in WLM-established address spaces</td>
<td>EQADDCXT</td>
</tr>
<tr>
<td>IMS TM</td>
<td>EQADICXT</td>
</tr>
<tr>
<td>Batch</td>
<td>EQADBCXT</td>
</tr>
</tbody>
</table>

To prepare a program by using the LE exit routine:

1. **“Editing the source code of CEEBXITA (optional)”**
2. “Linking the exit routine into the program” on page 71
3. “Creating the TEST runtime option data set” on page 71

**Editing the source code of CEEBXITA (optional)**

You can edit the sample assembler routine that is provided in hlq.SEQASAMP to customize the naming patterns or message display level. Copy the assembler routine that has the same name as the exit routine 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 TEST runtime option data set is in the form of a sequential data set name. One of the name-qualifiers must contain a &USERID token, which is substituted with the user ID of the current user that is dynamically obtained from the system.

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, a prefix character of P forms P&USERID, which is a valid name qualifier after the current user ID is substituted for &USERID.

The default naming pattern is &USERID.DBGTOOL.EQAUOPTS. This is the pattern that is in the load module provided in hlq.SEQAMOD.

The following examples are data set naming patterns and the corresponding data set names after user ID substitution.

Table 6. Data set naming patterns and data set names

<table>
<thead>
<tr>
<th>Naming pattern</th>
<th>User ID</th>
<th>Name after user ID substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;USERID.DBGTOOL.EQAUOPTS</td>
<td>JOHNDOE</td>
<td>JOHNDOE.DBGTOOL.EQAUOPTS</td>
</tr>
<tr>
<td>P&amp;USERID.EQAUOPTS</td>
<td>123456</td>
<td>P123456.EQAUOPTS</td>
</tr>
<tr>
<td>DT.&amp;&amp;USERID.TSTOPT</td>
<td>TESTID</td>
<td>DT.TESTID.TSTOPT</td>
</tr>
</tbody>
</table>

To customize the naming pattern of the TEST runtime option data set, change the value of the DSNT DC statement in the sample routine. 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 routine. For example:

* The following switch is to control WTO message display level.
* X'00' - no messages
Linking the exit routine into the program

To use the user exit routine CEEBXITA, you must link it into the application program. The following sample JCL links the user exit routine with the program TESTPGM. If you have customized the user exit routine, 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)
//*
```

Creating the TEST runtime option data set

The TEST runtime data set contains the values for the program name list, the TEST runtime option string, and other LE runtime option strings. You can use option 6 of the Debug Tool Utilities ISPF panel, "Manage TEST Run-time Option Data Set," to create this data set.

The data set has the following requirements:
- Sequential data set (DSORG=PS)
- Record format and length requirements:
  - RECFM(F) or RECFM(FB) and LRECL >=80
  - RECFM(V) or RECFM(VB) and LRECL >=84
- Not an HFS data set
- No line numbers
- Contents all upper case
- Name follows the naming pattern in the exit routine

The following three types of values are in the data set:

**Program name list**

The list of programs that are to be debugged starts with the `<PGM>` keyword. The data set must begin with this list. You can specify up to eight names, with each name separated by a comma. Each name can be up to eight characters. A wild card (*) at the end of a name indicates that zero or more characters can follow the specified characters. You can specify the wild card by itself to mean
that any program can be debugged. If the name of a program does not match any name on the list, the exit routine returns without generating a TEST runtime option string.

**TEST runtime option string**

The required TEST runtime option string starts with the `<TST>` keyword and follows the program name list. If the string is long, you can break it into multiple records. Put the breakpoint immediately after a comma because a space is inserted between records when the string is reconstructed. See [Debug Tool Reference and Messages](#) for the syntax of the string.

**Other LE runtime options string**

Other, optional LE runtime options follow the TEST runtime option string and starts with the `<RTO>` keyword. If the string is long, you can break it into multiple records. Put the breakpoint immediately after an option because a space is inserted between records when the string is reconstructed.

The following examples show the contents of a TEST runtime option data set.

```
<PGM>*
<TST>TEST(ALL,*,PROMPT,TCPIP&9.30.60.119%8001:*)
<PGM>*
<TST>TEST(ALL,*,PROMPT,
<TST>TCPIP&9.30.60.119%8001:*)
<RTO>TRAP(ON)
```
Part 3. Starting Debug Tool
Chapter 16. Starting Debug Tool from the Debug Tool Utilities

The ‘Manage and Use Debug Tool Setup Files’ function (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 sections for more information related to the material discussed in this section.

Related tasks
- "Creating the setup file"
- "Editing an existing setup file"
- "Saving your setup file" on page 78
- "Starting your program" on page 78

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 Manage and Use Debug Tool Setup Files 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. 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 sections for more information related to the material discussed in this section.

Related tasks
- "Entering file allocation statements, run-time options, and program parameters" on page 76

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 Manage and Use Debug Tool Setup Files 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 sections for more information related to the material discussed in this section.

**Related tasks**

"Entering file allocation statements, run-time 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.

---

**Entering file allocation statements, run-time 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 run-time 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. OS/VS COBOL programs do not run in Language Environment. If you are debugging an OS/VS COBOL, 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 run-time 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 run-time 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.

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 sections for more information related to the material discussed in this section.

**Related tasks**

"Saving your setup file" on page 78
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 sections for more information related to the material discussed in this section.

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 17. Starting Debug Tool by using the TEST run-time option

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 “Programs that start outside of Language Environment” on page 98.

To specify how Debug Tool gains control of your application and begins a debug session, you can 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.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 18, “Starting Debug Tool from a program,” on page 85

**Related references**
- “Special considerations while using the TEST run-time option”

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/NOSTEST 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, PLittest, 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 GO), all subsequent commands are discarded. However, USE files started from within a primary commands file take on the characteristics of the primary commands file and can be run until complete.

The initial command list, whether it consists of a command string included in the run-time options or a primary commands file, can contain a USE command to get commands from a secondary file. If started from the primary commands file, a USE file takes on the characteristics of the primary commands file.

Running in batch mode

In batch mode, when the end of your commands file is reached, a GO command is run at each request for a command until the program terminates. If another command is requested after program termination, a QUIT command is forced.

Starting 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 sections for more information related to the material discussed in this section.

**Related tasks**

- "Link-editing EQADCCXT into your program" on page 57

**Related references**

- [Debug Tool Reference and Messages](#)

---

**Precedence of Language Environment run-time options**

The Language Environment run-time 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 run-time options table. You can edit this table by using Debug Tool Utilities.
3. Options specified at the invocation of your application, using the TEST run-time option, unless accepting run-time options is disabled by Language Environment (EXECOPS/NONEEXECOPS).
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, 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 sections for more information related to the material discussed in this section.

**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, INSREF).

**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:)
For CICS dual terminal and CICS batch, Debug Tool is started on the
terminal F000 at the end of the environment initialization.

TEST(ALL,,MFI:TRMLU001:)
For use with a full-screen mode through a VTAM terminal without using
the Debug Tool 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 through a VTAM terminal.
- You must specify a network identifier.
- You are not using the Debug Tool Terminal Interface Manager

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 a full-screen mode through a VTAM terminal when using the
Debug Tool 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&9.24.104.79%8001:*)
NOTEST(,,TCPIP&9.24.111.55%8001:*)

Refer to the following sections for more information related to the material
discussed in this section.

Related references
z/OS Language Environment Programming Guide

Specifying additional run-time options with 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 sections for more information related to the material discussed in this section.

**Related tasks**

- [z/OS XL C/C++ User’s Guide](https://www.ibm.com)
Chapter 18. 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 "Programs that start outside of Language Environment" on page 98.

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 PLTEST or by including a PLIXOPT string that specifies the correct TEST run-time suboptions to start Debug Tool.

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 NOTEST, CEEOPT, or other "indirect" settings.

Example: using CEETEST to start Debug Tool from C/C++" on page 88
Example: using CEETEST to start Debug Tool from COBOL" on page 89
Example: using CEETEST to start Debug Tool from PL/I" on page 90

**Related tasks**
- "Starting Debug Tool with CEETEST"
- "Starting Debug Tool with PLTEST" on page 92
- "Starting Debug Tool with the __ctest() function" on page 93
- "Starting Debug Tool by using CEEOPT" on page 106

Refer to the following sections for more information related to the material discussed in this section.

**Related references**
- "Special considerations while using the TEST run-time option" on page 79

## 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++

\[
\text{void CEETEST(} \text{string_of_commands, } fc)) ;
\]

For COBOL

\[
\text{CALL "CEETEST" USING string_of_commands, } fc ;
\]

For PL/I

\[
\text{CALL CEETEST(} \text{string_of_commands, } fc)) ;
\]

\text{string_of_commands (input)}

Halfword-length prefixed string containing a Debug Tool command list. The command string \text{string_of_commands} is optional.

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

If \text{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 through a VTAM terminal
- remote debug mode

To indicate that you want to start Debug Tool in full-screen mode through a VTAM terminal without the Debug Tool Terminal Interface Manager, specify the MFI suboption of the TEST runtime option with the LU name of the VTAM terminal. For example, you can code the following call in your PL/I program:

\[
\text{Call CEETEST('MFI%TRMLU001:*:Query Location;Describe CUS;',*,)}
\]

For a COBOL program, you can code the following call:

\[
01 \text{ PARMS.}
05 \text{ LEN PIC S9(4) BINARY Value 43.}
05 \text{ PARM PIC X(43) Value 'MFI%TRMLU001:*:Query Location;Describe CUS;'.}
\]

\[
\text{CALL "CEETEST" USING PARMS FC.}
\]

To indicate that you want to start Debug Tool in full-screen mode through a VTAM terminal with the Debug Tool 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:

\[
\text{Call CEETEST(VTAM%USERABCD:*:Query Location;Describe CUS;*,)}
\]
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 with the Distributed Debugger, specify the TCPIP suboption of the TEST runtime option with the IP address and port number that the Distributed Debugger is listening to. For example, you can code the following call in your PL/I program:

```
Call CEETEST('TCPIP&your.company.com%8000:*;',*);
```

To indicate that you want to start Debug Tool in remote debug mode with one of the following remote debuggers, specify the TCPIP suboption of the TEST runtime option with the IP address and port number that the remote debugger is listening to:

- WebSphere Developer Debugger for zSeries
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- Compiled Language Debugger component of WebSphere Developer for zSeries

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 CEEDE00 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.

**Usage notes**

**C and C++**

Include `leawi.h` header file.
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.

```c
#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 60 is executed.
Note: If you include a STEP or GO in your command string, all commands after that are not processed. The command string operates like a commands file.

```c
#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.

```c
#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.
```

In the DATA DIVISION of your program. You are not prompted for commands.

```
CALL "CEETEST" USING COMMAND-STRING 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 CEETEST(*,*) ; /* 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.

```assembly
DCL ch char(50)
  init('AT STATEMENT 10 DO; LIST(x); LIST(y); ENO; GO;');
DCL 1 fb,
  5 Severity Fixed bin(15),
  5 MsgNo  Fixed bin(15),
  5 flags,
    8 Case  bit(2),
    8 Sev   bit(3),
    8 Ctrl  bit(3),
  5 FACID  Char(3),
  5 I_S_info Fixed bin(31);

DCL CEETEST ENTRY (CHAR(*) VAR OPTIONAL,
  1 optional,
  254 real fixed bin(15), /* MSGSEV */
  254 real fixed bin(15), /* MSGNUM */
  254 /* Flags */,
    255 bit(2), /* Flags Case */
    255 bit(3), /* Flags Severity */
    255 bit(3), /* Flags Control */
  254 char(3), /* Facility ID */
  254 fixed bin(31) ) /* I_S_Info */
  options(assembler);
CALL CEETEST(ch, fb);
```

**Example 3**

This example assumes that you use predefined function prototypes and macros by including CEEIBMCT, 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 CEETEST call, the feedback code is checked for proper execution.

**Note:** The feedback code returned is either CEE000 or CEE2F2. There is no way to check the result of the execution of the command passed.

%INCLUDE CEEIBMCT;
%INCLUDE CEEIBMMAW;
DCL 01 FC FEEDBACK;

/* if CEEIBMCT is NOT included, the following DECLARES need to be provided: */

Declare CEEIBMCT Character(8) Based;
Declare ADDR BuiltIn;
%DCL FBCHECK ENTRY;
%FBCHECK: PROC( fbtoken, condition ) RETURNS( CHAR );
DECLARE
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:

```plaintext
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.

Notes:
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.

```plaintext
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.

```plaintext
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 PLTEST(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 LANGLEVEL(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 60.
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.

```c
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 GO command.

```c
#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 19. 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

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 16, “Starting Debug Tool from the Debug Tool Utilities,” on page 75.

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 ALL0C F1(INSPLOG) DA(*).
5. 
6. 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 sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 17, “Starting Debug Tool by using the TEST run-time option,” on page 79
- “Starting a debugging session in full-screen mode through a VTAM terminal” on page 111
- “Recording your debug session in a log file” on page 135
- Chapter 18, “Starting Debug Tool from a program,” on page 85

**Related references**
- [Debug Tool Reference and Messages](#)
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 tstscrpt with the TEST run-time option:
ALLOCATE FILE(insppref) DATASET(setup.pref) REUSE
ALLOCATE FILE(insplog) DATASET(session.log) REUSE
CALL 'USERID1.MYLIB(TSTSCRPT)'/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 progi with the TEST run-time option:
ALLOC FI(insplog) DA(debug.log) REUSE
ALLOC FI(insppref) DA(debug.preferen) REUSE
CALL 'MYID.MYQUAL LOAD(PROGI)' +
  ' TRAP(ON) TEST(*,*,*,insppref)/'

All the data sets must exist prior to starting this CLIST.

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 16, “Starting Debug Tool from the Debug Tool Utilities,” on page 75.

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 debug an MVS batch job in full-screen mode by choosing one of the following options:
- In full-screen mode through a VTAM terminal. Follow the instructions in “Starting a debugging session in full-screen mode through a VTAM terminal” on page 111.
- In remote debug mode. Follow the instructions in Appendix E, “Notes on debugging in remote debug mode,” on page 371.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
- Appendix D, “Notes on debugging in batch mode,” on page 369
- Chapter 31, “Entering Debug Tool commands,” on page 223

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.

```plaintext
//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,) TEST( )'
//                             1
//                             Include the Debug Tool SEQAMOD data set
//                             ****************************************
//STEPLIB DD DISP=SHR,DSN=userid.TEST.LOAD
//                             ****************************************
//                             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
```

Chapter 19. Starting Debug Tool for batch or TSO programs 97
/* use the log file as a commands file in subsequent Debug Tool sessions. You can specify the log file like:
   //INSPLOG DD SYSOUT=*,DCB=(LRECL=72,RECFM=FB,BLKSIZE=0)
   //SYSOUT DD SYSOUT=*
   //SYSPRINT DD SYSOUT=*
   //SYSUDUMP DD DUMMY */

Modifying the example to debug in full-screen mode: The example in “Example: JCL that runs Debug Tool in batch mode” on page 97 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 through a VTAM terminal without the Debug Tool Terminal Interface Manager, use the following statement:

  // PARM=’/TEST(,INSPIN,,MFI%TRMLU001:)

- To use full-screen mode through a VTAM terminal with the Debug Tool Terminal Interface Manager, use the following statement:

  // PARM=’/TEST(,INSPIN,,VTAM%USERABCD:)

Programs that start outside of Language Environment

The functions described in this section are available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

To debug an MVS batch or TSO program that has an initial program that does not run under the control of Language Environment, including OS/VS COBOL programs, use the Debug Tool program EQANMDBG to start Debug Tool.

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 “Programs that start in Language Environment” on page 95 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 99 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 100 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 100 for instructions.

- Verify that the Debug Tool SEQAMOD and SEQABMOD libraries are in the load module search path.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 16, “Starting Debug Tool from the Debug Tool Utilities,” on page 75

**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):
  - **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 through a VTAM terminal session.
  - **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.

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 100
- “Passing parameters to EQANMDBG using the PARM string and EQANMDBG DD statement” on page 100

Refer to the following sections for more information related to the material discussed in this section.

**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:

```
user_program_name
run-time_parm
```

Chapter 19. Starting Debug Tool for batch or TSO programs  99
The following table compares how a sample JCL statement might look like after you modify the PARM string:

<table>
<thead>
<tr>
<th>Original sample JCL</th>
<th>Modified sample JCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'</td>
<td>//STEP1 EXEC PGM=EQANMDBG,</td>
</tr>
<tr>
<td>...</td>
<td>// PARM='MYPROG,NATLANG(UEN)/ABC,X(12)'</td>
</tr>
<tr>
<td>//</td>
<td>//EQANMDBG DD *</td>
</tr>
<tr>
<td></td>
<td>MYPROG,</td>
</tr>
<tr>
<td></td>
<td>TEST(ALL,INSPIN,,MFI:*),</td>
</tr>
<tr>
<td></td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td></td>
<td>/*</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>//</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Original JCL</th>
<th>Modified JCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'</td>
<td>//STEP1 EXEC PGM=EQANMDBG,</td>
</tr>
<tr>
<td>...</td>
<td>// PARM='ABC,X(12)'</td>
</tr>
<tr>
<td>//EQANMDBG DD *</td>
<td>//EQANMDBG DD *</td>
</tr>
<tr>
<td>MYPROG,</td>
<td>MYPROG,</td>
</tr>
<tr>
<td>TEST(ALL,INSPIN,,MFI:*),</td>
<td>TEST(ALL,INSPIN,,MFI:*),</td>
</tr>
<tr>
<td>NATLANG(ENU)</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>/*</td>
<td>/*</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Original JCL</th>
<th>Modified JCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>//STEP1 EXEC PGM=MYPROG,PARM='ABC,X(12)'</code></td>
<td><code>//STEP1 EXEC PGM=EQANMDBG,</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>// PARM='MYPROG/ABC,X(12)'</code></td>
</tr>
<tr>
<td><code>//EQANMDBG DD *</code></td>
<td><code>//EQANMDBG DD *</code></td>
</tr>
<tr>
<td><code>*,TEST(ALL,INSPIN,,MFI:*)</code>,NATLANG(ENU)</td>
<td><code>*,TEST(ALL,INSPIN,,MFI:*)</code>,NATLANG(ENU)</td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>//</code></td>
<td><code>//</code></td>
</tr>
</tbody>
</table>
Chapter 20. Starting Debug Tool under CICS

After you decide what level of testing you want to employ during your debug session, you can start your program using the proper TEST run-time option for your language. If you are using Debug Tool, this requires no special procedures, although there are certain considerations depending on the environment where you are debugging your program. Before you begin your session, make sure all Debug Tool and program libraries are available and that all necessary Debug Tool files, such as the session log file, the primary commands file, the preferences file, and any desired USE files are defined and created. If the program you want to debug is authorized, ensure that the Debug Tool load library SEQAMOD is authorized and placed in the MVS LNKLST concatenation.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks

Choosing a debug mode

To use Debug Tool under CICS, you need to ensure 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.

You can start Debug Tool in one of the following ways:

Single terminal mode
Debug Tool displays its screens on the same terminal as the application. This can be set up using CADP, DTCN, CEETEST, pragma, or CEEUOPT(TEST).

Dual terminal mode
Debug Tool displays its screens on a different terminal than the one used by the application. This can be set up with CADP, DTCN or CEDF.

If you are using Debug Tool in a multiple-CICS region environment and sharing the EQADTCN2 temporary storage queue, set your profile to a debugging Display ID that is located in the same CICS region that the task you want to debug will run in.

Batch mode
Debug Tool does not have a terminal, but uses a commands file for input and writes output to the log. This can be set up using CADP, DTCN, CEETEST, pragma, or CEEUOPT(TEST).

Remote debug mode
Debug Tool works with a remote debugger to display results on a graphical user interface. This can be set up using CADP, DTCN, CEETEST, pragma, or CEEUOPT(TEST).

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
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** which is a full-screen CICS transaction that allows you to dynamically modify any Language Environment TEST or NOTEST run-time option with which your application was originally link-edited. You can also use DTCN to modify other Language Environment run-time options that are not specific to Debug Tool. See Chapter 13, “Preparing a CICS program,” on page 57 to learn how to set up profiles by using DTCN.

- **CADP** which is a CICS transaction that enables 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 over DTCN:
  - With CADP, multiple profiles with a single program name can be added from the same display device. There is no limit to the number of profiles supported. With DTCN, a single profile, with up to eight program ids, can be added from a single display device. In either case, the program names can be specified with wild cards.
  - CADP provides the same abilities as DTCN for managing debug profiles for Language Environment applications. CADP can also help 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 the CADP profile was present before a CICS region is restarted, the CADP profile will be present after the CICS region is restarted. For DTCN profiles, if the CICS region that owns the temporary storage queue where the debugging profiles were defined is restarted, the DTCN profiles must be added again after the 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.

- CICS CEDF utility where you can start a debug session in Dual Terminal mode alongside CEDF, using a special option on the CEDF command.
This mechanism does not require you to change the application link-edit options or code, so it can be useful if you need to debug programs that have been compiled with the TEST option, but do not have invocation mechanisms built into them.

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.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
- “Starting Debug Tool by using CEEUOPT” on page 106
- “Starting Debug Tool under CICS by using CEDF” on page 107

Related references
- “Debug modes under CICS” on page 311

Starting Debug Tool by using DTCN

DTCN is a menu-driven tool that allows you to create a profile that contains a pattern of CICS resource names that identify a task that you want to debug. When CICS programs are started, Debug Tool tries to match the executing resources to find a profile whose resources match those specified in a DTCN profile that you created. During this pattern-matching process, Debug Tool selects the best matching profile, which is the one with the greatest number of resource IDs that match the active task. If two tasks have an equal number of matching resource IDs, the older debug profile is selected.

To start Debug Tool by using DTCN:

1. Ensure that you have prepared your CICS program by following the instructions in Chapter 13, “Preparing a CICS program,” on page 57. This step includes creating the DTCN profiles that identify which task you want to debug and how you want to debug it; for example, specifying that you want a full-screen mode or remote debug mode session.

2. Run your CICS programs. If a task that matches the DTCN profile you created is started, Debug Tool is started.

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 EQADEXT 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 a CICS program is chosen for debugging

For example, consider the following two profiles:

- First, profile A is saved, specifying resource ID program PROG1
- Later, profile B is saved, specifying resource ID 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 resource IDs. In the above example, each profile should specify both a user ID and a program ID.
Refer to the following sections for more information related to the material discussed in this section.

Related tasks
Chapter 5, “Preparing a COBOL program,” on page 25

Starting Debug Tool by using 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 tried 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 resource IDs that match the active task.

When you start the CICS Transaction Server region and you set the DEBUGT00L system initialization parameter to YES, Debug Tool uses the CADP profile repository instead of the DTCN profile repository to find a matching debugging profile.

Before using CADP, verify that you have done all the preparation tasks described in Chapter 13, “Preparing a CICS program,” on page 57 and in the CICS books.

The CICS books provide a thorough description of how to use CADP.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
Chapter 13, “Preparing a CICS program,” on page 57

Related references
CICS Supplied Transactions

Starting Debug Tool 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 17, “Starting Debug Tool by using the TEST run-time option,” on page 29
Starting Debug Tool 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 sections for more information related to the material discussed in this section.

Related tasks
“Starting Debug Tool with CEETEST” on page 85

Starting Debug Tool under CICS by using CEDF

No specific preparation is required to use CEDF to start Debug Tool other than compiling the application with the appropriate compiler options and saving the source/listing.

CEDF has an ",I" option that starts Debug Tool. This option starts both EDF and Debug Tool in Dual Terminal mode. In Dual Terminal mode, EDF and Debug Tool screens are displayed on the terminal where you issue the CEDF command; application screens are displayed on the application terminal.

Note: You need to know the id of each terminal. One way to get this information is by using the CEBT transaction. The output will include Ter(xxxx), where xxxx is the terminal id.

To start Debug Tool, enter the CEDF transaction as follows:
CEDF xxxx,ON,I

where xxxx is the terminal on which you want to start the transaction to be debugged. This terminal is where the application is started. It performs 3270 application I/O, while a Debug Tool session is started at the terminal where CEDF is started.

CICS will return a message verifying the terminal id of the second terminal. Then, on the xxxx terminal, enter:
TRAN

where TRAN is the id for the transaction being debugged.

Once the command is entered, Debug Tool will be started for all Language Environment-enabled programs that are running on the terminal where Debug Tool is started. Debug Tool will continue to be active on this terminal, even if you turn off EDF.

For example, to begin a Debug Tool session using terminal T304 as the debugging terminal and T305 as the terminal where you want to run your application, start the CEDF transaction as follows on T304:
CEDF T305,ON,I

Then, on terminal T305, enter the name of the transaction you are debugging:
TRAN

When you run your application on T305, Debug Tool is started on T304. Terminal T305 displays only application output, that is, a specific CICS command to write to the screen.
Chapter 21. 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 "Programs that start in Language Environment" on page 95.

- For MVS batch programs that start in Language Environment, start your Language Environment program with the TEST run-time option and specify the appropriate suboptions, as described in "Starting Debug Tool in batch mode" on page 96.

- For MVS batch programs that do not start in Language Environment, including OS/VS COBOL programs, start the non-Language Environment Debug Tool (EQANMDBG), and pass your program name and the TEST run-time option. Specify the appropriate suboptions, as described in "Programs that start outside of Language Environment" on page 98.

- 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 OS/VS 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 OS/VS COBOL programs are defined in a DTCN or CADP debugging profile, Debug Tool will debug the non-Language Environment assembler or OS/VS 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 sections for more information related to the material discussed in this section.

Related tasks
- Chapter 16. “Starting Debug Tool from the Debug Tool Utilities,” on page 75
- Chapter 8. “Preparing a C program,” on page 37
- Chapter 9. “Preparing a C++ program,” on page 43
- Chapter 5. “Preparing a COBOL program,” on page 25
- Chapter 7. “Preparing a PL/I program,” on page 33
- “Ending a full-screen debug session” on page 153
- “Entering commands on the session panel” on page 125
- “Passing parameters to EQANMDBG” on page 99

Related references
- “Debug Tool session panel” on page 117
Chapter 22. Starting Debug Tool in other environments

You can start Debug Tool to debug batch programs in full-screen mode and from DB2 stored procedures.

Starting a debugging session in full-screen mode through a VTAM terminal

You can debug batch programs interactively by using a full-screen mode debugging session through a VTAM terminal. 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 VTAM terminal, which removes the need to update your runtime parameter string whenever the VTAM terminal LU name changes.

To start a debugging session in full-screen mode through a VTAM 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 VTAM 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 run time 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.

To start a debugging session in full-screen mode through a VTAM terminal with the Debug Tool Terminal Interface Manager, do the following steps:
1. Start two terminal emulator sessions. Connect the second emulator session to a terminal that can handle a full-screen mode debugging session through a VTAM terminal, 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 TM001 connected for user USER1
   EQAY001I Ready for Debug Tool

   PF3=EXIT PF12=LOGOFF

   The terminal is now ready to receive a Debug Tool full-screen mode through a VTAM terminal session.

4. Edit the PARM string of your batch job so that you specify the TEST run time 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.

**Starting Debug Tool from DB2 stored procedures: troubleshooting**

The examples used in this section are based on examples introduced in Chapter 12, “Preparing a DB2 stored procedures program,” on page 55.

If the remote debugger does not start when the stored procedure calls it, verify that the TCP/IP address is defined correctly in the DB2 catalog. When the stored procedure is called, the remote debugger source window is refreshed with the source or listing of the stored procedure.

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)
```

To switch from remote debug mode to full-screen mode through a VTAM terminal, change the definition of the stored procedure as follows, where `TRMLU001` is the VTAM LU name:

```
alter procedure SPROC1 run options 'TEST(,,,MFI%TRMLU001:)'```

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

Chapter 4, “Planning your debug session and collecting resources,” on page 21
Part 4. Debugging your programs in full-screen mode
Chapter 23. 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.

Note: The PF key definitions used in these topics are the default settings.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
Chapter 21, “Starting a full-screen debug session,” on page 109
“Ending a full-screen debug session” on page 153
“Entering commands on the session panel” on page 125
“Navigating through Debug Tool session panel windows” on page 130
“Recording your debug session in a log file” on page 135
“Setting breakpoints to halt your program at a line” on page 137
“Setting breakpoints in a load module that is not loaded or in a program that is not active” on page 137
“Stepping through or running your program” on page 137
“Displaying and monitoring the value of a variable” on page 144
“Displaying error numbers for messages in the Log window” on page 151
“Finding a renamed source, listing, or separate debug file” on page 151
“Requesting an attention interrupt during interactive sessions” on page 152
Chapter 27, “Debugging a C program in full-screen mode,” on page 181
Chapter 28, “Debugging a C++ program in full-screen mode,” on page 191
Chapter 24, “Debugging a COBOL program in full-screen mode,” on page 155
Chapter 26, “Debugging a PL/I program in full-screen mode,” on page 173

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 windows.

Source window
Displays your program source code

Log window
Records your commands and Debug Tool responses

Monitor window
Continuously displays the value of monitored variables and other items, depending on the command used

The Debug Tool session panel below shows the default layout for the Monitor window 1, the Source window 2, and the Log window 3.
Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- “Customizing the layout of windows on the session panel” on page 214

**Related references**
- “Session panel header”
- “Monitor window” on page 121
- “Source window” on page 120
- “Log window” on page 122

### 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.

Below is an example header for a C program.

```
C  1 LOCATION: MYID.SOURCE(TSTPGM1):>248
Command ====> 5 5 SCROLL ====> PAGE 4
```

Below is an example header for a COBOL program.

```
COBOL 1 LOCATION: XYZPROG::SUBR:>118
Command ====> 5 5 SCROLL ====> PAGE 4
```

The header fields are described below.

1. **Assemble, C, COBOL, OS COBOL, Disassem, or PL/I**
   - The name of the current programming language. This language is not necessarily the programming language of the code in the Source window. The language that is displayed in this field determines the syntax rules that you must follow for entering commands.
Notes:
1. Debug Tool does not differentiate between C and C++ programs. If there is a C++ program in the Source window, only C is displayed in this field.
2. OS COBOL is used to indicate OS/VS COBOL.

LOCATION
The program unit name and statement where execution is suspended, usually in the form compile unit:>:nnnnnn.

In the C example above, execution in MYID.SOURCE(TSTPGM1) is suspended at line 248.

In the COBOL example above, execution in XYZPROG is suspended at XYZPROG:::SUBR:>:118, or line 118 of subroutine SUBR.

If you are replaying recorded statements, the word "LOCATION" is replaced by PBK<LOC or PBK>LOC. The < and > symbols indicate whether the recorded statements are being replayed in the backward (<) or forward (>) direction.

If you are using the Enterprise PL/I compiler or the C/C++ compiler, the compile unit name is the entire data set name of the source. If the setting for LONGCUNAME is ON (the default) to display the CU name in long form, the name might be truncated. If your PL/I program was compiled with the following compiler and running in the following environment, the package statement or the name of the main procedure is displayed.

- Enterprise PL/I for z/OS, Version 3.5, compiler with the PTFs for APARs PK35230 and PK35489 applied
- Language Environment Version 1.4 through 1.8 with the PTF for APAR PK33738 applied

COMMAND
The input area for the next Debug Tool command. You can enter any valid Debug Tool command here.

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 7 lists all the scrolling commands.

Table 7. Scrolling commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Scroll by n number of lines.</td>
</tr>
<tr>
<td>HALF</td>
<td>Scroll by half a page.</td>
</tr>
<tr>
<td>PAGE</td>
<td>Scroll by a full page.</td>
</tr>
<tr>
<td>TOP</td>
<td>Scroll to the top of the data.</td>
</tr>
<tr>
<td>BOTTOM</td>
<td>Scroll to the bottom of the data.</td>
</tr>
<tr>
<td>MAX</td>
<td>Scroll to the limit of the data.</td>
</tr>
<tr>
<td>LEFT x</td>
<td>Scroll to the left by x number of characters.</td>
</tr>
<tr>
<td>RIGHT x</td>
<td>Scroll to the right by x number of characters.</td>
</tr>
</tbody>
</table>
Table 7. Scrolling commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR</td>
<td>Position of the cursor.</td>
</tr>
<tr>
<td>TO x</td>
<td>Scroll to line x, where x is an integer.</td>
</tr>
</tbody>
</table>

**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.                                                   |
 74 **********************                                    |
 75 DISPLAY "MULTCU COBOL SOURCE STARTED." UPON CONSOLE.      |
 76 MOVE 25 TO PROGRAM-USHORT-BIN.                            |
 77 MOVE -25 TO PROGRAM-SSHORT-BIN.                           |
 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 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.

When SET MONITOR COLUMN ON is in effect, the Monitor value area scale is visible under the header line for the monitor window. If SET MONITOR WRAP OFF is in effect, the Monitor value area column indicator appears on the left of the Monitor value area scale and shows the columns that are displayed. In this mode, if you scroll left or right, the monitor value area scale scrolls and the Monitor value area column indicator changes to indicate the columns that are displayed.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- “Using prefix commands on specific lines or statements” on page 127
- “Customizing profile settings” on page 217

**Monitor window**

<table>
<thead>
<tr>
<th>COBOL</th>
<th>LOCATION: DTAM01 -&gt; 109.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td><code>MONITOR</code></td>
</tr>
<tr>
<td></td>
<td><code>scroll</code></td>
</tr>
<tr>
<td></td>
<td><code>PAGE</code></td>
</tr>
</tbody>
</table>

The Monitor window has a Monitor value scale (1), which is the scale for the Monitor value area (2) below it. This area, where the values of the monitored variables are displayed, begins in column 30 and extends to the right, the full width of the displayable area of the monitor window. When you enter the command `SET MONITOR WRAP OFF` on the default screen, the value of the displayed columns in the Monitor value area is shown above the monitor variable names (3).

Use the Monitor window to continuously display output from the MONITOR LIST, MONITOR QUERY, MONITOR DESCRIBE, and SET AUTOMONITOR commands. If this window is not open, Debug Tool opens it when you enter a MONITOR or SET AUTOMONITOR command. Its contents are refreshed whenever Debug Tool receives control and after every Debug Tool command that can affect the display.

When you issue a MONITOR command, it is assigned a reference number between 1 and 99, then added to the monitor list. You can specify the monitor number; however, you must either replace an existing monitor number or use the next sequential number.

When you issue the SET AUTOMONITOR ON command, the following line is displayed at the bottom of the list of monitored variables:

```
********** AUTOMONITOR **********
```

Variables that are added to the Monitor window as a result of the SET AUTOMONITOR command are displayed underneath this line.
While the MONITOR command can generate an unlimited amount of output, bounded only by your storage capacity, the Monitor window can display a maximum of only 1000 scrollable lines of output.

If a window is not wide enough to show all the output it contains, you can either issue SCROLL RIGHT (to scroll the window to the right) or ZOOM (to make it fill the screen).

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.

You can update the values of monitored variables by typing new values over the displayed values.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- "Displaying and monitoring the value of a variable" on page 144
- "Scrolling the windows" on page 130

### Log window

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 (which is specified by using the TEST runtime option)
- 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.

```plaintext
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.

---

**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 used in the session panel.](#) “Saving customized settings in a preferences file” on page 219 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:
2. If you specify a preference file, looks for that preferences file and runs any commands in that preferences file.
3. If you specify a commands file, looks for that commands file and runs any commands in that 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.
Displaying the source

Debug Tool displays your source in the Source Window using a source, listing, or separate debug file, depending on the compiler.

When you start Debug Tool, if your source is not displayed, press PF4. This puts you in the Source Identification panel. The Source Identification panel indicates the name of the source, listing or separate debug file that Debug Tool intended to use. With this name, you can verify if the file exists or if you have authorization to access it. If your file is stored at a different location, do one of the following:

- Use the SET SOURCE command with the new name of the source, listing, or debug file.
- Use the SET DEFAULT LISTINGS command with the new name of the source, listing or debug file (provided they are stored in a PDS).
- Type over the Listing/Source file field in the Source Identification panel with the new name for the source, listing, or separate debug file.
- Use the EQADEBUG DD statement to define the location of the file.
- Code the EQAUEDAT user exit with the location.

If there is no debug data, you can display the disassembled code by entering the SET DISASSEMBLY command.

To be able to view your COBOL source code while debugging in full-screen or remote debug mode, you must direct the listing to a non-temporary file that is available during the debug session. If you are debugging a Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM program and specified the SEPARATE suboption of the TEST compiler option, the listing does not need to be saved, but the separate debug file must be a non-temporary file. The separate debug file must also be available during the debug session.

To be able to view your PL/I source while debugging in full-screen or remote debug mode, PL/I for MVS & VM and OS PL/I programs must be compiled using the PL/I SOURCE compiler option. You must also 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.

To be able to view the source of your Enterprise PL/I programs while debugging in full-screen or remote debug mode, you must direct the source to a non-temporary file that is available during the debug session. If you are debugging an Enterprise PL/I for z/OS Version 3 Release 5 programs that was compiled with the SEPARATE suboption of the TEST compiler option, you must direct the separate debug file to a non-temporary files that is available during the debug session.

To be able to view the source for your C and C++ programs while debugging in full-screen or remote debug mode, you must direct the source to a non-temporary file that is available during the debug session.

If your programs contain DB2 or CICS code, you might need to use a different file. See Chapter 11, “Preparing a DB2 program,” on page 51 or Chapter 13, “Preparing a CICS program,” on page 57 for more information.

If your source code is being managed by a library system that requires the SUBSYS= parameter when the data set is allocated, and you are debugging C, C++ or Enterprise PL/I compiled without the SEPARATE suboption of the TEST
compiler option, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=ssss allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 5, “Preparing a COBOL program,” on page 25
- Chapter 6, “Preparing an OS/VS COBOL program,” on page 29
- Chapter 7, “Preparing a PL/I program,” on page 33
- Chapter 8, “Preparing a C program,” on page 37
- Chapter 9, “Preparing a C++ program,” on page 43
- Chapter 10, “Preparing an assembler program,” on page 47
- Chapter 11, “Preparing a DB2 program,” on page 51
- Chapter 12, “Preparing a DB2 stored procedures program,” on page 55
- Chapter 13, “Preparing a CICS program,” on page 57
- Chapter 14, “Preparing an IMS program,” on page 65

**Related references**
- Appendix B, “How does Debug Tool locate debug information and source or listing files?,” on page 359
- Debug Tool Reference and Messages

### Entering commands on the session panel

You can enter a command or modify what is on the session panel in seven areas, as shown below.

<table>
<thead>
<tr>
<th>C LOCATION: MYID.SOURCE(ICFSSCU1)</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command ==</td>
<td>1</td>
</tr>
<tr>
<td>MONITOR</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1 VARBL1</td>
</tr>
<tr>
<td>0002</td>
<td>2 VARBL2</td>
</tr>
<tr>
<td>0003</td>
<td>STEP;</td>
</tr>
<tr>
<td>0004</td>
<td>AT 87;</td>
</tr>
<tr>
<td>0005</td>
<td>MONITOR</td>
</tr>
<tr>
<td>0006</td>
<td>LIST VARBL1;</td>
</tr>
<tr>
<td>0007</td>
<td>LIST VARBL2;</td>
</tr>
<tr>
<td>0008</td>
<td>STEP;</td>
</tr>
<tr>
<td>0009</td>
<td>STEP;</td>
</tr>
<tr>
<td>0010</td>
<td>STEP;</td>
</tr>
</tbody>
</table>

**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.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
"Using the session panel command line" on page 127
"Issuing system commands" on page 127
"Using prefix commands on specific lines or statements" on page 127
"Using commands that are sensitive to the cursor position" on page 128
"Using Program Function (PF) keys to enter commands" on page 128
"Retrieving previous commands" on page 129
"Retrieving commands from the Log and Source windows" on page 129

Related references
"Order in which Debug Tool accepts commands from the session panel" on page 129
"Initial PF key settings" on page 128

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

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 back slash (\) as a continuation character.

Debug Tool also provides automatic continuation if your command is not complete; for example, if the command was begun with a left brace (\{) that has not been matched by a right brace (\}). If you do need to continue your command, Debug Tool provides a **MORE ***>** prompt that is equivalent to another command line. You can continue to request additional command lines with continuation characters until you complete your command.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 31, “Entering Debug Tool commands,” on page 223

**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.

**TSO** is a synonym for the **SYSTEM** command. Truncation of the **TSO** command is not allowed.

**Using prefix commands on specific lines or statements**

Certain commands, known as **prefix commands**, can be typed over the prefix area in the Source or Monitor window, and then processed by pressing Enter. These commands (**AT**, **CLEAR**, **DISABLE**, **ENABLE**, **QUERY**, and **SHOW** in the Source window; **HEX**, **DEF**, **CL**, and **LIST** in the Monitor window) pertain only to the line or lines of code before which they are typed. For example, the **AT** command typed in the prefix area of a specific line sets a statement breakpoint only at that line.

You can use prefix commands to specify the particular verb or statement in the line where you want the command to apply. For example, **AT** typed in the prefix area of a line sets a statement breakpoint at the first relative statement in that line, while
AT 3 sets a statement breakpoint at the third relative statement in that line. Typing DISABLE 3 in the prefix area and pressing Enter disables that breakpoint.

**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 sections for more information related to the material discussed in this section.

- **Related tasks**
  - "Defining PF keys" on page 213

**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 sections for more information related to the material discussed in this section.

- **Related tasks**
  - "Defining PF keys" on page 213
  - "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.

<table>
<thead>
<tr>
<th>PF key</th>
<th>Label</th>
<th>Definition</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF1</td>
<td>?</td>
<td>?</td>
<td>&quot;Getting online help for Debug Tool command syntax&quot; on page 227</td>
</tr>
<tr>
<td>PF2</td>
<td>STEP</td>
<td>STEP</td>
<td>&quot;Stepping through or running your program&quot; on page 137</td>
</tr>
<tr>
<td>PF3</td>
<td>QUIT</td>
<td>QUIT</td>
<td>&quot;Ending a full-screen debug session&quot; on page 153</td>
</tr>
<tr>
<td>PF4</td>
<td>LIST</td>
<td>LIST</td>
<td>&quot;Finding a renamed source, listing, or separate debug file&quot; on page 151</td>
</tr>
<tr>
<td>PF4</td>
<td>LIST</td>
<td>LIST variable_name</td>
<td>&quot;Displaying and monitoring the value of a variable&quot; on page 144</td>
</tr>
</tbody>
</table>
### PF key | Label | Definition | Use
---|---|---|---
PF5 | FIND | IMMEDIATE FIND | “Finding a string in a window” on page 131
PF6 | AT/CLEAR | AT TOGGLE CURSOR | “Setting breakpoints to halt your program at a line” on page 137
PF7 | UP | IMMEDIATE UP | “Scrolling the windows” on page 130
PF8 | DOWN | IMMEDIATE DOWN | “Scrolling the windows” on page 130
PF9 | GO | G0 | “Stepping through or running your program” on page 137
PF10 | ZOOM | IMMEDIATE ZOOM | “Zooming a window to occupy the whole screen” on page 216
PF11 | ZOOM LOG | IMMEDIATE ZOOM LOG | “Zooming a window to occupy the whole screen” on page 216
PF12 | RETRIEVE | IMMEDIATE RETRIEVE | “Retrieving previous commands”

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

“Defining PF keys” on page 213

### 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 sections for more information related to the material discussed in this section.

**Related tasks**

“Retrieving commands from the Log and Source windows”

### Retrieving commands from the Log and Source windows

You can retrieve lines from the Log and Source windows and use them as new commands.

To retrieve a line, move the cursor to the desired line, modify it (for example, delete any comment characters) and press Enter. The input line appears on the command line. You can further modify the command, then press Enter to issue it.

When retrieving long or multiple Debug Tool commands, a pop-up window is displayed, with the command as typed in so far. However, trailing blanks on the last line are removed. To expand the pop-up window, place the cursor below it and press Enter.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

“Retrieving previous commands”
Navigating through Debug Tool session panel 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, 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 sections for more information related to the material discussed in this section.

Related tasks
- “Moving the cursor between windows”
- “Scrolling the windows”
- “Scrolling to a particular line number” on page 131
- “Finding a string in a window” on page 131
- “Changing which source file appears in the Source window” on page 133
- “Displaying the line at which execution halted” on page 134
- “Customizing profile settings” on page 217

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 sections for more information related to the material discussed in this section.

Related tasks
- “Defining PF keys” on page 213

Scrolling the windows

When the monitor window is in WRAP OFF mode, you can scroll the monitor value area LEFT and RIGHT with scroll commands typed on the command line and entered while the cursor is in the Monitor value area. For example, type, SCROLL RIGHT 10, place the cursor in the Monitor value area, and press Enter. The Monitor value area shifts right by 10 characters. The monitor value area is only scrollable after you enter the command SET MONITOR WRAP OFF.

If the cursor is on the command line, you can scroll the Source window by pressing PF7 (UP) or PF8 (DOWN). To scroll through other windows, place the cursor in the desired window before pressing PF7 or PF8.

You can toggle one of the Source, Log or Monitor windows to full screen (temporarily not displaying the others) by moving the cursor into the window you
want to zoom and pressing PF10 (ZOOM). To toggle back, press PF10 again. PF11 (ZOOM LOG) toggles the Log window the same way without the cursor needing to be in the Log window.

You can scroll any of the windows vertically and horizontally by issuing the SCROLL UP, DOWN, LEFT, and RIGHT commands (the SCROLL keyword is optional). You can use the command line to specify which window to scroll. For example, to scroll the monitor window up 5 lines, enter SCROLL UP 5 MONITOR.

Alternately, you can use the position of the cursor to indicate the window you want to scroll; if the cursor is in a window, that window is scrolled. If you do not specify the window, the default window (determined by the setting of the DEFAULT WINDOW command) is scrolled. You can change the default window by changing the settings of Default window and Default scroll amount under the Profile Settings panel.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- “Customizing the layout of windows on the session panel” on page 214
- “Scrolling to a particular line number” on page 132
- “Customizing profile settings” on page 217

**Scrolling to a particular line number**

To display a particular line at the top of a window, use the SCROLL TO command with the statement numbers shown in the window prefix areas. Enter 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 SCROLL TO 345 on the command line. The selected window is scrolled vertically so that your specified line is displayed at the top of that window.

**Finding a string in a window**

You can search for strings in the Source, Monitor, or Log window and you can search either forward or backward. 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.

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 in “Syntax of a search string” on page 132.
2. Press Enter.

If you want to repeat the previous search, hit the PF5 key.

Refer to the following sections for more information related to the material discussed in this section.

**Related concepts**

- “How does Debug Tool search for strings?” on page 132

**Related tasks**

**Related references**
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 single or double quotes:
- spaces
- an asterisk ("*"
- a question mark ("?")
- a semicolon (";")

Use the following rules to determine whether to use single or double quotes:
- If you are debugging an assembler, C, C++, COBOL, or disassembly program, the string must be enclosed in double quotes.
- If you are debugging an assembler, disassembly, or PL/I program, the string must be enclosed in single quotes.

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 when SET MONITOR WRAP OFF is in effect
Type the FIND command with the string, and place the cursor in the monitor window. Debug Tool will search 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 ;

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;
Changing which source file appears in the Source window

To change which source file appears in the Source window, type over the name after \texttt{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 (CU) that is known to Debug Tool. To determine which CUs are known to Debug Tool, enter the \texttt{LIST NAMES CUS} command. By default, the \texttt{LIST NAMES CUS} command does not display the names of disassembled CUs. If you are debugging an assembler or disassembly program and want the \texttt{LIST NAMES CUS} to display the names of the disassembled CUs, enter the \texttt{SET DISASSEMBLY ON} or \texttt{SET ASSEMBLER ON} command before you enter the \texttt{LIST NAMES CUS} command.

Alternately, you can enter the command:

\texttt{LIST NAMES CUS}

and a list of compile units will be written to the Log window, as shown below.

\begin{verbatim}
USERID.MFISTART.C(CALC)
USERID.MFISTART.C(PUSHPOP)
USERID.MFISTART.C(READTOKN)
\end{verbatim}

You can type over or insert characters on one of these lines in the Log window and press Enter to display the modified text on the command line, for example:

\texttt{SET QUALIFY CU "USERID.MFISTART.C(READTOKN)"}

and then press Enter to issue the command. Typing over a line in the Log window and issuing them as commands is a way to save keystrokes and reduce errors in long commands.

Another way to change which source file appears in the Source window is to press PF4 (LIST) with the cursor on the command line. This displays the Source Identification Panel, where associations are made between listings or source files shown in the Source window and their compile units. Type over the Listings/Source File field with the new name.

For C and C++ only

For C and C++ compile units, Debug Tool requires a file containing the source code. By default, when Debug Tool encounters a new C and C++ compile unit, it looks for the source code in a file whose name is the one that was used in the compile step.

If your source code is managed by a library system that requires the \texttt{SUBSYS=SSSSS} parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the \texttt{SUBSYS=SSSSS} allocation parameter. This support is not available when debugging a program under CICS. See the \textit{Debug Tool Customization Guide} for details.

For COBOL only

For COBOL compile units, Debug Tool looks for information in one of the following ways:

- For VS COBOL II, Debug Tool looks for the listing in a file named \texttt{userid.cuname.LIST}.
- For COBOL/370 and COBOL for MVS & VM, Debug Tool looks for the listing in the data set specified during the compile step.
- For Enterprise COBOL for z/OS and OS/390 and COBOL for OS/390 & VM, Debug Tool looks for compiler listing in one of the following locations:
- Debug Tool looks for the listing in the data set specified during the compile step.
- If your program is compiled with the SEPARATE suboption of the TEST compiler option, Debug Tool looks for the compiler listing in the separate debug file specified during the compile step.

**Related tasks**
- "Finding a renamed source, listing, or separate debug file" on page 151

**For PL/I only**
For PL/I compile units, Debug Tool looks for information in one of the following ways:
- For non-Enterprise PL/I compile unit, Debug Tool looks for the listing in a file named userid.cuname.LIST.
- For Enterprise PL/I prior to Version 3 Release 5, Debug Tool looks for the source in the data set specified during the compile step.
- For Enterprise PL/I Version 3 Release 5 or later, Debug Tool looks for the source in one of the following locations:
  - Debug Tool looks for the source in the data set specified during the compile step.
  - If your program is compiled with the SEPARATE suboption of the TEST compiler option, Debug Tool looks for the source in the separate debug file specified during the compile step.

If Debug Tool needs to access your Enterprise PL/I source code and this code is managed by a library system that requires the SUBSYS=ssss parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=ssss allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

**Related tasks**
- "Finding a renamed source, listing, or separate debug file" on page 151

**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 one of the following commands:
- **SET QUALIFY RESET**
- **Q LOC**
- **LIST %LINE**

**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.

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.

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 and C++ or PL/I 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 (intercepted console messages, exceptions, etc.)

The default ddname associated with the Debug Tool log file is INSPLOG. If you do not allocate a file with ddname INSPLOG, no default log file is created.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
- “Creating the log file”
- “Saving and restoring settings, breakpoints, and monitor specifications” on page 141

Creating the log file

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 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. Allocate the INSPLOG file if you want to keep a permanent record of the results.
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 sections for more information related to the material discussed in this section.

**Related tasks**

*Creating the log file” on page 135*
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 sections for more information related to the material discussed in this section.

Related tasks

"Halting on a line in C only if a condition is true” on page 185
"Halting on a line in C++ only if a condition is true” on page 196
"Halting on a COBOL line only if a condition is true” on page 160
"Halting on a PL/I line only if a condition is true” on page 177

Setting breakpoints in a load module that is not loaded or in a program that is not active

If you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1(5655-R45), 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.

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).
To run your program up to the next hook, press PF2 (STEP). If you compiled with 
TEST for C or C++, TEST(ALL,SYM) for COBOL or PL/I, or TEST(NONE,SYM) for 
Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM with the 
Dynamic Debug facility activated, STEP performs one statement.

To run your program until a breakpoint is reached, the program ends, or a 
condition is raised, press PF9 (60).

**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 sections for more information related to the material 
discussed in this section.

**Related tasks**
- Chapter 4, “Planning your debug session and collecting resources,” on page 21
- Chapter 17, “Starting Debug Tool by using the TEST run-time option,” on page 79

### Recording and replaying statements

The commands described in this section are available only if you purchase and 
install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 
(5655-R45).

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 RUNT0 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.

### 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 and are running with the following Language Environment run time and APARs installed:

- Compilers:
  - Enterprise COBOL for z/OS and OS/390, Version 3 Release 2
  - 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
- Language Environment APARs:
  - z/OS Version 1 Release 4, with APAR PQ65176

Refer to the following sections for more information related to the material discussed in this section.

Related tasks

“Stop the recording” on page 140

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.

**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.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

[Debug Tool Reference and Messages](#)

**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 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 from 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 LOADDDEBUGDATA (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 DBGTOOL SAVESETS (a sequential data set) is used to save the settings.
• userid DBGTOOL SAVEBPS (a PDS or PDSE data set) is used to save the breakpoints, monitor specifications, and LDD specifications.

In non-interactive mode (MVS batch mode without using full-screen mode through a VTAM terminal), 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
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 through a VTAM terminal), 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 through a VTAM terminal), you must include an INSPBPM DD statement that references this data set.
   • 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 through a VTAM terminal), you must add INSPSAFE and INSPBPM DD statements that reference the data sets you allocated in step 1 on page 142 and 2 on page 142.

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 of automatic saving and restoring
To disable automatic saving of breakpoints and monitors, you must enter one or both of the following commands:
• SET SAVE BPS NOAUTO;
• SET SAVE MONITORS NOAUTO;

To disable automatic saving of settings, you must enter the command SET SAVE SETTINGS NOAUTO;

To disable automatic restoring of breakpoints and monitors, you must enter one or both of the following commands:
• SET RESTORE BPS NOAUTO;
• SET RESTORE MONITORS NOAUTO;

To disable automatic restoring of settings, you must enter the command SET RESTORE SETTINGS NOAUTO;

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
• Debugging in an IMS/DC environment

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.
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;
   ```

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 (INSPPREF).

---

**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 or the PF4 key. One-time display displays the value of the variable at the moment you enter the LIST command or press the PF4 key. If you step or run through your program, any changes to the value of the variable are not displayed.

- **Continuous display**, called monitoring, by using the MONITOR LIST command or the SET AUTOMONITOR 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](https://example.com) for more information about this command.

If Debug Tool cannot display the value of a variable in its declared data type, Debug Tool displays the value of the variable in hexadecimal format. To change the value of the variable, type in the new value in hexadecimal format.

**One-time display of the value of variables**

To change the format of the output for arrays and structures to tabular format when displaying a variable:

1. Move the cursor to the command line.
2. Enter the following command: `SET LIST TABULAR ON`

To change the format of the output for arrays and structures to linear format when displaying a variable:

1. Move the cursor to the command line.
2. Enter the following command: `SET LIST TABULAR OFF`
The SET LIST TABULAR command is available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

To format the logged output of arrays and structures when SET AUTOMONITOR ON LOG is in effect:
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 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.

To display the contents of a variable once 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.

**Monitoring the value and datatype of variables**

To monitor the value of a variable by using 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. The variable variable-name is added to the Monitor window and the current value of variable-name is displayed. As you step through your program, the value of variable-name is updated in the Monitor window so that the window always displays the current value of variable-name.

To monitor the value of a variable by using the SET AUTOMONITOR ON command, do the following steps:
1. Move the cursor to the command line.
2. Enter the following command:
   SET AUTOMONITOR ON;

   Debug Tool opens a separate section of the Monitor window, called the automonitor section, and displays the name and value of variables in the next statement that you run. Each time you enter the STEP command or a breakpoint is encountered, Debug Tool does the following:
   a. Removes the previous names and values.
   b. 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.

The command SET MONITOR DATATYPE ON displays the datatype that is associated with variables that are monitored through the MONITOR LIST command and
automonitor. The datatype is ordinarily that which appeared in the declaration of the variable. The command SET MONITOR DATATYPE OFF disables this function.

To monitor the value and datatype of a variable by using MONITOR LIST command:
1. Move the cursor to the command line.
2. Enter the following command:
   SET MONITOR DATATYPE ON
3. Enter the following command, substituting your variable name for
   variable-name:
   MONITOR LIST variable-name

   The variable is added to the Monitor window and the current value of the variable and its datatype are displayed.

The SET MONITOR DATATYPE command is available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

You can also monitor the value and datatype of a variable by using the SET AUTOMONITOR ON command.

To monitor the value and datatype of a variable by using the SET AUTOMONITOR ON command:
1. Move the cursor to the command line.
2. Enter the following command:
   SET MONITOR DATATYPE ON
3. Enter the following command:
   SET AUTOMONITOR ON

   The names, values and datatypes of the variables in the current line are displayed in the automonitor area of the monitor window. If LOG was specified for automonitor, the values and datatypes of the variables in the current line are also displayed in the Debug Tool log.

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.

Variables that are monitored through the MONITOR LIST command are displayed above the automonitor section of the Monitor window.

To close the automonitor section of the Monitor window and stop adding the name and value of variables in the statements your run, enter the SET AUTOMONITOR OFF command.
The SET AUTOMONITOR command is available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

**Example: SET AUTOMONITOR ON command**
The example in this section assumes that the following two lines of COBOL code are to be run:

```cobol
COMPUTE LOAN-AMOUNT = FUNCTION NUMVAL(LOAN-AMOUNT-IN).
COMPUTE INTEREST-RATE = FUNCTION NUMVAL(INTEREST-RATE-IN).
```

Before you run the statement in Line 1, enter the following command:

```cobol
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.

**Formatting values in the Monitor window**
To monitor the value of the variable in columnar format, use the SET MONITOR COLUMN ON command. The variable names that are displayed in the Monitor window are aligned in the same column and values are aligned in the same column.

To display the value of the monitored or automonitored variable, either wrapped in the monitor window or on a single scrollable line, use the SET MONITOR WRAP command. The SET MONITOR WRAP OFF command enables the variable name and value to be displayed on the same line in the monitor window with the value field being scrollable. SET MONITOR WRAP OFF is valid only if the command SET MONITOR COLUMN ON is in effect.

The SET MONITOR WRAP command is available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

**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 it's declared data type, follow the instructions described in "Monitoring the value and datatype of variables" on page 145. For PL/I programs, use the PL/I %HEX built-in function to monitor the value of a variable in hexadecimal format. If you monitor 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 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:
  ```plaintext
  MONITOR n HEX ;
  ```
  Substitute n with the number in the monitor list the corresponds to the monitored expression that you would like to display in hex.

- If the variable is not being monitored, enter the following command:
  ```plaintext
  MONITOR LIST (expression) HEX ;
  ```
  Substitute expression with the name of the variable or a complex expression you want to monitor.

**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 OS/VS 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*.

**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:
   • Only one value at a time can be modified. If you type over the values of more
     than one variable, only the first one is modified.
   • 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
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.
   • You can not 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 modify a long value and the setting of MONITOR WRAP is OFF, Debug
Tool creates the STORAGe command to modify the value. If you are debugging a
program that is optimized, the STORAGe command does not necessarily alter the
value that is used by the program.

   If you type quotation marks 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.

Refer to the following sections for more information related to the material
discussed in this section.

Related tasks
“Displaying values of COBOL variables” on page 232
“Displaying values of C and C++ variables or expressions” on page 258
“Accessing PL/I program variables” on page 251
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. These commands are available only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

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
  * SF5800001/SF590000 EQAZSTEP BCARTER.TEST.LOAD
  * SF5800002/SF590000 SHARE.CEE210.SCEERUN
  * VIO NEW DELETE * INSPLLOG BCARTER.DTOOL.LOGV
  * SF5800003/SF590000 ISPEXEC BCARTER.MVS.EXEC
  * IPLB13 * SHR KEEP ISPF.SISPEXEC.VB
  * VIO NEW DELETE ISPLST1 SYS02190.T085429.RA000.BCARTER.R0100269
  * IPLB13 * SHR KEEP ISPF.SISPMENU
  * SMS278 * SHR KEEP SHARE.ANALYZ21.SIDIMLIB
  * SHARE89A * SHR KEEP SHARE.ISPMLIB
  * SMS25F * SHR KEEP SHARE.PROD.ISPPLIB
  * SMS091 * SHR KEEP SHARE.ISPPLIB
  * SMS25F * SHR KEEP SHARE.ANALYZ21.5IPLIB
  * IPLB13 * SHR KEEP ISPF.ISPPENU
  * IPLB13 * SHR KEEP ISPF.ISFPPLIB
  * COD002 * OLD KEEP * ISPPROF BCARTER.ISPPROF
  * NEW DELETE SYSPRINT 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 sections for more information related to the material discussed in this section.

**Related tasks**

[“Customizing profile settings” on page 217](#)

### Finding a renamed source, listing, or separate debug file

If the source, listing, or separate debug file has been renamed since your program was compiled, Debug Tool will not be able to find it, and it will not appear in the Source window when you debug your program.

To point Debug Tool to the renamed file:

- 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, listings, or separate debug files that show in the Source window with their compile units.
     
     2. Type over the **Listing/Source File** field with the new name.
  
- Use the SET SOURCE or SET DEFAULT LISTINGS commands to direct Debug Tool to the new files:
  1. With the cursor on the command line, type SET SOURCE ON (cuname)
     
     new_file_name, where new_file_name is the renamed source file. Press Enter.
  2. 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.CRTTEST.PROGR, enter the following command:

```bash
SET DEFAULT LISTINGS (SVTRSAMP.TS99992.MYPROG, PGRSAMP.LLTEST.PROGA, RRSAMP.CRTTEST.PROGR) ;
```
If you need to do this repeatedly, use one of the following methods:

- Note 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.
- Use the EQAD debug DD statement to define the location of the files.
- Code the EQAUEDAT user exit with the location of the files.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- "Changing which source file appears in the Source window" on page 133

### 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 through a VTAM terminal

**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 sections for more information related to the material discussed in this section.

**Related tasks**
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 OS/VS COBOL program, QUIT ends Debug Tool and the task ends with an ABEND 4038.

**Method B**

1. Enter the QUIT 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 OS/VS 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. Your program continues to run.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

*Debug Tool Reference and Messages*
Chapter 24. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 32, “Debugging COBOL program,” on page 229
“Halting when certain routines are called in COBOL” on page 158
“Modifying the value of a COBOL variable” on page 159
“Halting on a COBOL line only if a condition is true” on page 160
“Debugging COBOL when only a few parts are compiled with TEST” on page 161
“Capturing COBOL I/O to the system console” on page 161
“Displaying raw storage in COBOL” on page 162
“Getting a COBOL routine traceback” on page 162
“Halting on a COBOL line only if a condition is true” on page 163
“Running the run-time path for COBOL code compiled with TEST” on page 162
“Generating a COBOL run-time paragraph trace” on page 163
“Finding unexpected storage overwrite errors in COBOL” on page 164
“Halting before calling an invalid program in COBOL” on page 164

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. Several COBOL intrinsic functions are utilized.

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".

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) 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 sections for more information related to the material discussed in this section.

**Related tasks**

[Chapter 24, “Debugging a COBOL program in full-screen mode,” on page 155]

---

**Halting when certain routines are called in COBOL**

[“Example: sample COBOL program for debugging” on page 155]

To halt just before COBLOAN is called, issue the command:

AT CALL COBLOAN ;

If the CU COBVALU is known to Debug Tool (that is, it has been called previously), to halt just after COBVALU is called, issue the command:

AT ENTRY COBVALU ;

If the CU COBVALU is not known to Debug Tool (that is, it has not been called previously), to halt just before COBVALU is entered the first time, issue the command:

AT APPEARANCE COBVALU ;

You can display a list of all compile units that are known to Debug Tool by entering the command:

LIST NAMES CUS ;

The Debug Tool Log window displays something similar to:
LIST NAMES CUS;
The following CUs are known in *:
COBCALC
COBLOAN
COBVALU

Additionally, you can combine the breakpoints as follows:
AT APPEARANCE COBVALU AT ENTRY COBVALU ; GO ;

The purpose for the appearance breakpoint is to gain control the first time the COBVALU compile unit is run.

To take advantage of either AT ENTRY or AT APPEARANCE, you must compile the routine program (COBVALU in the above example) with the TEST compiler option.

If you have many breakpoints set in your program, you can issue the command:
QUERY LOCATION

To indicate where in your program execution has been interrupted. The Debug Tool Log window displays something similar to:
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 155

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 \texttt{CALC1}, and enter AT 46 ; GO ; on the Debug Tool command line. Move the cursor over INPUT-1 and press LIST (PF4). The following appears in the Log window:

\begin{verbatim}
LIST ( INPUT-1 ) ;
INPUT-1 = 'LOAN' ;
\end{verbatim}

To modify the value of INPUT-1, enter on the command line:

\begin{verbatim}
MOVE 'pvalue' to INPUT-1 ;
\end{verbatim}

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 \texttt{VALU2} is reached. To view the attributes of the variable INTEREST, issue the Debug Tool command:

\begin{verbatim}
DESCRIBE ATTRIBUTES INTEREST ;
\end{verbatim}

The result in the Log window is:

\begin{verbatim}
ATTRIBUTES FOR INTEREST
IT'S LENGTH IS 4
IT'S ADDRESS IS 00011DC8
02 COBVALU:INTEREST S999V99 COMP
\end{verbatim}
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 sections for more information related to the material discussed in this section.

Related tasks
“Using COBOL variables with Debug Tool” on page 231

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 60.

“Example: sample COBOL program for debugging” on page 155

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 ; 60 ;
CLEAR AT 67 ; STEP 4 ;

Now set the breakpoint like this:
AT 44 IF INTEREST > -1 THEN 60 ; 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 60 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

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 sections for more information related to the material discussed in this section.

**Related tasks**

- [Halting when certain routines are called in COBOL](#) on page 158

### 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 ;
```

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 : IGZ00001 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)

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 155

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;
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.

```
* 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;
```

Assume you have a PDS member, USERID.DT.COMMANDS(COBCALC2), that contains the following Debug Tool commands.

```
AT EXIT * PERFORM;
  LIST ( "Exit:", LEVEL);
  COMPUTE LEVEL = LEVEL - 1;
  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:
### 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.

```cobol
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 25. Debugging an OS/VS COBOL program in full-screen mode

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

"Example: sample OS/VS 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. Please read the information regarding non-Language Environment programs for instructions on how to start Debug Tool and debug OS/VS COBOL programs, unless OS/VS COBOL-specific information is provided.

Example: sample OS/VS COBOL 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. Prepare the sample program as described in Chapter 6, “Preparing an OS/VS COBOL program,” on page 29.
2. Verify that the debug information for this program is located in the COB03O and COB03AO members of the yourid.EQALANGX data set.
3. Start Debug Tool as described in "Programs that start outside of Language Environment" on page 98.
4. To load the debug information for this program, enter the following command:
   LDD (COB03O,COB03AO);

This program is a small example of an OS/VS COBOL program (COB03O) that calls another OS/VS COBOL program (COB03AO).

Load module: COB03O

COB03O

******************************************************************************
* PROGRAM NAME: COB03O                                          *
*                                                                     *
* COMPILED WITH IBM OS/VS COBOL COMPILER                         *
******************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID. COB03O.
******************************************************************************
* LICENSED MATERIALS - PROPERTY OF IBM                            *
*                                                               *
* 5655-P14: Debug Tool for z/OS                                  *
* 5655-P15: Debug Tool Utilities and Advanced Functions 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-DAY REDEFINES JULIAN-DATE.
     05 J-YEAR       PIC 9(4).
     05 J-DAY       PIC 9(3).

* SAVE DATE
  01 SAVE-DATE     PIC 9(7).

PROCEDURE DIVISION.

PROG.
  ACCEPT JULIAN-DATE FROM DAY
  DISPLAY 'JULIAN-DATE: ' JULIAN-DATE
  MOVE JULIAN-DATE TO SAVE-DATE
  MOVE 10000 TO LOAN
  CALL 'COB03AO' USING LOAN INTEREST-DUE.
  DISPLAY 'LOAN: ' LOAN
  DISPLAY 'INTEREST-DUE: ' INTEREST-DUE
  STOP RUN.

COB03AO

******************************************************************************
******************************************************************************
* PROGRAM NAME: COB03AO
* *
* COMPILED WITH IBM OS/VS COBOL COMPILER
* *
******************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID. COB03AO.
******************************************************************************

* LICENSED MATERIALS - PROPERTY OF IBM
* *
* 5655-P14: Debug Tool for z/OS
* 5655-P15: Debug Tool Utilities and Advanced Functions 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.
* *
******************************************************************************
Defining a compilation unit as OS/VS COBOL and loading debug information

Before you can debug an OS/VS COBOL program, you must define the compilation unit (CU) as an OS/VS 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 an OS/VS 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 an OS/VS 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: 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 OS/VS 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 an OS/VS 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 OS/VS COBOL**

You must use the LDD command to identify a CU as an OS/VS 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 sections for more information related to the material discussed in this section.

**Related tasks**
- “Defining a compilation unit as OS/VS COBOL and loading debug information” on page 169

**Halting when certain OS/VS COBOL programs are called**

“Example: sample OS/VS COBOL program for debugging” on page 167

To halt after the COB03AO routine is called, enter the following command:

```
AT ENTRY COB03AO ;
```

The AT CALL command is not supported for OS/VS COBOL routines. Do not use the AT CALL command to halt Debug Tool when an OS/VS COBOL routine is called.

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 OS/VS 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; 60; 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 9 (999999)
```

To step into the call to COB03AO, press PF2 (STEP).

---

**Halting on a line in OS/VS COBOL only if a condition is true**

Often a particular part of your program works fine for the first few thousand times, but it fails under certain conditions. Setting a line breakpoint is inefficient because you will have to repeatedly enter the GO command.

"Example: sample OS/VS COBOL program for debugging" on page 167

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 GO; 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 OS/VS COBOL when debug information is only available for a few parts**

"Example: sample OS/VS COBOL program for debugging" on page 167

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 an OS/VS 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 OS/VS COBOL program for debugging” on page 167

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 OS/VS COBOL program COB030 ::> COB03AO.
From LINE 74 in OS/VS COBOL program COB030 ::> COB030.

Finding unexpected storage overwrite errors in OS/VS 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 an OS/VS COBOL program traceback” on page 171, 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 26. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 34, “Debugging PL/I programs,” on page 247
"Halting when certain PL/I functions are called” on page 176
"Modifying the value of a PL/I variable” on page 176
"Halting on a PL/I line only if a condition is true” on page 177
"Debugging PL/I when only a few parts are compiled with TEST” on page 177
"Displaying raw storage in PL/I” on page 178
"Getting a PL/I function traceback” on page 178
"Tracing the run-time path for PL/I code compiled with TEST” on page 179
"Finding unexpected storage overwrite errors in PL/I” on page 180
"Halting before calling an undefined program in PL/I” on page 180

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
DECLALC: 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;
/*
* 1 stack,
* 2 stkptr fixed bin(15,0) init(0),
* 2 stknum(50) fixed bin(31,0);
* 1 bufin,
* 2 bufptr fixed bin(15,0) init(0),
* 2 bufchr char (100) varying;
* 1 tok char (100) varying;
* 1 tstop char(1) init ('s');
* 1 mdx 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);
pull list ('PLICALC: ', num) skip;
call push(stack,num);
end;
otherwise do; /* must be an integer */
num = atoi(tok);
call push(stack,num);
end;
end;
end;
return;

TOK function
atoi: procedure(tok) returns (fixed bin(31,0));
/*------------------------------------------------------------------*/
/* convert character string to number */
/* (note: string validated by readtok) */

dcl 1 tok char (100) varying;
dcl 1 num fixed bin (31,0);
dcl 1 j fixed bin(15,0);
num = 0;
do j = 1 to length(tok);
   num = (10 * num) + (index('0123456789',substr(tok,j,1))-1);
end;
return (num);
end atoi;
end plicalc;
PUSH function

push: procedure(stack,num);
/*-------------------------------------*/
/**/
/* a simple push function for a stack of integers */
/**/
/*-------------------------------------*/
dcl 1 stack connected,
  2 stkptr fixed bin(15,0),
  2 stknum(50) fixed bin(31,0);
dcl num fixed bin(31,0);
stkptr = stkptr + 1;
stknum(stkptr) = num; /* PUSH 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;
tok = substr(bufchr,bufptr,1); /* get ready to return single char */
select (tok);
  when ('+', '-', '/', 'x', '=')
    bufptr = bufptr;
otherwise do; /* possibly an integer */
    tok = '1';
    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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 26, “Debugging a PL/I program in full-screen mode,” on page 173

Halting when certain PL/I functions are called

“Example: sample PL/I program for debugging” on page 173

To halt just before READTOK is called, issue the command:

AT CALL READTOK;

To halt just after READTOK is called, issue the command:

AT ENTRY READTOK;

To take advantage of the AT ENTRY command, you must compile your program with the TEST option.

If you have many breakpoints set in your program, you can issue the command:

QUERY LOCATION

to indicate where in your program execution has been interrupted. The Debug Tool Log window displays something similar to:

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 PUSH 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 173
```

For example, in PLICALC you want to stop at the division selection only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 31 DO; IF NUM ^= 0 THEN GO; END;
```

Line 31 is the statement labeled CALC2. The command causes 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 173
```
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)
```

### 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 173

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 2 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:
   
   ```pli
   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;
   ```


4. Enter the following command:

   ```pli
   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:

```pli
*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;
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 27. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 35, “Debugging C and C++ programs,” on page 257
"Halting when certain functions are called in C” on page 184
"Modifying the value of a C variable” on page 184
"Halting on a line in C only if a condition is true” on page 185
"Debugging C when only a few parts are compiled with TEST” on page 186
"Capturing C output to stdout” on page 186
"Calling a C function from Debug Tool” on page 187
"Displaying raw storage in C” on page 187
"Debugging a C DLL” on page 187
"Getting a function traceback in C” on page 188
"Tracing the run-time path for C code compiled with TEST” on page 188
"Finding unexpected storage overwrite errors in C” on page 189
"Finding uninitialized storage errors in C” on page 189
"Halting before calling a NULL C function” on page 190

Example: sample C program for debugging

The program below is used in various topics to demonstrate debugging tasks.

This program is a simple calculator that reads its input from a character buffer. If integers are read, they are pushed on a stack. If one of the operators (+ − * /) is read, the top two elements are popped off the stack, the operation is performed on them, and the result is pushed on the stack. The = operator writes out the value of the top element of the stack to a buffer.

CALC.H

/*----- FILE CALC.H -------------------------------*/
/*
/* Header file for CALC.C PUSHPOP.C READTOKN.C
/* a simple calculator
 façon de texte naturel
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[])
Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

[Chapter 27, “Debugging a C program in full-screen mode,” on page 181](#)

### Halting when certain functions are called in C

[“Example: sample C program for debugging” on page 181](#)

To halt just before read_token is called, issue the command:

```
AT CALL read_token;
```

To halt just after read_token is called, issue the command:

```
AT ENTRY read_token;
```

To take advantage of either of the above actions, you must compile your program with the TEST compiler option.

### 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 181](#)

Run the CALC program above to the statement labeled **CALC**. move the cursor over num and press PF4 (LIST). The following appears in the Log window:
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 08B6E10 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 afterwards because a specific condition is present. Setting a simple line breakpoint is an inefficient way to debug the program because you need to execute the GO command a thousand times to reach the specific condition. You can instruct Debug Tool to continue executing a program until a specific condition is present.

For example, in the main procedure of the program above, you want to stop at T_DIVIDE only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 40 { if(num2 ! = 0) GO; }
```

Line 40 is the statement labeled CALC2. The command causes 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 181

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:
3. Enter the INPUT command to enter the input data.

Refer to the following sections for more information related to the material discussed in this section.

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 181

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);
```

**Debugging a C DLL**

"Example: sample C program for debugging” on page 181

Build PUSHPOPC as a DLL, exporting push() and pop(). Build CALC.C and READTOKN.C as the program that imports push() and pop() from the DLL named PUSHPOP. When the application CALC starts the DLL, PUSHPOP will not be known to 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 181

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(READTOKEN)" ::> 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:

```c
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.

```c
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 181

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 60 BYPASS command. This allows you to continue your debug session without raising a condition.
Chapter 28. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 35, “Debugging C and C++ programs,” on page 257
"Halting when certain functions are called in C++” on page 195
"Modifying the value of a C++ variable” on page 195
"Halting on a line in C++ only if a condition is true” on page 196
"Viewing and modifying data members of the this pointer in C++” on page 197
"Debugging C++ when only a few parts are compiled with TEST” on page 197
"Capturing C++ output to stdout” on page 198
"Calling a C++ function from Debug Tool” on page 198
"Displaying raw storage in C++” on page 199
"Debugging a C++ DLL” on page 199
"Getting a function traceback in C++” on page 199
"Tracing the run-time path for C++ code compiled with TEST” on page 200
"Finding unexpected storage overwrite errors in C++” on page 200
"Finding uninitialized storage errors in C++” on page 201
"Halting before calling a NULL C++ function” on page 202

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

```cpp
/*----- FILE CALC.HPP -----------------------------------------------*/
/*
/* Header file for CALC.CPP PUSHPOP.CPP READTOKN.CPP
/* a simple calculator
/*-----------------------------------------------*/
typedef enum tok {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"
/*--------------------------------------------------------------------*/
int IntStack::push(int num)
{
    IntLink *ptr;
    ptr = new IntLink;
    ptr->set_i(num);
    ptr->set_next(top);
    top = ptr;
}
/*--------------------------------------------------------------------*/
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"

static char nextchar(void)
{
    char *buf_in = "2 18 + = 5 / = ";
    static int index; /* starts at 0 */
    char ret;
    ret = buf_in[index];
    ++index;
    return ret;
}

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 sections for more information related to the material discussed in this section.
Halting when certain functions are called in C++

You need to include the C++ signature along with the function name to set an AT ENTRY or AT CALL breakpoint for a C++ function.

```
“Example: sample C++ program for debugging” on page 191
```

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 issue the command:

```
LIST NAMES
```

which displays the names of all the blocks and variables for the currently qualified program. Debug Tool displays information similar to the following 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 command:
AT CALL IntStack::push(int)
```

```
To halt just after IntStack::push(int) is called, issue the command:
AT ENTRY IntStack::push(int) ;
```

in the same way as the AT CALL command.

```
To be able to halt, the file with the calling code must be compiled with the TEST compiler option.
```

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 191
```
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 sections for more information related to the material
discussed in this section.

**Related tasks**

- [Using C and C++ variables with Debug Tool](#) on page 258

---

**Halting on a line in C++ only if a condition is true**

Often a particular part of your program works fine for the first few thousand
times, but fails under certain conditions. You don't want to set a simple line
breakpoint because you will have to keep entering GO.

“Example: sample C++ program for debugging” on page 191

For example, in main you want to stop in T_DIVIDE only if the divisor is 0 (before
the exception occurs). Set the breakpoint like this:

AT 40 { if(num2 != 0) GO; }

---
Line 40 is the statement labeled \texttt{CALC2}. The command causes Debug Tool to stop at line 40. If the value of \texttt{num} is not 0, the program will continue. Debug Tool stops on line 40 only if \texttt{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 \texttt{IntLink}, the command:

\texttt{LIST TITLED ;}

responds with a list that includes \texttt{this}. With the command:

\texttt{DESCRIBE ATTRIBUTES *this ;}

you will see the types of the data elements pointed to by the \texttt{this} pointer. With the command:

\texttt{LIST *this ;}

you will list the data member of the object pointed to and see something like:

\texttt{LIST * this ;}
\texttt{(* this).i = 4}
\texttt{(* this).next = 0x0}

in the Log window. To modify element \texttt{i}, enter either the command:

\texttt{i = 2001;}

or, if you have ambiguity (for example, you also have an auto variable named \texttt{i}), enter:

\texttt{(* this).i = 2001 ;}

**Debugging C++ when only a few parts are compiled with TEST**

“Example: sample C++ program for debugging” on page 191

Suppose you want to set a breakpoint at entry to function \texttt{IntStack::push(int)} in the file \texttt{PUSHPOP.CPP}. \texttt{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:

\texttt{LIST NAMES CUS}

The \texttt{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 \texttt{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 \texttt{PUSHPOP} might or might not be displayed. If it is displayed, enter:

\texttt{SET QUALIFY CU "USERID.MFISTART.CPP(PUSHPOP)"}
\texttt{AT ENTRY IntStack::push(int) ;}
\texttt{GO ;}

or

\texttt{AT ENTRY "USERID.MFISTART.CPP(PUSHPOP)":>push}
\texttt{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 sections for more information related to the material
discussed in this section.

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 191

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);

Debugging a C++ DLL

"Example: sample C++ program for debugging" on page 191

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(READTOKN)" : > read_token.
From LINE 18 in C function "USERID.MFISTART.CPP(CALC)" : > main" => %BLOCK2.

which shows the traceback of callers.
Tracing the run-time path for C++ code compiled with TEST

To trace a program showing the entry and exit of that program without requiring any changes to it, place the following 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:

```c
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:

```c
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.

```c
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 */
```
Find the address of a with the command:

```
LIST &a;
```

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, it is likely that it is 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 sections for more information related to the material discussed in this section.

**Related references**

zos/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 29. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 36, “Debugging an assembler program,” on page 281
"Defining a compilation unit as assembler and loading debug data” on page 206
"Deferred LDDs” on page 207
"Halting when certain assembler routines are called” on page 209
"Displaying and modifying the value of assembler variables or storage” on page 209
"Halting on a line in assembler only if a condition is true” on page 210
"Getting an assembler routine traceback” on page 210
"Finding unexpected storage overwrite errors in assembler” on page 211

Example: sample assembler program for debugging

The program below is used in various topics to demonstrate debugging tasks.

To run this sample program, do the following steps:
1. Verify that the debug file for this assembler program is located in the SUBXMP and DISPARM members of the yourid.EQALANGX data set.
2. Start 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

SUBXMPASM

***************************************************************
* * 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 CEEOUT to issue the greeting message CALL CEEOUT,(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
CALL  DISPARM

* 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'
COUNTER DC F'-1'
XMPPPA CEEPPA ,
* The Workarea and DSA
WORKAREA DSECT
ORG ++CEEDSASZ
CALLMOUT CALL ,,(,),VL,MF=L
FBCODE DS 3F
DS 0D
WORKSIZE EQU *=WORKAREA
PRINT NOGEN
CEEDSA ,
* Mapping of the dynamic save area
CEECAA ,
* Mapping of the common anchor area
R0 EQU 0
R01 EQU 1
R13 EQU 13
END SUBXMP Nominate SUBXMP as the entry point

DISPARM.ASM
**********************************************************************************
* * NAME: DISPARM *
* * Shows an assembler subroutine that displays inbound parameters and returns. *
* *
**********************************************************************************

DISPARM CEEENTRY PPA=PARAMPPA,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),CEEO80
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
SAVCTR ST   R1,COUNTER
CL   R1,=F'5'
BH   LOOPEND
LA   R1,(R1)
B   SAVCTR
LOOPEND DS 0H
CLC CHARPARM(80),=CL80' ' Is the parm empty?
BNE DISPLAY_PARM No. Print it out.

* Invoke CEEMOUT to issue the error message for us
CALL CEEMOUT,(NOPARM,DEST,FBCODE),VL,=E,CALLMOUT
B  GO_TEST Time to go....

DISPLAY_PARM DS OH
* 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,=E,CALLMOUT

* AMODE Testing
GO_TEST DS OH
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 OH
LA R0,3
ST R0,COUNTER
CEETERM RC=0

* -------------------------------

WORKAREA DSECT
ORG **CEEDSAZ Leave space for the DSA fixed part
CALL3PRM CALL ,,,VL,=L 2-argument parameter list
CALLMOUT CALL ,,,VL,=L 3-argument parameter list
FBCODE DS 3F Space for a 12-byte feedback code
COUNTER DS F
BUFFSIZE DS H Halfword prefix for following string
CHARPARM DS CL255 80-byte buffer
DS 0D

WORKSIZE EQU =-WORKAREA
PRINT NOKEN
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
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) hlq.qual1.dsn
```

Alternatively, you might wait until you have stopped for some other reason after "mycu" has appeared and then use the SET SOURCE or SET DEFAULT LISTING commands to direct 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

To halt after the DISPARM routine is called, enter the command:

```
AT ENTRY DISPARM
```

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.

If you have many breakpoints set in your program, you can issue the command:

```
QUERY LOCATION
```

to indicate where in your program execution has been interrupted. The Debug Tool Log window displays something similar to:

```
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 CALL by entering AT 70 ; 60 ; 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  180E2150 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 203

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 203

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 30. 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 sections for more information related to the material discussed in this section.

Related tasks
- Chapter 23, “Using full-screen mode: overview,” on page 117
- Chapter 30, “Customizing your full-screen session”
- “Defining PF keys” on page 214
- “Defining a symbol for commands or other strings” on page 214
- “Customizing the layout of windows on the session panel” on page 214
- “Customizing session panel colors” on page 216
- “Customizing profile settings” on page 217
- “Saving customized settings in a preferences files” on page 219

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 one of the following commands:

- For PL/I:
  ```
  SET PF8 'Down' = SCROLL DOWN PAGE ;
  ```
- For C and C++:
  ```
  SET PF8 "Down" = SCROLL DOWN PAGE ;
  ```

Use single quotation marks for PL/I, double quotation marks for C and C++. Assembler, COBOL, and disassembly allow the use of single or double quotation marks. The string set apart by the quotation marks (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 sections for more information related to the material discussed in this section.

Related references
- “Initial PF key settings” on page 128
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. Afterwards, 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)". The use of single quotes (', ') or double quotes ("",") is language dependent.
- 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 windows on the session panel

To change the relative layout of the Source, Monitor, and Log windows, use the PANEL LAYOUT command (the PANEL keyword is optional).

The PANEL LAYOUT command displays the panel below, showing the six possible window layouts.

Initially, the session panel uses the default window layout 1.

Follow the instructions on the screen, then press the END PF key to save your changes and return to the main session panel in the new layout.
Note: You can choose only one of the six layouts. Also, only one of each type of window can be visible at a time on your session panel. For example, you cannot have two Log windows on a panel.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
“Opening and closing session panel windows”
“Resizing session panel windows”
“Zooming a window to occupy the whole screen” on page 216
“Saving customized settings in a preferences files” on page 219

Related references
“Debug Tool session panel” on page 117

Opening and closing session panel windows
To close a window, do one of the following:

• Type the WINDOW CLOSE command, move the cursor to the window you want to close, then press Enter.
• Enter the WINDOW CLOSE LOG, WINDOW CLOSE MONITOR, or WINDOW CLOSE SOURCE command.

When you close a window, the remaining windows occupy the full area of the screen.

To open a window, enter the WINDOW OPEN LOG, WINDOW OPEN MONITOR, or WINDOW OPEN SOURCE command.

The WINDOW CLOSE command can be assigned to a PF key.

If you want to monitor the values of selected variables as they change during your Debug Tool session, the Monitor window must be open. If it is closed, open it as described above. The Monitor window occupies the available space according to your selected window layout.

If at any time during your session you open a window and the contents assigned to it are not available, the window is empty.

Resizing session panel windows
To resize windows, type WINDOW SIZE on the command line, move the cursor to where you want the window boundary, then press Enter. The WINDOW keyword is optional.

Rather than using the cursor, you can also explicitly specify the number of rows or columns you want the window to contain (as appropriate for the window layout). For example, to change the Source window from 10 rows deep to 12 rows deep, enter:
WINDOW SIZE 12 SOURCE

WINDOW SIZE can be assigned to a PF key.

To restore window sizes to their default values for the current 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Color</th>
<th>Highlight</th>
<th>Intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>: field headers</td>
<td>TURQ</td>
<td>NONE</td>
</tr>
<tr>
<td>output fields</td>
<td>GREEN</td>
<td>NONE</td>
<td>LOW</td>
</tr>
<tr>
<td>Monitor: contents</td>
<td>TURQ</td>
<td>REVERSE</td>
<td>LOW</td>
</tr>
<tr>
<td>line numbers</td>
<td>TURQ</td>
<td>REVERSE</td>
<td>LOW</td>
</tr>
<tr>
<td>Source : listing area</td>
<td>WHITE</td>
<td>REVERSE</td>
<td>LOW</td>
</tr>
<tr>
<td>prefix area</td>
<td>TURQ</td>
<td>REVERSE</td>
<td>LOW</td>
</tr>
<tr>
<td>suffix area</td>
<td>YELLOW</td>
<td>REVERSE</td>
<td>LOW</td>
</tr>
<tr>
<td>current line</td>
<td>RED</td>
<td>REVERSE</td>
<td>HIGH</td>
</tr>
<tr>
<td>breakpoints</td>
<td>GREEN</td>
<td>NONE</td>
<td>LOW</td>
</tr>
<tr>
<td>Log : program output</td>
<td>TURQ</td>
<td>NONE</td>
<td>HIGH</td>
</tr>
<tr>
<td>test input</td>
<td>YELLOW</td>
<td>NONE</td>
<td>LOW</td>
</tr>
<tr>
<td>test output</td>
<td>GREEN</td>
<td>NONE</td>
<td>HIGH</td>
</tr>
<tr>
<td>line numbers</td>
<td>BLUE</td>
<td>REVERSE</td>
<td>HIGH</td>
</tr>
<tr>
<td>Command line</td>
<td>WHITE</td>
<td>NONE</td>
<td>HIGH</td>
</tr>
<tr>
<td>Window headers</td>
<td>GREEN</td>
<td>REVERSE</td>
<td>HIGH</td>
</tr>
<tr>
<td>Toplevel delimiter</td>
<td>BLUE</td>
<td>REVERSE</td>
<td>HIGH</td>
</tr>
<tr>
<td>Search target</td>
<td>RED</td>
<td>NONE</td>
<td>HIGH</td>
</tr>
<tr>
<td>Enter END/QUIT to return with current settings saved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANCEL to return without current settings saved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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]

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 command 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 sections for more information related to the material discussed in this section.

**Related tasks**

- “Saving customized settings in a preferences files”

---

**Saving customized settings in a preferences files**

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 30, “Customizing your full-screen session,” on page 213 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 136 for instructions.
Part 5. Debugging your programs by using Debug Tool commands
Chapter 31. 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 sections for more information related to the material discussed in this section.

Related tasks
- “Entering commands on the session panel” on page 125
- “Abbreviating Debug Tool keywords” on page 224
- “Entering multiline commands in full-screen and line mode” on page 225
- “Entering multiline commands in a command file” on page 225
- “Entering multiline commands without continuation” on page 226
- “Using blanks in Debug Tool commands” on page 226
- “Entering comments in Debug Tool commands” on page 226
- “Using constants in Debug Tool commands” on page 227
- “Getting online help for Debug Tool command syntax” on page 227

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:

- All keywords and identifiers must be the correct case. Debug Tool does not do conversion 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 "|".
- 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 (lbrace), right brace (rbrace), left bracket (lbracket), and right bracket (rbracket). Although alternate code points will be accepted as input for the braces and brackets, the primary code points will always be logged.

**Character case in COBOL and PL/I**

When the current programming language setting is *not* C, commands can generally be either uppercase, lowercase, or mixed. Characters in the range a through z are automatically converted to uppercase except within comments and quoted literals. Also, in PL/I, only "|" and "~" can be used as the boolean operators for OR and NOT.

---

**Abbreviating 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<space>= 2;" not "TSO<space>= 2;". If you want to define a procedure named USE, you must enter "USE<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:

<table>
<thead>
<tr>
<th>If you enter the following command...</th>
<th>It will be interpreted as...</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT 3</td>
<td>A 3</td>
</tr>
<tr>
<td>GO</td>
<td>6</td>
</tr>
<tr>
<td>QUALIFY BLOCK B</td>
<td>Q B B</td>
</tr>
</tbody>
</table>
If you enter the following command... | It will be interpreted as...
---|---
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 and line mode**

If you need to use more than one line when entering a command, you must use a continuation character.

When you are entering a command in interactive mode, the continuation character must be the last nonblank character in each line that is to be continued. In the following example:

```
LIST (" this is a very very vvvvvvvvvvvvvvvvvvvv – very long string");
```

the continuation character is the single-byte character set (SBCS) hyphen (-).

If you want to end a line with a character that would be interpreted as a continuation character, follow that character with another valid nonblank 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++, the back slash character (\) can also be used.

When Debug Tool is awaiting the continuation of a command in full-screen mode, you receive a continuation prompt of "MORE..." until the command is completely entered and processed. When continuation is indicated in line mode, you receive a continuation prompt of "PENDING..." until the command is completely entered and processed.

**Entering multiline commands in a command 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 back slash continuation character is used. The following is an example of the continuation character for C:

```
LIST (" this is a very very vvvvvvvvvvvvvvvvvvvv\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, an additional double (") or single (’) quote is required in the continuation line, and the character following the quote is considered to follow immediately after the last character in the continued line. The following is an example of line continuation for COBOL:

```plaintext
123456 LIST (* this is a very very very vvvvvvvvvvvvvvvvv
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
- 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

### 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 delimiters, 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 double (") or single (') quotes 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'0C12C3C4' 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 single quotes (') 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 sections for more information related to the material discussed in this section.

#### Related tasks

- [“Using constants in COBOL expressions” on page 236](#)

#### Related references

- [“C and C++ expressions” on page 261](#)

### 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 32. 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 237
- "Debug Tool evaluation of COBOL expressions” on page 235
- Chapter 24, “Debugging a COBOL program in full-screen mode,” on page 155
- "Using COBOL variables with Debug Tool” on page 231
- "Using DBCS characters in COBOL” on page 233
- "Using Debug Tool functions with COBOL” on page 237
- "Debug Tool commands that resemble COBOL statements”
- "%PATHCODE values for COBOL” on page 233
- "Debugging VS COBOL II programs” on page 240

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Subroutine call</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Computational assignment (including expressions)</td>
</tr>
<tr>
<td>Declarations</td>
<td>Declaration of session variables</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>Multiway switch</td>
</tr>
<tr>
<td>IF</td>
<td>Conditional execution</td>
</tr>
<tr>
<td>MOVE</td>
<td>Noncomputational assignment</td>
</tr>
<tr>
<td>PERFORM</td>
<td>Iterative looping</td>
</tr>
<tr>
<td>SET</td>
<td>INDEX and POINTER assignment</td>
</tr>
</tbody>
</table>

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 quote 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 command file input.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**
- "COBOL compiler options in effect for Debug Tool commands"
- "COBOL reserved keywords"

**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

Refer to the following sections for more information related to the material discussed in this section.

**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 sections for more information related to the material discussed in this section.

**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 sections for more information related to the material discussed in this section.

**Related tasks**
- “Accessing COBOL variables”
- “Assigning values to COBOL variables”
- “Displaying values of COBOL variables” on page 232
- “Declaring session variables in COBOL” on page 234

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 sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 5, “Preparing a COBOL program,” on page 25

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 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).  
77 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 \((a + e(1))/c \times 2\).

\[
\text{COMPUTE } xx = (a + e(1))/c \times 2;
\]

You can also use table elements in such assignments as shown in the following example:

\[
\text{COMPUTE } \text{itm-2}(1,2) = (a + 1)/e(2);
\]

The value assigned to a variable is always assigned to the storage for that variable. In an optimized program, a variable can be temporarily assigned to a register, and a new value assigned to that variable does not necessarily 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 \(b\):

\[
\text{MOVE } a \text{ OF } b \text{ TO } c \text{ OF } d;
\]

Note the qualification used in this example.

Assign the value of 123 to the first table element of \(\text{itm-2}\):

\[
\text{MOVE } 123 \text{ TO } \text{itm-2}(1,1);
\]

You can also use reference modification to assign values to variables as shown in the following two examples:

\[
\text{MOVE } aa(2:3) \text{ TO } bb;
\]

\[
\text{MOVE } aa \text{ TO } bb(1:4);
\]

Assign the value 3 to \(\text{inx1}\), the index to \(\text{itm-1}\):

\[
\text{SET } \text{inx1} \text{ TO } 3;
\]

Assign the value of \(\text{inx1}\) to \(\text{inx2}\):

\[
\text{SET } \text{inx2} \text{ TO } \text{inx1};
\]

Assign the value of an invalid address (nonnumeric 0) to \(\text{ptr}\):

\[
\text{SET } \text{ptr} \text{ TO NULL};
\]

Assign the address of \(XX\) to \(\text{ptr}\):

\[
\text{SET } \text{ptr} \text{ TO ADDRESS OF } XX;
\]

Assigns the hexadecimal value of \(X'20000'\) to the pointer \(\text{ptr}\):

\[
\text{SET } \text{ptr} \text{ TO } H'20000';
\]

**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:

\[
\text{AT 52 LIST TITLED (aa, bb, one); GO;}
\]

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 6, 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 sections for more information related to the material discussed in this section.

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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Debug Tool is not in control as the result of a path or attention situation.</td>
</tr>
<tr>
<td>0</td>
<td>Attention function (<strong>not</strong> ATTENTION condition).</td>
</tr>
<tr>
<td>1</td>
<td>A block has been entered.</td>
</tr>
<tr>
<td>2</td>
<td>A block is about to be exited.</td>
</tr>
<tr>
<td>3</td>
<td>Control has reached a label coded in the program (a paragraph name or section name).</td>
</tr>
<tr>
<td>4</td>
<td>Control is being transferred as a result of a CALL or INVOKE. The invoked routine’s parameters, if any, have been prepared.</td>
</tr>
</tbody>
</table>
Control is returning from a CALL or INVOKE. If GPR 15 contains a return code, it has already been stored.

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.)

The logic following an IF...THEN is about to be executed.

The logic following an ELSE is about to be executed.

The logic following a WHEN within an EVALUATE is about to be executed.

The logic following a WHEN OTHER within an EVALUATE is about to be executed.

The logic following a WHEN within a SEARCH is about to be executed.

The logic following an AT END within a SEARCH is about to be executed.

The logic following the end of one of the following structures is about to be executed:
- An IF statement (with or without an ELSE clause)
- An EVALUATE or SEARCH
- A PERFORM

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.)

The logic associated with one of the following phrases is about to be run:
- [NOT] ON SIZE ERROR
- [NOT] ON EXCEPTION
- [NOT] ON OVERFLOW
- [NOT] AT END (other than SEARCH AT END)
- [NOT] AT END-OF-PAGE
- [NOT] INVALID KEY

The logic following the end of a statement containing one of the following phrases is about to be run:
- [NOT] ON SIZE ERROR
- [NOT] ON EXCEPTION
- [NOT] ON OVERFLOW
- [NOT] AT END (other than SEARCH AT END)
- [NOT] AT END-OF-PAGE
- [NOT] INVALID KEY.

Note: Values in the range 3–16 can be assigned to %PATHCODE only if your program was compiled with an option supporting path hooks.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
Chapter 5, “Preparing a COBOL program,” on page 25

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 sections for more information related to the material discussed in this section.

Related tasks
"Using session variables across different languages“ on page 334

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 sections for more information related to the material discussed in this section.

Related tasks
"Displaying the results of COBOL expression evaluation“ on page 236
"Using constants in COBOL expressions“ on page 236
Enterprise COBOL for z/OS Programming Guide

Related references
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 sections for more information related to the material discussed in this section.

**Related references**

"COBOL compiler options in effect for Debug Tool commands" on page 230

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'00001234'.

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. Remember to enclose the block name in double (") or single (') quotes if case sensitive. If the name is not inside quotes, 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:
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

The above method of changing the point of view is necessary for command 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:

```cobol
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:

```cobol
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:

```cobol
LIST (main:>var-name);
```

The above method of changing the point of view is necessary for command files.
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. 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: userid.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
  ```
- In remote debug mode, add the **-qremote** option to the command that starts the daemon.
idebug -qdaemon -quiport=8000 -qlang=cobol -qremotesource=my.listing.pds

- Use the EQDEBUG 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 on how you can debug VS COBOL II programs, see Using CODE/370 with VS COBOL II and OS PL/I, SC09-1862.
Chapter 33. Debugging an OS/VS COBOL program

You can use most of the Debug Tool commands to debug OS/VS COBOL programs that have debug information available. Any exceptions are noted in Debug Tool Reference and Messages. Before debugging an OS/VS COBOL program, prepare your program as described in Chapter 6, “Preparing an OS/VS COBOL program,” on page 29.

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. Please read the information regarding non-Language Environment programs for instructions on how to start Debug Tool and debug OS/VS COBOL programs, unless OS/VS COBOL-specific information is provided.

Loading an OS/VS COBOL program’s debug information

Use the LOADDEBUGDATA (LDD) command to indicate to Debug Tool that a compile unit is an OS/VS 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 OS/VS 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. In remote debug mode, the remote debugger prompts you for the data set information when you step into the program.

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.
The Debug Tool session panel below shows the information displayed in the Source window while you debug an OS/VS COBOL program.

1 **OS COBOL**
   This indicates that the current source program is OS/VS 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 an OS/VS COBOL program

When you debug OS/VS COBOL programs the following general restrictions apply:

- When you compose Debug Tool commands, all expressions must be enclosed in single-quotes.
- 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 GOTO command.
- The only path-points for the AT PATH statement that are supported in an OS/VS COBOL program are Entry and Label.
- There are behavioral differences between OS/VS COBOL programs and other COBOL programs. OS/VS COBOL programs behaves 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, OS/VS COBOL CU’s are all known to Debug Tool when the module is loaded.
- 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 an OS/VS COBOL transaction is single-threaded; therefore, when multiple users try to concurrently debug an OS/VS COBOL program, the CICS environment initializes one OS/VS COBOL transaction at a time. Consider the following example:
  1. Three users start a transaction that runs an OS/VS 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 OS/VS COBOL program.
%PATHCODE values for OS/VS COBOL programs

This table shows the possible values for the Debug Tool variable %PATHCODE when the current programming language is OS/VS COBOL:

<table>
<thead>
<tr>
<th>%PATHCODE</th>
<th>Entry Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A block has been entered</td>
</tr>
<tr>
<td>3</td>
<td>Control has reached a label coded in the program.</td>
</tr>
</tbody>
</table>

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 an 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 34. Debugging PL/I programs

The topics below describe how to use Debug Tool to debug your PL/I programs.

Refer to the following sections for more information related to the material discussed in this section.

Related concepts
“Debug Tool evaluation of PL/I expressions” on page 252

Related tasks
Chapter 26, “Debugging a PL/I program in full-screen mode,” on page 173
Chapter 34, “Debugging PL/I programs” on page 251

Related references
“Accessing PL/I program variables” on page 251
“Debug Tool subset of PL/I commands” on page 252

“Supported PL/I built-in functions” on page 252

---

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>Scalar and vector assignment</td>
</tr>
<tr>
<td>BEGIN</td>
<td>Composite command grouping</td>
</tr>
<tr>
<td>CALL</td>
<td>Debug Tool procedure call</td>
</tr>
<tr>
<td>DECLARE or DCL</td>
<td>Declaration of session variables</td>
</tr>
<tr>
<td>DO</td>
<td>Iterative looping and composite command grouping</td>
</tr>
<tr>
<td>IF</td>
<td>Conditional execution</td>
</tr>
<tr>
<td>ON</td>
<td>Define an exception handler</td>
</tr>
<tr>
<td>SELECT</td>
<td>Conditional execution</td>
</tr>
</tbody>
</table>

---

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description or changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE</td>
<td>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.</td>
</tr>
<tr>
<td>ON</td>
<td>Performs as the AT OCCURRENCE command except it takes PL/I conditions as operands.</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BEGIN/END blocks of logic.</td>
</tr>
<tr>
<td>DECLARE</td>
<td>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.</td>
</tr>
<tr>
<td>DO</td>
<td>The three forms of DO are added; one is an extension of C’s do. 1. DO; command(s); END; 2. DO WHILE</td>
</tr>
<tr>
<td>IF</td>
<td>The IF / ELSE does not require the ENDF.</td>
</tr>
<tr>
<td>SELECT</td>
<td>The SELECT / WHEN / OTHERWISE / END programming structure is added.</td>
</tr>
</tbody>
</table>

%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.

<table>
<thead>
<tr>
<th>%PATHCODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An attention interrupt occurred.</td>
</tr>
<tr>
<td>1</td>
<td>A block has been entered.</td>
</tr>
<tr>
<td>2</td>
<td>A block is about to be exited.</td>
</tr>
<tr>
<td>3</td>
<td>Control has reached a label constant.</td>
</tr>
</tbody>
</table>
4 Control is being sent somewhere else as the result of a CALL or a function reference.

5 Control is returning from a CALL invocation or a function reference. Register 15, if it contains a return code, has not yet been stored.

6 Some logic contained in a complex DO statement is about to be executed.

7 The logic following an IF..THEN is about to be executed.

8 The logic following an ELSE is about to be executed.

9 The logic following a WHEN within a select-group is about to be executed.

10 The logic following an OTHERWISE within a select-group is about to be executed.

**PL/I conditions and condition handling**

All PL/I conditions are recognized by 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:

<table>
<thead>
<tr>
<th>Triggered condition</th>
<th>%CONDITION value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA</td>
<td>AREA</td>
</tr>
<tr>
<td>ATTENTION</td>
<td>CEE35J</td>
</tr>
<tr>
<td>COND ( CC#1 )</td>
<td>CONDITION</td>
</tr>
<tr>
<td>CONVERSION</td>
<td>CONVERSION</td>
</tr>
<tr>
<td>ENDFILE ( MF )</td>
<td>ENDFILE</td>
</tr>
<tr>
<td>ENDPAGE ( MF )</td>
<td>ENDPAGE</td>
</tr>
<tr>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>FINISH</td>
<td>CEE066</td>
</tr>
<tr>
<td>FOFL</td>
<td>CEE348</td>
</tr>
<tr>
<td>KEY ( MF )</td>
<td>KEY</td>
</tr>
<tr>
<td>NAME ( MF )</td>
<td>NAME</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>CEE34C</td>
</tr>
<tr>
<td>PENDING ( MF )</td>
<td>PENDING</td>
</tr>
<tr>
<td>RECORD ( MF )</td>
<td>RECORD</td>
</tr>
<tr>
<td>SIZE</td>
<td>SIZE</td>
</tr>
<tr>
<td>STRG</td>
<td>STRINGRANGE</td>
</tr>
<tr>
<td>STRINGSIZE</td>
<td>STRINGSIZE</td>
</tr>
<tr>
<td>SUBRG</td>
<td>SUBSCRIPTRANGE</td>
</tr>
<tr>
<td>TRANSMIT ( MF )</td>
<td>TRANSMIT</td>
</tr>
<tr>
<td>UNDEFINEDFILE ( MF )</td>
<td>UNDEFINEDFILE</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>CEE34D</td>
</tr>
<tr>
<td>ZERODIVIDE</td>
<td>CEE349</td>
</tr>
</tbody>
</table>

**Note:** For Enterprise PL/I programs, the following conditions are not supported:
- AT OCCURRENCE file conditions (ENDFILE, ENDPAGE, KEY, NAME, RECORD, TRANSMIT, UNDEFINEDFILE)
• 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 as long as the statement is not ambiguous.

This will change the description or characteristics of LIST NAMES in that:

```plaintext
LIST NAMES db<.c.skk.w>ord
```

will search for
```
<.D.B.Cskk.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.

```plaintext
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 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 sections for more information related to the material discussed in this section.

**Related tasks**

- "Using session variables across different languages" on page 334
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:
- An OFFSET variable,
- An expression, or
- A pointer that either is based or defined, a parameter, or member of either an array or a structure

must be explicitly qualified when used in expressions. 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 sections for more information related to the material discussed in this section.

Related tasks

Chapter 7, “Preparing a PL/I program,” on page 33

Accessing PL/I structures

You cannot reference elements of arrays of structures. For example, suppose a structure called PAYROLL is declared as follows:

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);
Given the way PAYROLL is declared, the following examples of commands are **valid** in Debug Tool:

```
LIST ( PAYROLL(1).NAME.LAST, PAYROLL(1).HOURS.REGULAR );
LIST ( ADDR ( PAYROLL ) );
LIST STORAGE ( PAYROLL.HOURS, 128 );
```

Given the way PAYROLL is declared, the following examples of commands are **invalid** in Debug Tool:

```
LIST ( PAYROLL(1) );
LIST ( ADDR ( PAYROLL(5) ) );
LIST STORAGE ( PAYROLL(15).HOURS, 128 ) ;
```

**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.

All PL/I constant types are supported, plus the Debug Tool PX constant.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

“Unsupported PL/I language elements” on page 254

**Supported PL/I built-in functions**

Debug Tool supports the following built-in functions for PL/I for MVS & VM:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>CSTG2</td>
<td>LOG1</td>
<td>REAL</td>
</tr>
<tr>
<td>ACOS</td>
<td>CURRENTSTORAGE</td>
<td>LOW2</td>
<td>REPEAT</td>
</tr>
<tr>
<td>ADDR</td>
<td>DATAFIELD</td>
<td>LOG2</td>
<td>SAMEKEY</td>
</tr>
<tr>
<td>ALL</td>
<td>DATE</td>
<td>MPSTR</td>
<td>SIN</td>
</tr>
<tr>
<td>ALLOCATION</td>
<td>DATETIME</td>
<td>NULL</td>
<td>SIND</td>
</tr>
<tr>
<td>ANY</td>
<td>DIM</td>
<td>OFFSET</td>
<td>SINH</td>
</tr>
<tr>
<td>ASIN</td>
<td>EMPTY</td>
<td>ONCHAR</td>
<td>SQRT</td>
</tr>
<tr>
<td>ATAN</td>
<td>ENTRYADDR</td>
<td>ONCODE</td>
<td>STATUS</td>
</tr>
<tr>
<td>ATAND</td>
<td>ERF</td>
<td>ONCOUNT</td>
<td>STORAGE</td>
</tr>
<tr>
<td>ATANH</td>
<td>ERF</td>
<td>ONFILE</td>
<td>STRING</td>
</tr>
<tr>
<td>BINARYVALUE</td>
<td>EXP</td>
<td>ONKEY</td>
<td>SUBSTR</td>
</tr>
<tr>
<td>BINVALUE2</td>
<td>GRAPHIC</td>
<td>ONLOC</td>
<td>SYSNULL</td>
</tr>
<tr>
<td>BIT</td>
<td>HBOUND</td>
<td>ONSOURCE</td>
<td>TAN</td>
</tr>
<tr>
<td>BOOL</td>
<td>HEX</td>
<td>PLIRETV</td>
<td>TAND</td>
</tr>
<tr>
<td>CHAR</td>
<td>HIGH</td>
<td>POINTER</td>
<td>TANH</td>
</tr>
<tr>
<td>COMPLETION</td>
<td>IMAG</td>
<td>POINTERADD</td>
<td>TIME</td>
</tr>
<tr>
<td>COS</td>
<td>LBOUND</td>
<td>POINTERVALUE</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>COSD</td>
<td>LENGTH</td>
<td>PTRADD3</td>
<td>UNSPEC</td>
</tr>
<tr>
<td>COSH</td>
<td>LINENO</td>
<td>PTRVALUE4</td>
<td>VERIFY</td>
</tr>
<tr>
<td>COUNT</td>
<td>LOG</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Notes:
1. Abbreviation for BINARYVALUE
2. Abbreviation for CURRENTSTORAGE
3. Abbreviation for POINTERADD
4. Abbreviation for POINTERVALUE

Debug Tool supports the following built-in functions for Enterprise PL/I:

<table>
<thead>
<tr>
<th>Function</th>
<th>Abbreviation</th>
<th>Function</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>HEXIMAGE</td>
<td>OFFSETVALUE</td>
<td>POINTERDIFF</td>
</tr>
<tr>
<td>ADDR</td>
<td>HIGH</td>
<td>ORDINALNAME</td>
<td>PTRDIFF</td>
</tr>
<tr>
<td>ALLOCATION</td>
<td>IAND</td>
<td>ORDINALPRED</td>
<td>POINTVALUE</td>
</tr>
<tr>
<td>ASIN</td>
<td>IEO</td>
<td>ORDINALSUCC</td>
<td>PTRVALUE</td>
</tr>
<tr>
<td>ATAN</td>
<td>IOR</td>
<td>ONCODE</td>
<td>PLIRETV</td>
</tr>
<tr>
<td>ATAND</td>
<td>INDEX</td>
<td>ONCONDCOND</td>
<td>RAISE2</td>
</tr>
<tr>
<td>ATANH</td>
<td>INOT</td>
<td>ONCHAR</td>
<td>REPEATE</td>
</tr>
<tr>
<td>BIF_DIM</td>
<td>ISRL</td>
<td>ONGSOURCE</td>
<td>SAMEKEY</td>
</tr>
<tr>
<td>BINARYVALUE</td>
<td>ISLL</td>
<td>ORNSOURCE</td>
<td>SEARCH</td>
</tr>
<tr>
<td>BINVALUE</td>
<td>LBOUND</td>
<td>ONCONDID</td>
<td>SEARCHR</td>
</tr>
<tr>
<td>COPY</td>
<td>LENGTH</td>
<td>ONCOUNT</td>
<td>SIN</td>
</tr>
<tr>
<td>COS</td>
<td>LINENO</td>
<td>ONFILE</td>
<td>SIND</td>
</tr>
<tr>
<td>COSD</td>
<td>LOG</td>
<td>ONKEY</td>
<td>SINH</td>
</tr>
<tr>
<td>COSH</td>
<td>LOG10</td>
<td>ONLOC</td>
<td>SQRT</td>
</tr>
<tr>
<td>COUNT</td>
<td>LOG2</td>
<td>PAGENO</td>
<td>SUBSTR</td>
</tr>
<tr>
<td>DATAFIELD</td>
<td>LOGGAMMA</td>
<td>POINTER</td>
<td>SYSNULL</td>
</tr>
<tr>
<td>DATE</td>
<td>LOW</td>
<td>PTR</td>
<td>TAN</td>
</tr>
<tr>
<td>DATETIME</td>
<td>LOWER2</td>
<td>POINTADD</td>
<td>TAND</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>LOWERCASE</td>
<td>PTRADD</td>
<td>TANH</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>MAXLENGTH</td>
<td>POINTERSUBTRACT</td>
<td>TALLY</td>
</tr>
<tr>
<td>ENTRYADDR</td>
<td>NULL</td>
<td>PTRSUBTRACT</td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERF</td>
<td>OFFSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERF</td>
<td>OFFSETADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>OFFSETSUBTRACT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILEOPEN</td>
<td>OFFSETDIFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAMMA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBOUND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
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 run-time PTF for APAR PQ94347, which is available for z/OS Version 1 Release 4, Version 1 Release 5, and 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 run-time PTF for APAR PK16316, which is available for z/OS Version 1 Release 4, Version 1 Release 5, Version 1 Release 6, and Version 1 Release 7.
Debug Tool does not support the following built-in functions for Enterprise PL/I:

<table>
<thead>
<tr>
<th>Function</th>
<th>Abs</th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All</td>
<td>Graphic</td>
</tr>
<tr>
<td>ANY</td>
<td>Any</td>
<td>Imag</td>
</tr>
<tr>
<td>BIT</td>
<td>Bit</td>
<td>Mpsrt</td>
</tr>
<tr>
<td>BOOL</td>
<td>Bool</td>
<td>Real</td>
</tr>
<tr>
<td>CHAR</td>
<td>Char</td>
<td>Status</td>
</tr>
<tr>
<td>COMPLETION</td>
<td>Completion</td>
<td>Storage</td>
</tr>
<tr>
<td>CSTG(2)</td>
<td>CSTG</td>
<td>String</td>
</tr>
<tr>
<td>CURRENTSTORAGE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Refer to the following sections for more information related to the material discussed in this section.

**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: userid.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 EQAUSEDAT 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 (for file conditions: ENDFILE, ENDPAGE, KEY, NAME, RECORD, TRANSMIT, UNDEFINEDFILE)
- AT OCCURRENCE CONDITION conditions (name)
- GOTO LABEL

While debugging Enterprise PL/I programs, the following restrictions apply:
- If you want to use the AT LABEL command, be aware of the following restrictions:
  - 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 Version 3 Release 4 programs and you comply with the following requirements, you can use the AT LABEL command to set breakpoints (except at a label variable):
    - You apply PTFs for APARs PK00118 and PK00339 to the Enterprise PL/I Version 3 Release 4 compiler.
- For expressions, you cannot do either of the following:
  - prefacen 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 252 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.
- If you apply the Language Environment run-time PTF for APAR PQ95664 (available for z/OS Version 1 Release 4 through Version 1 Release 6), you can use the DESCRIBE ENVIRONMENT command.
- If you apply the Language Environment run-time PTF for APAR PK30522 (available for z/OS Version 1 Release 4 through Version 1 Release 8), you can use the DESCRIBE ATTRIBUTES command.
Chapter 35. 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 271

Refer to the following sections for more information related to the material discussed in this section.

Related concepts
"C and C++ expressions" on page 261
"Debug Tool evaluation of C and C++ expressions" on page 265
"Scope of objects in C and C++" on page 268
"Blocks and block identifiers for C" on page 270
"Blocks and block identifiers for C++" on page 270
"Monitoring storage in C++" on page 278

Related tasks
Chapter 27, "Debugging a C program in full-screen mode," on page 181
Chapter 28, "Debugging a C++ program in full-screen mode," on page 191
"Using C and C++ variables with Debug Tool" on page 258
"Declaring session variables with C and C++" on page 260
"Calling C and C++ functions from Debug Tool" on page 262
"Intercepting files when debugging C and C++ programs" on page 266
"Displaying environmental information" on page 272
"Stepping through C++ programs" on page 275
"Setting breakpoints in C++" on page 275
"Examining C++ objects" on page 277
"Qualifying variables in C and C++" on page 273

Related references
"Debug Tool commands that resemble C and C++ commands"
"%PATHCODE values for C and C++" on page 260
"C reserved keywords" on page 263
"C operators and operands" on page 264
"Language Environment conditions and their C and C++ equivalents" on page 264

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block ({})</td>
<td>Composite command grouping</td>
</tr>
<tr>
<td>break</td>
<td>Termination of loops or switch commands</td>
</tr>
<tr>
<td>declarations</td>
<td>Declaration of session variables</td>
</tr>
<tr>
<td>do/while</td>
<td>Iterative looping</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>expression</td>
<td>Any C expression except the conditional (?) operator</td>
</tr>
<tr>
<td>for</td>
<td>Iterative looping</td>
</tr>
<tr>
<td>if</td>
<td>Conditional execution</td>
</tr>
<tr>
<td>switch</td>
<td>Conditional execution</td>
</tr>
</tbody>
</table>

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 sections for more information related to the material discussed in this section.

**Related references**
- “C reserved keywords” on page 263
- 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 sections for more information related to the material discussed in this section.

**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 259

**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 sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 8, “Preparing a C program,” on page 37
- Chapter 9, “Preparing a C++ program,” on page 43

**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] ); GO;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (GO), stops at line 25, and displays the variable names and their values.

If you want to see the result of their addition, enter:

```
AT 25 LIST ( X + row[X] + col[X] ); GO;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (GO), stops at line 25, and displays the result of the expression.

Put commas between the variables when listing more than one. If you do not want to display the variable names when issuing the LIST command, enter LIST UNTITLED.

You can also list variables with the printf function call as follows:

```
printf ("X=%d, row=%d, col=%d\n", X, row[X], col[X]);
```

The output from printf, however, does not appear in the Log window and is not recorded in the log file unless you SET INTERCEPT ON FILE stdout.

**Assigning values to C and C++ variables**

To assign a value to a C and C++ variable, you use an assignment expression. Assignment expressions assign a value to the left operand. The left operand must be a modifiable lvalue. An lvalue is an expression representing a data object that can be examined and altered.

C contains two types of assignment operators: simple and compound. A simple assignment operator gives the value of the right operand to the left operand.

**Note:** Only the assignment operators that work for C will work for C++, that is, there is no support for overloaded operators.

The following example demonstrates how to assign the value of number to the member employee of the structure payroll:

```
payroll.employee = number;
```

Compound assignment operators perform an operation on both operands and give the result of that operation to the left operand. For example, this expression gives the value of index plus 2 to the variable index:

```
index += 2
```

Debug Tool supports all C operators except the tenary operator, as well as any other full C language assignments and function calls to user or C library functions.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

“Calling C and C++ functions from Debug Tool” on page 262
%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++.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Debug Tool is not in control as the result of a path or attention situation.</td>
</tr>
<tr>
<td>0</td>
<td>Attention function <em>(not ATTENTION condition).</em></td>
</tr>
<tr>
<td>1</td>
<td>A block has been entered.</td>
</tr>
<tr>
<td>2</td>
<td>A block is about to be exited.</td>
</tr>
<tr>
<td>3</td>
<td>Control has reached a user label.</td>
</tr>
<tr>
<td>4</td>
<td>Control is being transferred as a result of a function reference. The invoked routine’s parameters, if any, have been prepared.</td>
</tr>
<tr>
<td>5</td>
<td>Control is returning from a function reference. Any return code contained in register 15 has not yet been stored.</td>
</tr>
<tr>
<td>6</td>
<td>Some logic contained by a conditional do/while, for, or while statement is about to be executed. This can be a single or null statement and not a block statement.</td>
</tr>
<tr>
<td>7</td>
<td>The logic following an if(...) is about to be executed.</td>
</tr>
<tr>
<td>8</td>
<td>The logic following an else is about to be executed.</td>
</tr>
<tr>
<td>9</td>
<td>The logic following a case within an switch is about to be executed.</td>
</tr>
<tr>
<td>10</td>
<td>The logic following a default within a switch is about to be executed.</td>
</tr>
<tr>
<td>13</td>
<td>The logic following the end of a switch, do, while, if(...), or for is about to be executed.</td>
</tr>
<tr>
<td>17</td>
<td>A goto, break, continue, or return is about to be executed.</td>
</tr>
</tbody>
</table>

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 sections for more information related to the material discussed in this section.

Related tasks

"Using session variables across different languages" on page 334
"Qualifying variables and changing the point of view in C and C++" on page 272

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:

```c
int x;
int y;
x = y = 1;
printf("%d %d %d\n", 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
"));
  ```

- 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_c = "abcdef" + 0x2;
  *pointer_to_long = 3521L = 0x69a1;
  float_val = 3e-11 + 6.6E-10;
  char_val = '\"';
  ```

- 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 sections for more information related to the material discussed in this section.

**Related references**

- [Debug Tool evaluation of C and C++ expressions](http://z/OS XL C/C++ Language Reference) on page 265
- [Calling C and C++ functions from Debug Tool](http://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:

```c
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 sections for more information related to the material discussed in this section.

**Related tasks**
- Chapter 8, “Preparing a C program,” on page 37
- Chapter 9, “Preparing a C++ program,” on page 43

**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.

<table>
<thead>
<tr>
<th>auto</th>
<th>else</th>
<th>long</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>const</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>continue</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>int</td>
<td>struct</td>
<td>_Packed</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Precedence level</th>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>left to right</td>
<td>() [ ] . -&gt;</td>
</tr>
<tr>
<td>Unary</td>
<td>right to left</td>
<td>++ -- - + ! ~ &amp;</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>left to right</td>
<td>* / %</td>
</tr>
<tr>
<td>Additive</td>
<td>left to right</td>
<td>+ -</td>
</tr>
<tr>
<td>Bitwise shift</td>
<td>left to right</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>left to right</td>
<td>&lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>left to right</td>
<td>++ !=</td>
</tr>
<tr>
<td>Bitwise logical AND</td>
<td>left to right</td>
<td>&amp;</td>
</tr>
<tr>
<td>Bitwise exclusive OR</td>
<td>left to right</td>
<td>^ or ~</td>
</tr>
<tr>
<td>Bitwise inclusive OR</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>Logical AND</td>
<td>left to right</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Logical OR</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>right to left</td>
<td>= += -= *= /=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;&lt;= &gt;&gt;= %= &amp;= ^=</td>
</tr>
<tr>
<td>Comma</td>
<td>left to right</td>
<td>,</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Language Environment condition</th>
<th>Description</th>
<th>Equivalent C/C++ condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE341</td>
<td>Operation exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>CEE342</td>
<td>Privileged operation exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>CEE343</td>
<td>Execute exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>CEE344</td>
<td>Protection exception</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>CEE345</td>
<td>Addressing exception</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>CEE346</td>
<td>Specification exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>CEE347</td>
<td>Data exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE348</td>
<td>Fixed point overflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE349</td>
<td>Fixed point divide exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE34A</td>
<td>Decimal overflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE348</td>
<td>Decimal divide exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE34C</td>
<td>Exponent overflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>Language Environment condition</td>
<td>Description</td>
<td>Equivalent C/C++ condition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>CEE34D</td>
<td>Exponent underflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE34E</td>
<td>Significance exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>CEE34F</td>
<td>Floating-point divide exception</td>
<td>SIGFPE</td>
</tr>
</tbody>
</table>

**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](https://www.ibm.com/support/knowledgecenter/SSEZFT_8.2.0/com.ibm.zos.v2r2.mpXLr.mi80000f.pdf). 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 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 \(-2^{31} \text{ to } 2^{31}\)
- 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 sections for more information related to the material discussed in this section.

**Related references**

- ["C and C++ expressions" on page 261](#)
- [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 sections for more information related to the material discussed in this section.

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 dllibload(), dllibfree(), fetch(), and release().

"Example: referencing variables and setting breakpoints in C and C++ blocks" on page 271

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.
- Blocks enclosed in this outermost block are sequentially named: %BLOCK2, %BLOCK3, %BLOCK4, and so on in order of their appearance in the function.

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.

```c
#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;
    int j;
    for (i = 0; i < length–1; i++) { /* Block %BLOCK2 */
        int j;
        for (j = i+1; j < length; j++) { /* Block %BLOCK3 */
            static int temp;
            temp = table[i];
            table[i] = table[j];
            table[j] = temp;
        }
    }
}
```

---

**Scope and visibility of objects**

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:

```c
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 the program above, the function sort has three blocks:

```c
sort
%BLOCK2
%BLOCK3
```

The following example sets a breakpoint on entry to the second block of sort:

```c
at entry sort:>%BLOCK2;
```

The following example sets a breakpoint on exit of the first block of main and lists
the entries of the sorted table.

```c
at exit main {
    for (i = 0; i < length; i++)
        printf("table entry %d is %d\n", i, table[i]);
}
```

The following example lists the variable temp in the third block of sort. This is
possible since temp has the static storage class.

```c
LIST sort:>%BLOCK3:temp;
```

**Displaying environmental information**

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:

```plaintext
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 274

**Related tasks**

"Qualifying variables in C and C++"

"Changing the point of view in C and C++"

**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:

```plaintext
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 double 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 double 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" on page 274

Refer to the following sections for more information related to the material discussed in this section.

**Related concepts**

"Blocks and block identifiers for C" on page 270

**Changing the point of view in C and C++**

To change the point of view from the command line or a command file, use qualifiers in conjunction 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**

The examples below use the following program.

LOAD MODULE NAME: MAINMOD
SOURCE FILE NAME: MVSID.SORTMAIN.C

```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

```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**

- 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:
Because length is in the current load module and compilation unit, it can also be changed by:

```c
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:

```c
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"::"MVSID.SORTSUB.C":>sort:%BLOCK3:>temp;
```

### Changing the point of view

- Qualify to the second nested block in the function sort() in sort.

```c
SET QUALIFY BLOCK %BLOCK2;
```

You can do this in a number of other ways, including:

```c
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:

```c
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.

```c
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()
```

The following examples are invalid:

```
AT ENTRY shapes
AT ENTRY shapes::square
AT ENTRY shapes::square(int)
```

Where shapes is a class. Cannot set breakpoint on a class. (There is no block identifier for a class.)

Invalid since method square must be followed by its parameter list.

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 sections for more information related to the material discussed in this section.

**Related tasks**

“Retrieving commands from the Log and Source windows” on page 129
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 sections for more information related to the material discussed in this section.

Related tasks

Example: displaying attributes of C++ objects

The examples below use the following definitions.

```cpp
class shape { ... }

class line : public shape {
    member variables of class line...
}  

line edge;
```

Displaying object attributes

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

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

If a class contains static data, the static data will be shown as part of the class when displayed. For example:

```cpp
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**

To avoid ambiguity, variables declared at file scope can be referenced using the global scope operator `::`. For example:

```c
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.

```c
int dbl(int j) /* line 1 */
{
  /* line 2 */
  return 2*j;
  /* line 3 */
}
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.

```assembly
* int dbl(int j)
  ST  r1,152(/r13)
* {
  EX  r0,HOOK..PGM-ENTRY
  * return 2*j;
  EX  r0,HOOK..STMT
L  r15,152(/r13)
```
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 36. 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 10, “Preparing an assembler program,” on page 47.

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 >: 34.1
Command >>> Scrol1 >>> CSR
MONITOR >>>> 1:------2:------3:------4:------5:------6:------7:------8:------9:------10:------ LINE: 0 OF 0
**********************************************************
0009
0008
0005
0004
*********************************************************
LOG MONITOR Command Assemble
7:UP 1:?
0----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----10----+---
GO AT SET LDD Source
39 38 37 36 34 34 34 34
--+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---
===>
/ SF5800002/SF590000
Debug PUBS 00000088 00000080 00000078 00000078 00000078
Tool 0----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+---
or
Tool An and
4:11:41
/ SF5800003/SF590000
error 5655-R45:
PM:>
data 0000 CSR
/SF5800002/SF590000/SF5800006/SF590000
by
The Source or 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 first line within the statement that contains either a macro invocation or a machine instruction.

   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.

2. **Statement offset**

   This is the offset of the statement from the beginning of the CSECT.
### offset
The offset from the start of the CSECT. This column matches the left-most column in the assembler listing.

### 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.

### 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.

### 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.

### source statement
The original source statement. This column corresponds to the "Source Statement" column in the assembler listing.

Refer to the following sections for more information related to the material discussed in this section.

**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:

<table>
<thead>
<tr>
<th>%PATHCODE</th>
<th>Entry type</th>
<th>Instruction</th>
<th>Additional requirements or comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A block has been entered.</td>
<td>Any</td>
<td>External symbol whose offset corresponds to an instruction</td>
</tr>
<tr>
<td>2</td>
<td>A block is about to be exited.</td>
<td>BR R14 (07FE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BALR R14,R15</td>
<td>These instructions are considered an Exit only if this instruction is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(05EF)</td>
<td>followed by an valid instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASR R14,R15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0DEF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASSM R14,R15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0CEF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCR 15,x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(07Fx)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Control has reached a label coded in the program.</td>
<td>Any</td>
<td>Label whose offset corresponds to an instruction.</td>
</tr>
<tr>
<td>%PATHCODE</td>
<td>Entry type</td>
<td>Instruction</td>
<td>Additional requirements or comments</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Control is being transferred as a result of a CALL.</td>
<td>BALR R14,R15 (05EF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASR R14,R15 (0DEF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASSM R14,R15 (0CEF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SVC (0A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC (B218)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAL (45)</td>
<td>Except BAL 1,xxx is not considered a CALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAS (4D)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BALR x,y (05)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASR x,y (0D)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASSM x,y (0C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRAS (A7x5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRASL (C0x5)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Control is returning from a CALL.</td>
<td>Statement after CALL</td>
<td>If the statement after a CALL is an instruction, it gets an entry here.</td>
</tr>
<tr>
<td>6</td>
<td>A conditional branch is about to be executed.</td>
<td>BC x (47x)</td>
<td>x^=15 &amp; X^=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCR x (07x)</td>
<td>x^=15 &amp; X^=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCT (46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCTR (06)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCTGR (B946)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BXH (86)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BXHG (EB44)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BXLE (87)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BXLEG (EB45)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRC x (A7x4)</td>
<td>x^=15 &amp; X^=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRCL (C0x4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRCT (A7x6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRCTG (A7x7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRXH (84)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRXHG (EB44)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRXLE (85)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRXLG (EB45)</td>
<td></td>
</tr>
<tr>
<td>%PATHCODE</td>
<td>Entry type</td>
<td>Instruction</td>
<td>Additional requirements or comments</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>A conditional branch was not executed and control has “fallen-through” to the next instruction.</td>
<td>Statement after Conditional Branch</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>An unconditional branch is about to be executed.</td>
<td>BC 15,x (47Fx)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRC 15,x (A7F4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRCL 15,x (C0F4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSM (OB)</td>
<td></td>
</tr>
</tbody>
</table>

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.

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.
- No support is provided for debugging subtasks. If an ATTACH is run, the debugger always follows the parent task.
- 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 programs that use 64-bit addressing is not supported.
- Debugging of programs that use Access Register mode is not supported.
- If you are debugging a program that uses ESTAE or ESTAEX, the program behaves as if TRAP(0FF) 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(0FF) 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.

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 prior to the completion of Language Environment initialization.
- If you set a breakpoint in the prologue prior to the completion of Language Environment initialization, the breakpoint is accepted. However, Debug Tool does not stop or gain control at this breakpoint.
- There is no support for debugging Language Environment-enabled assembler MAIN programs by using Debug Tool under CICS because of the current restriction between CICS and Language Environment that prohibits running Language Environment-enabled assembler MAIN programs under CICS.

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 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 an 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 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  BRAN  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
```
Restrictions for debugging self-modifying 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 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 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, 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 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, where BC 0,Target is in a fixed-pitch font.
- Do not modify part of an instruction. Instead, replace an instruction. The following table compares coding techniques:
<table>
<thead>
<tr>
<th>Coding that modifies an instruction</th>
<th>Coding that replaces an instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModInst  BC 0,Target</td>
<td>ModInst  BC 0,Target</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MVI ModInst+1,X'F0'</td>
<td>MVI ModInst(4),NewInst</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>NewInst  BC 15,Target</td>
<td>NewInst  BC 15,Target</td>
</tr>
</tbody>
</table>
Chapter 37. 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 DYNDBUG 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
******************************************************************************
******************************************************************************
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 00C3 ??? .
  8 1950C778 0000 ??? .
A  1950C77A 0000 ??? .
E 1950C77E 00C4 ??? .
 10 1950C780 47F0 F001 BC 15,1,(R15) .
 14 1950C784 90EC D00C STM R14,R12,12(R13) .
  1A 1950C788 18BF LR R2,R1 .
  1E 1950C78E 58F0 B134 L R15,308,(R11) .
  22 1950C792 58E0 C2F0 L R14,752,(R12) .
  26 1950C796 58E0 C2F0 L R14,752,(R12) .
  2A 1950C79A 9680 E008 OI R8(R14),128 .
  2E 1950C79E 05B0 BALR R11,0 .
LOG 0---1+++--2+++++3+++--4++---5++--6 LINE: 1 OF 5
******************************************************************************
0001 IBM Debug Tool Version 7 Release 1 Mod 0
0002 08/28/2006 4:11:41 PM
0003 5655-R44 and 5655-R45: (C) Copyright IBM Corp. 1992, 2006
0004 EQA1872E An error occurred while opening file: INSPPREF. The file may not exist, or is not accessible.
0006 SET DISASSEMBLY ON ;

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.
Columns 29-32
Displays the op-code mnemonic or ?? if the op-code is not valid.

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
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
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 as long as 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:

<table>
<thead>
<tr>
<th>Coding that modifies an instruction</th>
<th>Coding that replaces an instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModInst BC 0,Target</td>
<td>ModInst BC 0,Target</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MVI ModInst+1,X'F0'</td>
<td>MVC ModInst(4),NewInst</td>
</tr>
<tr>
<td></td>
<td>NewInst BC 15,Target</td>
</tr>
</tbody>
</table>

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

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

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.

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 38. 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 11, “Preparing a DB2 program,” on page 51
- “Processing SQL statements” on page 51
- “Linking DB2 programs for debugging” on page 52
- “Binding DB2 programs for debugging” on page 53
- “Debugging DB2 programs in batch mode”
- “Debugging DB2 programs in full-screen mode” on page 300

Refer to the following sections for more information related to the material discussed in this section.

Related tasks

Chapter 11, “Preparing a DB2 program,” on page 51

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 sections for more information related to the material discussed in this section.

Related tasks

Chapter 11, “Preparing a DB2 program,” on page 51
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. Fill in all the fields with the appropriate information. 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. The TEST run-time option can be specified as a parameter on the RUN subcommand. An example for a COBOL program is:

   ```
   RUN PROG(progname) PLAN(planname) LIB('user.library')
   PARMS('/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 through a VTAM terminal without the Debug Tool Terminal Interface Manager
1. Specify the MF1%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 through a VTAM terminal with the Debug Tool 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 11, “Preparing a DB2 program,” on page 51

Related references
DB2 UDB for z/OS Administration Guide
Chapter 39. 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 C, C++, COBOL, and PL/1 in any of the following debugging modes:

- remote debug mode
- full-screen mode through a VTAM terminal

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 Stored Procedure Builder or IBM DB2 Development Center.

If you are going to debug a stored procedure while another user runs that stored procedure, do the following tasks:

- Compile the stored procedure with the TEST(ALL,SYM) compiler option. Do not compile it with the TEST(NONE,SYM) compiler option.
- You cannot use the Dynamic Debug facility. If you have the facility installed, use the SET DYNDEBUG OFF command as the first command at the start of your debug session.

The topics below describe the steps for using Debug Tool to debug your DB2 stored procedures.

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- Chapter 12, “Preparing a DB2 stored procedures program,” on page 55
- “Starting Debug Tool from DB2 stored procedures: troubleshooting” on page 113

**Related references**

- Chapter 12, “Preparing a DB2 stored procedures program,” on page 55
- DB2 UDB for z/OS Application Programming and SQL Guide

Resolving some common problems

While debugging a stored procedure, it is useful to add the DB2 return codes and message code variables (SQLCODE, SQLERRMC) to the Debug Tool Program Monitor. Table 8 are some common errors and suggestions for resolving them:

**Table 8. Common problems while debugging stored procedures and resolutions to those problems**

<table>
<thead>
<tr>
<th>Error code</th>
<th>Error message</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCODE = 471, SQLERRMC = 00E79001</td>
<td>Stored procedure was stopped.</td>
<td>Start the stored procedure using DB2 Start Procedure command.</td>
</tr>
<tr>
<td>SQLCODE = 471, SQLERRMC = 00E79002</td>
<td>Stored procedure could not be started because of a scheduling problem.</td>
<td>Try using the DB2 Start Procedure command. If this does not work, contact the DB2 Administrator to raise the dispatching priority of the procedure.</td>
</tr>
</tbody>
</table>
Table 8. Common problems while debugging stored procedures and resolutions to those problems  (continued)

<table>
<thead>
<tr>
<th>Error code</th>
<th>Error message</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCODE = 471, SQLERRMC = 00E7900C</td>
<td>WLM application environment name is not defined or available.</td>
<td>Activate the WLM address space using the MVS WLM VARY command, for example: <code>WLM VARY APPLENV=applenv,RESUME</code> where <code>applenv</code> is the name of the WLM address space.</td>
</tr>
<tr>
<td>SQLCODE = 444, SQLERRMC (none)</td>
<td>Program not found.</td>
<td>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.</td>
</tr>
<tr>
<td>SQLCODE = 430, SQLERRMC (none)</td>
<td>Abnormal termination in stored procedure</td>
<td>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.</td>
</tr>
</tbody>
</table>
Chapter 40. Debugging IMS programs

You can use Debug Tool to debug IMS programs in the following ways:

- To debug your IMS Transaction Manager (TM) programs without Batch Terminal Simulator (BTS), use one of the following debugging modes:
  - full-screen mode through a VTAM terminal
  - remote debug mode
- To debug your IMS TM programs with BTS Full-Screen Image Support (FSS) to display your MFS screen formats on the TSO terminal, choose one of the following:
  - If you want all interaction to be on a single screen, use full-screen mode.
  - If you want BTS/FSS data displayed on your TSO terminal and your Debug Tool session to be displayed on another terminal, use one of the following debugging modes:
    - full-screen mode through a VTAM terminal
    - remote debug mode
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.
- To debug your batch IMS programs without BTS, use one of the following debugging modes:
  - batch mode
  - full-screen mode through a VTAM terminal
  - remote debug mode
- To debug your batch IMS programs with BTS, choose one of the following:
  - If you want all interaction to be on a single screen, use full-screen mode.
  - If you want BTS data displayed on your TSO terminal and your Debug Tool session to be displayed on another terminal, use one of the following debugging modes:
    - full-screen mode through a VTAM terminal
    - remote debug mode

Refer to the following sections for more information related to the material discussed in this section.

**Related tasks**

- Chapter 14, “Preparing an IMS program,” on page 65
- “Setting up the run-time options for your IMS Version 8 TM program by using Debug Tool Utilities” on page 66
- “Linking IMS programs for debugging” on page 67
- “Debugging IMS programs interactively” on page 306

**Related references**

*IMS/VS Batch Terminal Simulator Program Reference and Operations Manual*
Debugging IMS programs interactively

If you want to debug an IMS batch program interactively, where you enter Debug Tool commands when you need them, choose one of the following methods to start Debug Tool:

- Follow the instructions in "Starting a debugging session in full-screen mode through a VTAM terminal" on page 111 to use the MFI%LU_name or VTAM%userid parameter of the TEST run-time option. This starts Debug Tool in full-screen mode through a VTAM terminal.
- Specifying the VADTCP%IP&tcpiworkstation_id: or the TCP/IP
  &tcpiworkstation_id: parameter of the TEST run-time option. This starts Debug Tool in remote debug mode with a remote debugger.
- Run BTS in the TSO foreground, as described in the following instructions:
  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.

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.

Currently, Debug Tool can only be used to debug one iteration of a transaction at a time. When the program terminates you must close down Debug Tool before you can view the output of the transaction.

Therefore, if you use an input data set, you can only specify data for one transaction in that data set. The data for the next transaction must be entered from your TSO terminal.

A new debug session will be started automatically for the next transaction. When using FSS, you must enter the /* command on your TSO terminal to terminate the BTS session.

Debugging IMS 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 command 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 CETEST 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.

Under IMS, you can start Debug Tool in the following ways:
- Use the compiler run-time option (#pragma runopts for C and C++)
- Include CSECT CEEUOPT when linking your program (for C and C++)
- Use the Language Environment callable service CETEST (_ctest() for C and C++)
Debugging non-Language Environment IMS MPP programs

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 instructions.
2. Choose a debugging interface. See “Choosing an interface and gathering information” for instructions.
3. Run the EQASE T transaction, which identifies the debugging interface you chose and enables debugging. See “Running the EQASE T transaction” on page 308.
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.

You can debug IMS MPPs that do not run in Language Environment only if you purchase and install IBM Debug Tool Utilities and Advanced Functions, Version 7 Release 1 (5655-R45).

Verifying configuration and starting a region

Before you debug an IMS MPP program that does not run in Language Environment, do the following steps:

1. Consult with your system administrator and verify that you 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 EQANISE T.
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.”

Choosing an interface and gathering information

Choose from one of the following debugging interfaces and gather the indicated information:

• Use full-screen mode through a VTAM terminal without using the Debug Tool 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 through a VTAM terminal with the Debug Tool 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."

Running the EQASET transaction
When you run the EQASET transaction, you are indicating to the EQANIAFE application front-end routine your debugging preferences or requesting information about your existing preferences. You can do one of the following options:

- Enable debugging by entering the command EQASET ON. You must have already indicated a debugging preference by entering the EQASET command with any other option except OFF.
- Disable debugging by entering the command EQASET OFF.
- Debug in full-screen mode through a VTAM terminal without the Debug Tool Terminal Interface Manager by entering 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.
- Debug in full-screen mode through a VTAM terminal with the Debug Tool Terminal Interface Manager by entering the command EQASET VTAM=user_ID.
- Debug in remote debug mode with a single socket connection by entering the command EQASET TCP=IP_address%port_number. A single socket connection is available with the following remote debuggers:
  - WebSphere Developer Debugger for zSeries
  - Compiled Language Debugger component of WebSphere Studio Enterprise Developer
  - Compiled Language Debugger component of WebSphere Developer for zSeries
  - IBM Distribution Debugger Version 9.2
- Debug in remote debug mode with a multiple socket connection by entering the command EQASET VTCP=IP_address%port_number. A multiple socket connection is available with versions of the IBM Distributed Debugger that are earlier than version 9.2 (copyright date 2003/10/19).
- Display the current debugging preferences by entering the command EQASET STATUS.

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.

Syntax of the EQASET transaction
The syntax of the EQASET command is displayed in the following diagram:
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:

```plaintext
TEST(ALL,INSPIN,,debuggingPreference:*)
```

You cannot customize the other runtime options.

**MFI**

Use full-screen mode through a VTAM terminal without the Debug Tool Terminal Interface Manager. You must specify a VTAM 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 through a VTAM terminal with the Debug Tool Terminal Interface Manager. You must specify the user ID that was used to log on to the VTAM terminal designated for a debug session. 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**

Use remote debug mode with a single socket type of connection. 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.

**VTCP**

Use remote debug mode with a multiple socket type of connection type. 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.

**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 first 32 bytes of the debugging preference information is displayed.
Chapter 41. Debugging CICS programs

Before you can debug your programs under CICS, verify that the following tasks have been completed:

- Verify that your Systems Programmer has made the appropriate changes to your CICS region to support Debug Tool (see the [Debug Tool Customization Guide](#)).

Refer to the following sections for more information related to the material discussed in this section.

### Related tasks
- “Methods for starting Debug Tool under CICS” on page 104
- “Starting Debug Tool by using DTCN” on page 105
- “Link-editing EQADCCXT into your program” on page 57
- “Creating and storing a DTCN profile” on page 58
- “Starting Debug Tool by using CEFUOPT” on page 106
- “Starting Debug Tool by using compiler directives” on page 107
- “Starting Debug Tool under CICS by using CEDEF” on page 107
- “Preventing Debug Tool from stopping at EXEC CICS RETURN” on page 312
- “Saving settings while debugging a pseudo-conversational program” on page 312

### Related references
- “Debug modes under CICS”
- “Restrictions when debugging under CICS” on page 313

### Debug modes under CICS

Debug Tool can run in several different modes, providing you with the flexibility to debug your applications in the way that suits you best. These modes include:

#### Single terminal mode
This is probably the mode you will use the most. A single 3270 session is used by both Debug Tool and the application, swapping displays on the terminal as required.

As you step through your application, the terminal shows Debug Tool screens, but when an EXEC CICS SEND command is issued, that screen will be displayed. Debug Tool holds that screen on the terminal for you to review; simply press Enter to return to a Debug Tool screen. When your application issues EXEC CICS RECEIVE, the application screen again appears, so you can fill in the screen details.

#### Dual terminal mode
This mode can be useful if you are debugging screen I/O applications. Debug Tool displays its screens on a separate 3270 session than the terminal displaying the application.

You step through the application using the Debug Tool terminal and, whenever the application issues an EXEC CICS SEND, the screen is sent to the application display terminal. Note that, if you do not code IMMEDIATE on the EXEC CICS SEND command, the buffer of data might be held within CICS Terminal Control until an optimum opportunity to send it is encountered—usually the next EXEC CICS SEND or EXEC CICS RECEIVE. When the application issues an EXEC CICS RECEIVE, the Debug Tool terminal will wait until you respond to the application terminal.
Batch mode
Use this mode if you are debugging a transaction that does not have a
terminal associated with it. Debug Tool does not have a terminal, but uses
a commands file for input and writes output to the log.

Remote debug mode
Debug Tool works with a remote debugger to display results on a
graphical user interface.

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.

Saving settings while debugging a pseudo-conversational 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
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.
Refer to the following sections for more information related to the material discussed in this section.

**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.

- Debug Tool supports the use of CRTE terminals, if both the application and Debug Tool share the terminal as a principal facility in single terminal mode. CICS does not permit the use of a CRTE terminal by Debug Tool if the terminal is not the application task’s principal facility (which is the case in Dual terminal mode).
- The _ctest() function with CICS does nothing.
- The CDT# transaction is a Debug Tool service transaction used during Dual 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 Dual Terminal mode.
- References to ddnames 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 log file (INSPLOG) is not automatically started. You need to use the SET LOG ON command.
- 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 OS/VS 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 OS/VS COBOL program name that is dynamically called using EXEC CICS LOAD/CALL, Debug Tool will not start. Non-Language Environment assembler or OS/VS 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 on CEBR and CECI, see CICS Supplied Transactions.
Chapter 42. Debugging ISPF applications

You can debug ISPF applications in one of the following ways:

- Starting a separate terminal using full-screen mode through a VTAM terminal without the Debug Tool 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:
  \[ \text{TEST}(\ldots, \text{MFI}/\text{TRMLU001}) \]

- Starting a separate terminal using full-screen mode through a VTAM terminal with the Debug Tool Terminal Interface Manager. 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:
  \[ \text{TEST}(\ldots, \text{VTAM}/\text{USERABCD}) \]

- Using remote debug mode.

- Using the same emulator session. 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 display.

The rest of this section assumes that you are debugging ISPF applications using the same emulator session.

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.

Refer to the following sections for more information related to the material discussed in this section.

**Related concepts**

- "Remote debug mode" on page 5

**Related tasks**

- Chapter 30, “Customizing your full-screen session,” on page 213
- Appendix E, “Notes on debugging in remote debug mode,” on page 371
Chapter 43. 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 sections for more information related to the material
discussed in this section.

Related tasks

“Fine-tuning your programs with Debug Tool”
“Debugging without hooks, statement tables, and symbol tables” on page 318
“Debugging optimized COBOL programs” on page 319

Fine-tuning your programs with Debug Tool

After initial testing, you might want to consider the following options available to
improve performance and reduce size:
• 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 option TEST(NONE,SYM) creates
  programs that do not have hooks. 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) and the programs are compiled with certain compilers, the compiled-in hooks are replaced with run time 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 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

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 Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM can have the best performance and smallest module size, while retaining full debugging capabilities, if they are compiled with the TEST(NONE,SYM,SEPARATE) compiler option. This option specifies that no hooks are inserted and symbol table information is saved in a separate debug file.

### Debugging optimized COBOL programs

COBOL programs that are compiled with the following compilers are enhanced with features that make optimized programs easier to debug:

• Enterprise COBOL for z/OS and OS/390, Version 3 Release 2
• 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

You must compile your COBOL programs with one of the following combinations 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)
The following list describes the tasks you can do while debugging optimized COBOL programs:

- You can set breakpoints. If the optimizer moves or removes a statement, you can’t 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 change the value of a variable. Enter the SET WARNING OFF command before you begin modifying variables. You only need to enter the SET WARNING OFF command once during your debugging session. Whenever you enter a command that modifies the value of a variable, Debug Tool performs the command and displays a warning message. You must also apply APAR PQ71779 to Language Environment to take advantage of this feature.

The enhancements to the compilers help you create programs that can be debugged in the same way that you debug unoptimized programs. There are some exceptions:

- You cannot change the flow of your program.
- You cannot use the GOTO command.
- 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 44. 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 through a VTAM terminal

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 on 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 through a VTAM terminal 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.
Chapter 45. 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 sections for more information related to the material discussed in this section.

Related tasks
Chapter 16, “Starting Debug Tool from the Debug Tool Utilities,” on page 75

Debugging exclusively non-Language Environment programs

When Language Environment is not active, you can debug only assembler, disassembly, or OS/VS 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.

Refer to the following sections for more information related to the material discussed in this section.

Related references
Chapter 19, “Starting Debug Tool for batch or TSO programs,” on page 95
z/OS Language Environment Debugging Guide

Debugging CICS non-Language Environment assembler or OS/VS COBOL initial programs

The non-Language Environment assembler or OS/VS 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 46. 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 sections for more information related to the material discussed in this section.

Related tasks
“Qualifying variables and changing the point of view” on page 329
“Debugging multilanguage applications” on page 333
“Handling conditions and exceptions in Debug Tool” on page 331

Related references
“Debug Tool evaluation of HLL expressions”
“Debug Tool interpretation of HLL variables and constants” on page 328
“Debug Tool commands that resemble HLL commands” on page 328
“Coexistence with other debuggers” on page 336
“Coexistence with unsupported HLL modules” on page 336

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 sections for more information related to the material discussed in this section.

Related references
“Debug Tool evaluation of C and C++ expressions” on page 265
“Debug Tool evaluation of COBOL expressions” on page 235
“Debug Tool evaluation of PL/I expressions” on page 252
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.
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 sections for more information related to the material discussed in this section.

Related tasks

- "Qualifying variables"
- "Changing the point of view" on page 331

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: %BLOCKn, 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 sections for more information related to the material discussed in this section.

**Related references**
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 48, “Debugging across multiple processes and enclaves,” on page 339
- “Changing the point of view in C and C++” on page 273
- “Changing the point of view in COBOL” on page 239

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 sections for more information related to the material
discussed in this section.

Related tasks
“Handling conditions in Debug Tool”
“Handling exceptions within expressions (C and C++ and PL/I only)” on page

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 OS/VSE 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 60 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 TO out of block, so execution control can be passed to some other point in the program.
- If no circumstances exist explicitly directing the assignment of control, your primary commands file or terminal is queried for another command.

If, after the execution of any defined breakpoint, control returns to your program with a G0, the condition is raised again in the program (if possible and still applicable). If you use a G0TO to bypass the failing statement, you also bypass your program’s error handling facilities.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

- “Language Environment conditions and their C and C++ equivalents” on page 264
- “PL/I conditions and condition handling” on page 249
  
- 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 sections for more information related to the material discussed in this section.

**Related tasks**

- “Using SET WARNING PL/I command with built-in functions” on page 254

**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 sections for more information related to the material discussed in this section.

Related tasks
- Debugging an application fully supported by Language Environment
- Using session variables across different 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 sections for more information related to the material discussed in this section.

Related tasks
- Using session variables across different languages

Related references
- z/OS Language Environment Programming Guide

Using session variables across different 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.)
<table>
<thead>
<tr>
<th>Machine attributes</th>
<th>PL/I attributes</th>
<th>C and C++ attributes</th>
<th>COBOL attributes</th>
<th>Assembler and disassembly attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>CHAR(1)</td>
<td>unsigned char</td>
<td>PICTURE X</td>
<td>DS X or DS C</td>
</tr>
<tr>
<td>byte string</td>
<td>CHAR(j)</td>
<td>unsigned char[j]</td>
<td>PICTURE X(j)</td>
<td>DS XLj or DS CLj</td>
</tr>
<tr>
<td>halfword</td>
<td>FIXED BIN(15,0)</td>
<td>signed short int</td>
<td>PICTURE S9(j=4)</td>
<td>DS H</td>
</tr>
<tr>
<td></td>
<td>FIXED BIN(31,0)</td>
<td>signed long int</td>
<td>PICTURE S9(4&lt;j≤9)</td>
<td>DS F</td>
</tr>
<tr>
<td>floating point</td>
<td>FLOAT BIN(21) or FLOAT DEC(6)</td>
<td>float</td>
<td>USAGE COMP-1</td>
<td>DS E</td>
</tr>
<tr>
<td></td>
<td>FLOAT BIN(53) or FLOAT DEC(16)</td>
<td>double</td>
<td>USAGE COMP-2</td>
<td>DS D</td>
</tr>
<tr>
<td>extended floating point</td>
<td>FLOAT BIN(109) or FLOAT DEC(33)</td>
<td>long double</td>
<td>n/a</td>
<td>DS L</td>
</tr>
<tr>
<td>fullword pointer</td>
<td>POINTER</td>
<td>*</td>
<td>USAGE POINTER</td>
<td>DS A</td>
</tr>
</tbody>
</table>

**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 sections for more information related to the material discussed in this section.

**Related references**

“Debug Tool interpretation of HLL variables and constants” on page 328
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 sections for more information related to the material discussed in this section.

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 sections for more information related to the material discussed in this section.

Related references

“Coexistence with other debuggers”
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 sections for more information related to the material discussed in this section.

Related references

z/OS Language Environment Programming Guide
Chapter 48. 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 sections for more information related to the material discussed in this section.

Related tasks

- "Starting Debug Tool within an enclave"
- "Viewing Debug Tool windows across multiple enclaves" on page 340
- "Ending a Debug Tool session within multiple enclaves" on page 340
- "Using Debug Tool commands within multiple enclaves" on page 340

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, the Source or Listing windows and their related windows (Compact Source, Local Breakpoint, and Local Monitor windows) of the first enclave are hidden. You cannot open a Source or Listing window for a compile unit unless 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 run time 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 OS/VS 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.

<table>
<thead>
<tr>
<th>Debug Tool command</th>
<th>Affects current enclave only</th>
<th>Affects entire Debug Tool session</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>%CAAADDRESS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT GLOBAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AT TERMINATION</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CLEAR AT</td>
<td>X</td>
<td>X</td>
<td>In addition to clearing breakpoints set in the current enclave, CLEAR AT can clear global breakpoints.</td>
</tr>
<tr>
<td>Debug Tool command</td>
<td>Affects current enclave only</td>
<td>Affects entire Debug Tool session</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>CLEAR DECLARE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CLEAR VARIABLES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Declarations</td>
<td>X</td>
<td>X</td>
<td>Session variables are cleared at the termination of the process in which they were declared.</td>
</tr>
<tr>
<td>DISABLE</td>
<td>X</td>
<td>X</td>
<td>In addition to disabling breakpoints set in the current enclave, DISABLE can disable global breakpoints.</td>
</tr>
<tr>
<td>ENABLE</td>
<td>X</td>
<td>X</td>
<td>In addition to enabling breakpoints set in the current enclave, ENABLE can enable global breakpoints.</td>
</tr>
<tr>
<td>LIST AT</td>
<td>X</td>
<td>X</td>
<td>In addition to listing breakpoints set in the current enclave, LIST AT can list global breakpoints.</td>
</tr>
<tr>
<td>LIST CALLS</td>
<td>X</td>
<td></td>
<td>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. <strong>Note</strong>: Only compile units in the current thread will be listed for PL/I multitasking applications.</td>
</tr>
<tr>
<td>LIST EXPRESSION</td>
<td></td>
<td>X</td>
<td>You can only list variables in the currently active thread.</td>
</tr>
<tr>
<td>LIST LAST</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LIST NAMES CUS</td>
<td>X</td>
<td></td>
<td>Applies to compile unit names. In the Debug Frame window, compile units in parent enclaves are marked as deactivated.</td>
</tr>
<tr>
<td>LIST NAMES TEST</td>
<td>X</td>
<td></td>
<td>Applies to Debug Tool session variable names.</td>
</tr>
<tr>
<td>MONITOR GLOBAL</td>
<td>X</td>
<td></td>
<td>Applies to Global monitors.</td>
</tr>
<tr>
<td>PLAYBACK ENABLE</td>
<td>X</td>
<td></td>
<td>The PLAYBACK command that informs Debug Tool to begin the recording session.</td>
</tr>
<tr>
<td>PLAYBACK DISABLE</td>
<td>X</td>
<td></td>
<td>The PLAYBACK command that informs Debug Tool to stop the recording session.</td>
</tr>
<tr>
<td>PLAYBACK START</td>
<td>X</td>
<td></td>
<td>The PLAYBACK command that suspends execution of the program and indicates to Debug Tool to enter replay mode.</td>
</tr>
<tr>
<td>PLAYBACK STOP</td>
<td>X</td>
<td></td>
<td>The PLAYBACK command that terminates replay mode and resumes normal execution of Debug Tool.</td>
</tr>
<tr>
<td>PLAYBACK BACKWARD</td>
<td>X</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Debug Tool command</td>
<td>Affects current enclave only</td>
<td>Affects entire Debug Tool session</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------</td>
<td>----------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>PLAYBACK FORWARD</td>
<td>X</td>
<td>X</td>
<td>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.</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET AUTOMONITOR¹</td>
<td>X</td>
<td></td>
<td>Controls the monitoring of data items at the currently executing statement.</td>
</tr>
<tr>
<td>SET COUNTRY²</td>
<td>X</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>SET EQUATE³</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET INTERCEPT¹</td>
<td>X</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>SET NATIONAL LANGUAGE¹</td>
<td>X</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>SET PROGRAMMING LANGUAGE¹</td>
<td>X</td>
<td></td>
<td>Applies only to programming languages in which compile units known in the current enclave are written (a language is &quot;known&quot; the first time it is entered in the application flow).</td>
</tr>
<tr>
<td>SET QUALIFY¹</td>
<td>X</td>
<td></td>
<td>Can only be issued for load modules, compile units, and blocks that are known in the current enclave.</td>
</tr>
<tr>
<td>SET TEST¹</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIGGER condition²</td>
<td>X</td>
<td></td>
<td>Applies to triggered conditions. Conditions can be either an Language Environment symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.</td>
</tr>
<tr>
<td>TRIGGER AT</td>
<td>X</td>
<td>X</td>
<td>In addition to triggering breakpoints set in the current enclave, TRIGGER AT can trigger global breakpoints.</td>
</tr>
</tbody>
</table>

**Notes:**
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 49. 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 50. 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 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.

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.
Syntax of the NAMES INCLUDE command

The NAMES INCLUDE command enables you to indicate to Debug Tool the names of load modules or compile units that you want to debug. You can use this command in remote debug mode when you use one of the following remote debuggers:

- WebSphere Developer Debugger for zSeries
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- Compiled Language Debugger component of WebSphere Developer for zSeries

```
 NAMES INCLUDE LOADMOD name

 INCLUDE
 Indicates that you want to debug the specified user load modules or compile units.

 LOADMOD
 Indicates that you want to debug the specified load module.

 CU
 Indicates that you want to debug the specified compile unit.

 name
 Specifies the name of the load module or compile unit.
```

Restrictions: The NAMES INCLUDE command has the following restrictions:

- You cannot use the NAMES INCLUDE command on load modules or compile units that are already known to Debug Tool.
- You cannot use the NAMES INCLUDE command to indicate to Debug Tool that you want to debug the initial load module or the compile units contained in the initial load module. If you want to do this, you must code control statements into the EQAOPTS Debug Tool customization module with the equivalent NAMES INCLUDE command. See “Using EQAOPTS to implement NAMES commands” on page 351 for instructions.
- 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.

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.

### Syntax of the NAMES EXCLUDE command

The NAMES EXCLUDE command enables you to indicate to Debug Tool the names of load modules or compile units that are data-only modules. You can use this command in remote debug mode when you use one of the following remote debuggers:

- WebSphere Developer Debugger for zSeries
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- Compiled Language Debugger component of WebSphere Developer for zSeries

#### NAMES EXCLUDE

```
NAMESEXCLUDE LOADMOD pattern
```

- **EXCLUDE**
  Indicating that you do **not** want to debug the specified **user** load modules or compile units.

- **LOADMOD**
  Indicates that you do not want to debug the specified load module.

- **CU**
  Indicates that you do not want to debug the specified compile unit.

- **pattern**
  Specifies the name of the load module or compile unit, or a quoted string that contains a partial load module or compile unit name followed by an asterisk to indicate that you do not want to debug all load modules or compile units beginning with the specified string.

### Restrictions

The NAMES EXCLUDE command has the following restrictions:

- You cannot use the NAMES EXCLUDE command on load modules or compile units that are already known to Debug Tool.

  If you specify the name of a currently known load module or compile unit, it is added to the exclude list so that if the name becomes unknown, it is excluded in subsequent appearances. However, the currently known load module or compile unit remains known.

- You cannot use the NAMES EXCLUDE command to indicate to Debug Tool that you want to exclude the initial load module or the compile units contained in the initial load module. If you want to do this, you must code control statements into the EQAOPTS Debug Tool customization module with the equivalent NAMES EXCLUDE command. See "Using EQAOPTS to implement NAMES commands" on page 351 for instructions.

- For C and C++ programs, the **pattern** parameter case sensitive. For all other languages, the pattern is not case sensitive.
Optimizing the debugging of large programs

Some very large programs 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 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 might be situations where Debug Tool might still require debug data. In these situations, Debug Tool loads debug data as required and these modules can be known to Debug Tool, for example, in a calling chain.

See “Syntax of the NAMES EXCLUDE command” on page 349 for a description of the NAMES EXCLUDE command.

Displaying current NAMES settings

You can use the following command to display the current settings of the NAMES command:

```
>> NAMES DISPLAY [USER | ALL | EXCLUDED] [LOADMODS | CUS] [pattern]
```

**DISPLAY**
Indicates that you want a list of all the load modules or compile units that are currently excluded or included. If you do not specify the ALL parameter, only the names excluded by user commands appear in the list that is displayed. Names that Debug Tool excludes by default not included in the list that is displayed.

**USER**
Indicates that you want a list of load modules or compile units that are currently excluded at your request (by using NAMES EXCLUDE command).

**ALL**
Indicates that you want a list of all load modules or compile units that currently excluded, including those that Debug Tool excludes by default.

**LOADMOD**
Indicates that you want a list of load module names.

**CU**
Indicates that you want a list of compile unit names.

**pattern**
Specifies the name of the load module or compile unit, or a quoted string that contains a partial load module or compile unit name followed by an asterisk to indicate that you want a list of all load modules or compile units beginning with the specified string.
Using EQAOPTS to implement NAMES commands

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 code control statements into the EQAOPTS Debug Tool customization module with the equivalent NAMES command.

EQAOPTS enables you to customize specific Debug Tool functions. See Debug Tool Customization Guide for a complete description of EQAOPTS. You generate EQAOPTS by invoking the EQAXOPT assembler macro. Code the EQAXOPT assembler macro to process a NAMES command in the following format:

EQAOPTS NAMES, positional_parameter_2, positional_parameter_3, positional_parameter_4

In this macro, the second, third, and fourth positional parameters are coded like the fields in the corresponding NAMES command. The following table describes how the Debug Tool command is coded into the equivalent EQAXOPT macro:

<table>
<thead>
<tr>
<th>Debug Tool command</th>
<th>EQAXOPT macro invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMES EXCLUDE CU &quot;ABC1*&quot;;</td>
<td>EQAXOPT NAMES, EXCLUDE, CU, ABC1*</td>
</tr>
</tbody>
</table>

You can include as many invocations of the EQAXOPT macro with the NAMES option as you need in a single EQAOPTS.
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 your source code is being managed by a library system that requires the SUBSYS=SXXXX parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=SXXXX allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

**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.

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.

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 OS/VS 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 SYSDATA output from the High Level assembler as input to the EQALANGX program.

File produced by DEBUG(FORMAT(DWARF)) compiler option
This data set is produced by the C/C++ compiler. Debug Tool supports the DEBUG compiler option in z/OS C/C++ Version 1.6 or later. When you specify the FORMAT(DWARF) suboption of the DEBUG compiler option, the debug information is stored in the file specified by the FILE suboption of the DEBUG compiler option. It can be a sequential data set, a PDS member, or a z/OS UNIX System Services file. The [z/OS XL C/C++ User’s Guide] describes how the compiler generates the name of this file and where the file is stored.

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 your source code is being managed by a library system that requires the SUBSYS=SSSS parameter when the data set is allocated, you need a custom version of the EQAOPTS options module that specifies the SUBSYS=SSSS allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

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 when you compile your program...
with the SEPARATE compiler suboption of the TEST compiler option. It should be kept in a permanent PDS member, sequential file, or HFS file. The SEPARATE compiler suboption is available on the following compilers:

- 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 Release 5

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 listing and other debug data, such as the symbol table. The DD name used by the compiler for the separate debug file is SYSDEBUG.

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.

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. It should be kept in a permanent PDS member or a sequential file.

The default DD name for the Debug Tool preferences file is INSPPREF.

Preferences files are not used in remote debug mode.

Global preferences file

This data set is similar to the preferences file, but it is specified through the EQAOPTS options load module. See the Debug Tool Customization Guide for more information on the EQAOPTS options load module. If a global preferences file exists, the commands specified in it are run before commands found in the preferences file.

Global Preferences file is not used in remote debug mode.

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. It should be kept in a permanent PDS member or a sequential file.

If a preferences file is available, the commands in the commands file are run after the commands specified in the preferences file.

Commands files are not used in remote debug mode.

Log file

Debug Tool uses this file to record the progress of the debugging session. The results of the execution of commands are saved as comments. This allows you to use the log file as a commands file in subsequent debugging sessions. It should be kept in a permanent PDS member or a sequential file. As this data set is written to by Debug Tool, we recommend you use a sequential file to relieve any contentions for this file.

The DD name for the Debug Tool log file is INSPLGC.
Log files are not used in remote debug mode.
The record format needs to be either F, FB, V, or VB.

**Save settings file**
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, this default can be changed by using EQAOPTS. In non-interactive mode (MVS batch mode without using a VTAM terminal), the DD name used to locate this file is INSPSAFE.

You can not save the setting information into the same file that you save breakpoint and monitor specifications information.

Save setting files are not used for remote debug sessions.

Automatic save and restore of the setting 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 or under IMS/DC.

**Save breakpoints and monitor specifications file**
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.DBGTOOL.SAVEBPS*. However, this default can be changed by using EQAOPTS. In non-interactive mode (MVS batch mode without using a VTAM terminal), the DD name used to locate this file is INSPPBM.

You can not save the breakpoint and monitor specifications information into the same file that you save setting 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 settings files are not supported automatically when debugging DB2 stored procedures or under IMS/DC.
Appendix B. How does Debug Tool locate debug information and source or listing files?

Debug Tool obtains information (called debug information) it needs about a compilation unit (CU) by searching through the following sources:

1. In most cases, the debug information is found in the load module. In conjunction with this information, Debug Tool uses the information in the source or listing file to display source code on the screen.

2. For C and C++ CUs compiled with the FORMAT(DWARF) suboption of the DEBUG compiler option, Debug Tool uses debug information stored in the file specified by the FILE suboption of the DEBUG compiler option.

3. 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.

4. For assembler 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. This name is either the name of the data set where the compiler processed the source, listing, or separate debug file or a name constructed from the CU name. The way this default name is generated differs depending on the source language and compiler used. See Appendix A, “Data sets used by Debug Tool,” on page 355 for a description of this default name and how it is generated for each language and compiler.

The source or listing data, separate debug file data, or EQALANGX data is obtained from one of the following sources:

- the default data set name
- the SET SOURCE command
- the SET DEFAULT LISTINGS command
- the EQADEBUG DD statement

For C and C++ programs compiled with the DEBUG(FORMAT(DWARF)) compiler option, Debug Tool needs the following files:

- The source file. 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 file specified by the FILE suboption of the DEBUG compiler option. Debug Tool obtains the file from one of the following sources:
  - the default data set name
  - the EQADEBUG DD statement

The order in which these are located is different for each type of file. For the default data set name and the SET DEFAULT LISTINGS command, the EQAUEDAT
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 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.

If your source code is being managed by a library system that requires the SUBSYS parameter when the data set is allocated, you need a custom version of the EQAOPCS options module that specifies the SUBSYS allocation parameter. This support is not available when debugging a program under CICS. See the Debug Tool Customization Guide for details.

How does Debug Tool locate COBOL and PL/I separate debug file files?

Debug Tool might read from a COBOL 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 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 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 OS/VS 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 the EQAUEDAT user exit is implemented and a
   EQADEBUG DD statement is not specified, the data set name might be
   modified by the EQAUEDAT user exit.
4. SET DEFAULT LISTINGS command. If the EQAUEDAT user exit is implemented
   and a EQADEBUG DD statement is not specified, the data set name might be
   modified by the EQAUEDAT user exit.
5. the EQADEBUG DD statement

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 source and file produced by the
DEBUG(FORMAT(DWARF)) compiler option?

To debug C and C++ programs, Debug Tool needs the source file and the file
produced by the DEBUG(FORMAT(DWARF)) compiler option.

Debug Tool reads the source 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
source file is read can change. Each time Debug Tool needs to read the source 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,
   the data set name might be modified by the EQAUEDAT user exit.
3. default data set name. If the EQAUEDAT user exit is implemented, the data set
   name might be modified by the EQAUEDAT user exit.

If your source code is being managed by a library system that requires the
SUBSYS=*ssss parameter when the data set is allocated, you need a custom version
of the EQAOPTS options module that specifies the SUBSYS=*ssss allocation
parameter. This support is not available when debugging a program under CICS.
See the Debug Tool Customization Guide for details.

Debug Tool might read from the file produced by the DEBUG(FORMAT(DWARF)
compiler option more than once but it always reads this file from the same data
set. After Debug Tool locates this file, you cannot direct Debug Tool to a different
file. When the CU first appears, Debug Tool looks for the this file in the following
order:
1. if present, the EQDEBUG DD statement
2. default data set name.
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. You must have Debug Tool Utilities and Advanced Functions installed on your system to run the steps in this task. If you do not have this product installed, 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 9. Names and attributes to use when you create your own data sets.

<table>
<thead>
<tr>
<th>Data set name</th>
<th>LRECL</th>
<th>BLKSIZE</th>
<th>RECFM</th>
<th>DSORG</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix.SAMPLE.COBOL</td>
<td>80</td>
<td>*</td>
<td>FB</td>
<td>PO</td>
</tr>
<tr>
<td>prefix.SAMPLE.PLI</td>
<td>80</td>
<td>*</td>
<td>FB</td>
<td>PO</td>
</tr>
<tr>
<td>prefix.SAMPLE.C</td>
<td>80</td>
<td>*</td>
<td>FB</td>
<td>PO</td>
</tr>
<tr>
<td>prefix.SAMPLE.ASM</td>
<td>80</td>
<td>*</td>
<td>FB</td>
<td>PO</td>
</tr>
<tr>
<td>prefix.SAMPLE.DTSF</td>
<td>1280</td>
<td>*</td>
<td>VB</td>
<td>PO</td>
</tr>
</tbody>
</table>

* 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:

<table>
<thead>
<tr>
<th>SEQASAMP member name</th>
<th>Your sample data set</th>
<th>Description of member</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQAWPP1</td>
<td>prefix.SAMPLE.COBOL(WPP1)</td>
<td>COBOL source code</td>
</tr>
<tr>
<td>EQAWPP3</td>
<td>prefix.SAMPLE.PLI(WPP3)</td>
<td>PL/I source code</td>
</tr>
<tr>
<td>EQAWPP4</td>
<td>prefix.SAMPLE.C(WPP4)</td>
<td>C source code</td>
</tr>
<tr>
<td>EQAWPP5</td>
<td>prefix.SAMPLE.ASM(WPP5)</td>
<td>Assembler source code</td>
</tr>
<tr>
<td>EQAWSU1</td>
<td>prefix.SAMPLE.DTSF(WSU1)</td>
<td>setup file for EQAWPP1</td>
</tr>
<tr>
<td>EQAWSU3</td>
<td>prefix.SAMPLE.DTSF(WSU3)</td>
<td>setup file for EQAWPP3</td>
</tr>
<tr>
<td>EQAWSU4</td>
<td>prefix.SAMPLE.DTSF(WSU4)</td>
<td>setup file for EQAWPP4</td>
</tr>
<tr>
<td>EQAWSU5</td>
<td>prefix.SAMPLE.DTSF(WSU5)</td>
<td>setup file for EQAWPP5</td>
</tr>
</tbody>
</table>
Starting Debug Tool Utilities

To start Debug Tool Utilities, do one of 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 do the steps in this task, you must have Debug Tool Utilities and Advanced Functions (5655-R45) installed on your system. 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=PL1
     - 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 on page 364:
   - 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 on page 364:
   - 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 on page 364:
   - 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 on page 364:
    - 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 \(^2\) on page 364:
   - 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 \(^2\) on page 364:
   - 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 \(^2\) on page 364:
   - 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 \(^2\) on page 364:
   - 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,SCELKED'. 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).
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 on page 364:
   - 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 on page 366 of instructions on how to link the object modules.
   b. Replace &EQAPRFX. with the prefix your EQAW (Debug Tool) library.
   c. Replace &CEEPFX. with the prefix your CEE (Language Environment) library.
   d. Enter "e" in Cmd field next to CMDS DD name. In the window that is displayed, if there is a QUIT ; statement at the end of the data set, remove it. Press PF3 (Exit).
   e. Type "run" in command line. Press Enter.
4. 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 on page 364:
   - For the COBOL program, use the following values for each field: Project = prefix, Group = SAMPLE, Type = DTSF, Member = WSU1
   - For the PL/I program, use the following values for each field: Project = prefix, Group = SAMPLE, Type = DTSF, Member = WSU3
   - For the C program, use the following values for each field: Project = prefix, Group = SAMPLE, Type = DTSF, Member = WSU4
   - For the assembler program, use the following values for each field: Project = prefix, Group = SAMPLE, Type = DTSF, Member = WSU5

   Press Enter.

5. If you ran the steps beginning on page 367 (running the program in foreground), 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 on page 366 of instructions on how to link the object modules.
   b. Replace &EQAPRFX. with the prefix your EQAW (Debug Tool) library.
   c. Replace &CEEPRFX. with the prefix your CEE (Language Environment) library.

   Enter "e" in the Cmd field next to CMDS DD name. If there is not 'QUIT ;' statement at the end of the data set, then add the statement. Press PF3 (Exit).

7. Type submit in command line. Press Enter.
8. In panel ISREDDE2, type submit in the command line. Press Enter. Make a note of the job number that is displayed.
9. In panel ISREDDE2, press PF3 (Exit).
10. In panel EQAPFORS, press PF3 (Exit). The changes you made to the setup file are saved.
11. In panel EQAPFOR, press PF3 (Exit) to return to EQA@PRIM panel. locate the job output using the job number recorded. Check for zero return code and the command log output at the end of the job output.
Appendix D. 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 sections for more information related to the material discussed in this section.

Related tasks

- “Starting Debug Tool in batch mode” on page 96
Appendix E. Notes on debugging in remote debug mode

Debug Tool can run in remote debug mode, by using TCP/IP to connect to a remote debugger installed on your workstation. The following remote debuggers can be used:

- IBM Distributed Debugger (also known as VisualAge Remote Debugger)
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- Compiled Language Debugger component of WebSphere Developer for zSeries
- WebSphere Developer Debugger for zSeries

These remote debuggers support single and multiple connection types, except for earlier versions of the IBM Distributed Debugger. If you are using a version of IBM Distributed Debugger that is earlier than version 9.2 (copyright date 2003/10/19), you must specify multiple connection type.

Specifying a single connection type is recommended because it uses less resource and avoids security problems when you use a firewall. You specify single connection type by using the TCPIP& of the TEST run-time option. You specify multiple connection type by using the VADTCPIP& suboption of the TEST run-time option.

When you specify the TCPIP& or VADTCPIP& suboption, you must specify the port number that the remote debugger is using to listen for a debug session. By default, IBM Distributed Debugger uses port 8000. By default, the following remote debuggers use port 8001:

- WebSphere Developer Debugger for zSeries
- Compiled Language Debugger component of WebSphere Studio Enterprise Developer
- Compiled Language Debugger component of WebSphere Developer for zSeries

When you use remote debug mode, consider the following possible errors:

- The tcpip_workstation_id or port_id parameters must be syntactically and functionally correct. If they are not and you try to start a remote debug mode session, Debug Tool starts a full-screen mode session. For example, if you try to start a remote debug mode session from TSO or a CICS program by using incorrect parameters, a full-screen mode session is displayed on your 3270 type terminal. This error is recorded in the MVS SDSF log as an allocation failure.
- If the tcpip_workstation_id or port_id parameters are not syntactically and functionally correct and you try to debug batch program, Debug Tool terminates and the batch program runs as though no debug session was started. This error occurs when, for example, you run a JES batch job or CICS batch transaction. This error is recorded in the MVS SDSF log as an allocation failure.
- If your z/OS environment is not using the default TCP/IP data set named TCPIP.TCPIP.DATA and you try to start a remote debug mode session to debug a batch program, Debug Tool terminates. The batch program runs as though no debug session was started. This error is recorded in the MVS SDSF log as an allocation error.

To fix this error, specify the SYSTCPD DDNAME with the appropriate TCP/IP data set name. For example,
//SYSTCPD DD DISP=SHR, DSN=MY.TCPIP.DATA

- For TCP/IP sessions, the remote debug daemon must be started at the workstation before you start Debug Tool. Refer to the remote debugger information for help on using the remote debug daemon.

## Debug Tool commands supported in remote debug mode

The following table summarizes the commands available to each remote debugger:

<table>
<thead>
<tr>
<th>Command</th>
<th>WebSphere Developer Debugger for zSeries</th>
<th>Compiled Language Debugger component of WebSphere Studio Enterprise Developer</th>
<th>Compiled Language Debugger component of WebSphere Developer for zSeries</th>
<th>IBM Distributed Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL %VER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLEAR LOAD¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DESCRIBE CUS</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DESCRIBE LOADMODS¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>LOAD¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>LOADDEBUGDATA¹ (for assembler only)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NAMES</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUERY CURRENT VIEW¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUERY DEFAULT VIEW¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUERY LDD</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET ASSEMBLER¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET AUTOMONITOR¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET DEFAULT VIEW¹</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET DISASSEMBLY</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET DYNDDEBUG</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET LDD</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET LOG</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET QUALIFY CU²</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET QUALIFY LOAD²</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SET WARNING</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Notes:**

1. You must have Debug Tool Utilities and Advanced Functions installed to use these commands.
2. You must have Debug Tool Utilities and Advanced Functions installed to use some of the features of these commands. See [Debug Tool Reference and Messages](#) for more information about these commands.

See [Debug Tool Reference and Messages](#) for more information about each command.
Tip on monitoring variables in optimized COBOL program

After you start the remote debugger and start your optimized COBOL program, do the following steps:

1. Step into your program by using the Step Into button.
2. Monitor a variable. The variable's name and current value are displayed in the Monitor window.
3. Step through your program until you reach a statement that alters the value of the variable you are monitoring. If you attempt to run the statement, a Debugger Message window displays the following message:
   Error occurred: EQA2421E The assignment was not performed because the assigned value might not be used by the program, due to optimization.
4. Enter the SET WARNING OFF command in the input line of the Command Log window. The Command Log window displays a message that the SET WARNING OFF command was received.
5. Step through the statement. A Debugger Message window displays the following message:
   Error occurred: EQA2420W The assignment was performed but the assigned value might not be used by the program, due to optimization.

The new value of the variable you are monitoring is displayed in the Monitors window.
Appendix F. Syntax of the TEST and DEBUG compiler option

Debug Tool allows you to debug applications written in C, C++, COBOL, and PL/I. In order to debug your applications, you must compile them with the TEST compiler option, or, for C and C++ programs, the DEBUG compiler option. If you do not compile your applications with the TEST or DEBUG compiler option, then you can debug them only by using the disassembly view.

This chapter describes the TEST compiler option for C, C++, COBOL, and PL/I and the DEBUG compiler option for C and C++. For more information on how to use the compiler options, refer to the chapters in Part 2, “Preparing your program for debugging,” on page 19. For a description of all the compiler options available for each programming language, see the documentation provided with each programming language compiler.

The suboptions of the TEST and DEBUG compiler options control the generation of symbol tables and program hooks that Debug Tool needs to debug your programs. The choices that you make when compiling your program affect the amount of Debug Tool function available during your debug session. When a program is under development, you should compile the program with TEST(ALL) or DEBUG(FORMAT(DWARF),HOOK(LINE,NOBLOCK,PATH),SYMBOL) compiler option to get the full capability of Debug Tool.

Refer to the following sections for more information related to the material discussed in this section.

Related tasks
- z/OS XL C/C++ User’s Guide
- z/OS XL C/C++ Programming Guide
- COBOL for OS/390 & VM Programming Guide
- Enterprise COBOL for z/OS Programming Guide
- VisualAge PL/I for OS/390 Programming Guide
- Enterprise PL/I for z/OS Programming Guide

Related references
- z/OS XL C/C++ Language Reference
- COBOL for OS/390 & VM Language Reference
- Enterprise COBOL for z/OS Language Reference
- Enterprise PL/I for z/OS Language Reference
- VisualAge PL/I Language Reference

Syntax for the C TEST compiler option

The syntax for the C TEST compiler option is:
The following list explains what is produced by each option and suboption and how Debug Tool uses the information when you debug your program:

**NOTEST**
Specifies that no debugging information is to be generated. That is, no hooks are compiled into your program, no symbol tables are created, and Debug Tool does not have access to any symbol information.
- 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 GOTO.

**TEST**
Produces debugging information for Debug Tool to use during debugging. The extent of the information provided depends on which suboptions are selected.

The following restrictions apply when using TEST:
- 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.

**BLOCK**
Inserts hooks only at block entry and exit points into your program’s object. A block is any number of data definitions, declarations, or statements enclosed within a single set of braces. BLOCK also creates hooks for nested blocks. If the SYM suboption is specified, symbol tables are generated for variables local to these nested blocks.
- You can only gain control at entry and exit points of blocks.
- Issuing a command such as STEP causes your program to run until it reaches the exit point.

**NOBLOCK**
Prevents symbol information and hooks from being generated for nested blocks.

**LINE**
Hooks are generated at most executable statements. Hooks are not generated for:
- Lines that identify blocks (lines containing braces)
- Null statements
- Labels
NOLINE
Suppresses the generation of hooks at statements (line numbers).

PATH
Hooks are generated at all path points (if-then-else, calls, etc.)
- This option does not influence the generation of hooks at entry and exit points for nested blocks. The BLOCK suboption must be specified if such hooks are desired.
- 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.
- The Debug Tool command GOTO is valid only for statements and labels coinciding with path points.

NOPATH
No hooks are generated at path points.

SYM
Generates symbol tables in the program’s object which give Debug Tool access to variables and other symbol information.
- You can reference all program variables by name, allowing you to examine them or use them in expressions.
- You can use the Debug Tool command GOTO to branch to a label (paragraph or section name).

NOSYM
Suppresses the generation of symbol tables. Debug Tool does not have access to any symbol information.
- You cannot reference program variables by name.
- You cannot use commands such as LIST or DESCRIBE to access a variable or expression.
- You cannot use commands such as CALL or GOTO to branch to another label (paragraph or section name).

ALL
Hooks are inserted and a symbol table is generated. Hooks are generated at all statements, all path points (if-then-else, calls, and so on), and at all function entry and exit points.
ALL is equivalent to TEST(LINE, BLOCK, PATH, SYM).

NONE
Generates hooks only at function entry and exit points. Hooks at block and line points are not inserted, and the symbol tables are suppressed.
TEST(NONE) is equivalent to TEST(NOLINE, NOBLOCK, NOPATH, NOSYM).

Syntax for the C++ TEST compiler option
The syntax for the C++ TEST compiler option is:

```
/test (1)

HOOK NOHOOK
```
Notes:
1 The HOOK and NOHOOK options are available only with OS/390 C/C++, Version 2 Release 4, or later.

The following list explains what is produced by each option and how Debug Tool uses them when debugging your program:

**NOTEST**
Specifies that no debugging information is to be generated. That is, no hooks are compiled into your program, no symbol tables are created, and Debug Tool does not have access to any symbol information.
- 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 GOTO.

**TEST**
Produces debugging information for Debug Tool to use during debugging. The following restrictions apply when using the TEST option:
- 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.

**HOOK**
Generates some or all possible hook information, depending on NOOPT or OPT. This option is only available on Version 2, Release 4 C and C++ or later.

**NOHOOK**
No hook information is generated. This option is available only with OS/390 C/C++, Version 2 Release 4, or later.

---

**Syntax for the C and C++ DEBUG compiler option**

The z/OS XL C/C++ User’s Guide describes all the suboptions of the DEBUG compiler option and how to use them. If you compile your programs with the DEBUG compiler option, use the following combination of suboptions to obtain maximum Debug Tool functionality:

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

The compiler generates a file that stores debug information. Debug Tool needs this file and the source file to debug your program. If you omit the FILE suboption, the compiler creates the file in a default location. See z/OS XL C/C++ User’s Guide for more information.

```c
DEBUG(FORMAT(ISD), HOOK(LINE, NOBLOCK, PATH), SYMBOL)
```

The compiler stores debug information in the load module.

---

**Syntax for the COBOL TEST compiler option**

The syntax for the COBOL TEST compiler option is:
Notes:

1. The SEPARATE and NOSEPARATE suboptions are available for programs compiled only with Enterprise COBOL for z/OS and OS/390 or COBOL for OS/390 & VM compilers.

The TEST compiler suboptions control the production of debugging information such as symbol tables and hooks that Debug Tool needs to debug your program. The suboptions you choose can affect the size of your load module:

- To get the full capabilities of Debug Tool, compile your program with TEST(ALL,SYM).
- To get most of the capabilities of Debug Tool and a smaller load module, compile your programs with TEST(NONE,SYM,SEPARATE). You can then use the Dynamic Debug facility to debug your program.

If you compile your program with the TEST or NOTEST options and their corresponding suboptions, Debug Tool can do the following:

**NOTEST**

Specifies that no debug information is to be generated: no hooks are compiled into your program, no symbol tables are created, and Debug Tool does not have access to any symbol information. Using NOTEST produces the following results:

- 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 can not 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.

**TEST**

Produces debugging information for Debug Tool to use during debugging. The extent of the information provided depends on which of the following suboptions are selected.

**ALL**

Generates all hooks, including hooks at all statement, path, date
processing, and program entry and exit points. If you use this suboption, the size of your program increases significantly.

- The COBOL compiler only generates hooks for date processing statements 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 path points, and step through your program.
- Debug Tool can gain control of the program at all statements, path points, date processing statements, labels, and block entry and exit points, allowing you to enter Debug Tool commands.
- Branching to statements and labels using the Debug Tool command GOTO is allowed.

**BLOCK**

Hooks are inserted at all block entry and exit points.

- 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.

**NONE**

No hooks are inserted in the program.

- You can use the GOTO command only if the program is compiled with one of the following compilers:
  - Enterprise COBOL for z/OS and OS/390
  - COBOL for OS/390 & VM
- A call to CEETEST can be used at any point to start Debug Tool.

**PATH**

Hooks are inserted at all path points and at all program entry and exit points. A path point is anywhere in a program where the logic flow is not necessarily sequential or can change. Some examples of path points are IF-THEN-ELSE constructs, PERFORM loops, ON SIZE ERROR phrases, and CALL statements.

- 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.

**STMT**

Hooks are inserted at every statement and label, at every date processing statement, and at all entry and exit points.

- 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.

SYM
Generates symbol tables in the program’s object that give Debug Tool access to variables and other symbol information.

- You can reference all program variables by name, which allows you to examine them or use them in expressions.
- You can use the PLAYBACK ENABLE command with the DATA parameter.
- You can use the SET AUTOMONITOR ON command.
- SYM is required to support labels (paragraph or section names) as GOTO targets.

SEPARATE
Saves the symbolic table information in a separate debug file. This suboption is available only if you compile with the following compilers:

- 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

NOSYMP
Suppresses the generation of dictionary tables. Debug Tool does not have access to any symbol information. Using NOSYM produces the following results:

- 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).

Specifying TEST with no suboptions is equivalent to TEST(ALL, SYM, NOSEPARATE).

Refer to the following sections for more information related to the material discussed in this section.

Related references
- Debug Tool Reference and Messages
- Enterprise COBOL for z/OS Programming Guide
- COBOL for OS/390 & VM Programming Guide
Syntax for the PL/I and Enterprise PL/I TEST compiler options

The syntax for the PL/I TEST compiler option is:

```
/SM590000
NOTEST
```

The syntax for the Enterprise PL/I TEST compiler option is:

```
/SM590000
/SM630000
NOTEST,
ALL
TEST
(STMT
PATH
BLOCK
NONE
SYM
NOSYM
HOOK
NOHOOK
NOSEPARATE
SEPARATE)
```

The following list explains each option and suboption and the capabilities of Debug Tool when your program is compiled using these options:

**NOTEST**

Specifies that no debugging information is generated: no hooks are compiled into your program, no dictionary tables are created, and Debug Tool does not have access to any symbol information. Using NOTEST produces the following results:

- 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.
**TEST**

Produces debugging information for Debug Tool to use during debugging. The extent of the information provided depends on which of the following suboptions are selected:

**ALL**
Generates all compiled-in hooks, including all statement, path, and program entry and exit hooks.
- 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.

**BLOCK**
Hooks are inserted at all block entry and exit points.
- Enables Debug Tool to gain control at block boundaries: block entry and block exit.
- You can gain control only at entry and exit points of your program and all entry and exit points of internal program blocks.
- A call to PLTEST or CEETEST can be used to start Debug Tool at any point in your program.
- Issuing a command such as STEP causes your program to run until it reaches the next block entry or exit point.
- Hooks are not inserted into an empty ON-unit or an ON-unit consisting of a single GOTO statement.

**NONE**
No hooks are inserted in the program.
- A call to PLTEST or CEETEST can be used to start Debug Tool at any point in your program.

**PATH**
Hooks are inserted at all path points:
- Before the THEN part of an IF statement.
- Before the ELSE part of an IF statement.
- Before the first statement of all WHEN clauses of a SELECT-group.
- Before the OTHERWISE statement of a SELECT-group.
- At the end of a repetitive DO statement, just before the Do-group is to be executed.
- At every CALL or function reference, both before and after control is passed to the routine.
- Before the statement following a user label, excluding labeled FORMAT statements. If a statement has multiple labels, only one hook is inserted.

Specifying **PATH** also causes **BLOCK** hooks to be inserted.

**STMT**
Hooks are inserted before most executable statements and labels. **STMT** also causes **BLOCK** hooks to be inserted.
- 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.

**SYM**
Generates a symbol table in the program’s object file. The symbol table is required for examining program variables or program control constants by name.

- 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**
Suppresses the generation of a symbol table. Debug Tool does not have access to any symbol information that causes the following results:

- 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).

**HOOK (Enterprise PL/I for z/OS, Version 3 Release 4, or later)**
Causes the compiler to insert hooks into the generated code if any of the TEST suboptions ALL, STMT, BLOCK or PATH are in effect.

**NOHOOK (Enterprise PL/I for z/OS, Version 3 Release 4, or later)**
Causes the compiler not to insert hooks in to the generated code.

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.

**SEPARATE (Enterprise PL/I for z/OS, Version 3 Release 5, only)**
Causes the compiler to place most of the debug information it generates into a separate debug file. Using this option will substantially reduce the size of the object deck created by the compiler when the TEST option is in effect.

If your program contains GET or PUT DATA statements, the separate debug file will contain less debug information since those statements require that symbol table information be placed into the object deck.

The generated debug information will always include a compressed version of the source passed to the compiler. This means that source may be specified through SYSIN DD * or that the source may be a temporary data set created by an earlier job step (for example, the source may be the output of the SQL or CICS precompilers).

If this suboption is used in a batch compile, then the JCL for that compile must include a DD card for SYSDEBUG which must name a data set with RECFM=FB and with 80 <= LRECL <= 1024.

This suboption may not be used with the LINEDIR compiler option.
**NOSEPARATE (Enterprise PL/I for z/OS, Version 3 Release 5, only)**

Causes the compiler to place all of the debug information it generates into the object deck.

Under this option, the generated debug information will not include a compressed version of the source passed to the compiler. This means that source must in a data set that can be found by Debug Tool when you try to debug the program.

Compiling with TEST(STMT), TEST(PATH), or TEST(ALL) causes a statement number table to be generated. If the STMT compiler option is in effect, specifying TEST causes the GOSTMT compiler option to be in effect. If the NUMBER compiler option is in effect, specifying TEST causes the GONUMBER compiler option to be in effect.

Refer to the following sections for more information related to the material discussed in this section.

**Related references**

- *PL/I for MVS and VM Programming Guide*
- *VisualAge PL/I for OS/390 Programming Guide*
- *Enterprise PL/I for z/OS Programming Guide*
Appendix G. 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. To use the Load Module Analyzer, you must purchase and install Debug Tool Utilities and Advanced Functions Version 7.1 (5655-R45).

Choosing a starting method

You can start the Load Module Analyzer in one of the following ways:
- Editing sample JCL provided in member EQAZLMA of data set hlq.SEQASAMP, then submitting the JCL to run as a batch job.
- By selecting option 5 of 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

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.

Description of DD names

Load Module Analyzer uses the following DD names:

**EQALIB**

Specifies a concatenation of PDS or PDSE data sets containing the load
modules or program objects to be analyzed. If the same member is present in more than one of the concatenated data sets, only the first member is processed.

**EQAPRINT**

Specifies the output report. It can be in fixed block record format (RECFM=FBA) with a logical record length of 133 or more (LRECL >=133) or in variable block record format (RECFM=VBA) with a logical record length of 137 or more (LRECL >= 137).

**EQAIN**

Specifies the control statements. If you want only specific load modules or program objects to be processed, use the following syntax:

```
SELECT MEMBER=load_module_name
```

If you want all load modules to be processed, you can omit this DD statement, direct it to DUMMY, or direct it to empty data set. This file must be in fixed block record format (RECFM=FB) with a logical record length of 80 (LRECL=80). Each control statement must be on a separate line. The entries are free-form and you can use blanks before or after each keyword and operator. You can include comments by placing an asterisk in column 1.

**EQASYSPF**

Specifies a list of system prefixes. This is a list of prefixes of names of CSECTs that you want Load Module Analyzer to recognize as system routines. The list helps limit the amount of output displayed for these prefixes. This file must be in fixed block record format (RECFM=FB) with a logical record length of 80 (LRECL=80). Debug Tool provides data for this file in member EQALMPFX of the table library (SEQATLIB). See [“Description of EQASYSPF file format” on page 390](#) 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 EOALMPGM of the table library (EQATLIB). See [“Description of EQAPGMNM file format” on page 391](#) for directions on how to add entries to this list.

### Description of parameters

You can specify parameters by using the PARM= keyword of the EXEC JCL statement. The parameter string passed to this program can consist of any of the following parameters, separated by commas or blanks:

**CKVOLFPRS**

Lists only CSECTs or entries that use at least one of the Additional Floating-Point Registers 8 through 15. You cannot specify this parameter with the OSVSONLY parameter. If you specify both, the last one specified is used.

**DATEFMT=** 

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.

**LESCAN**
  Causes the actions described under the LEINFO parameter. In addition, the text for each CSECT is scanned looking for “hidden” Language Environment entry points that do not correspond to an external symbol. For example, these might be present for C static functions. If such “hidden” entry points are detected, the same output as described for LEINFO in generated.

**LISTLD**
  Lists all label definition (LD) entries in addition to CSECT names.

**LOUD**
  Specifies that the data read from the EQASYSPF and EQAPGMNM files is displayed in the output listing.

**NATLANG=language_code**
  Specifies the national language. Use one of the following values:

  **ENU**
  Mixed-case English. (Default)

  **UEN**
  Upper-case English.

  **JPN**
  Japanese.

  **KOR**
  Korean.

**OSVSONLY**
  Specifies that only CSECTs compiled with the OS/VS COBOL compiler are to be displayed in the output. Information about all other CSECTs is suppressed.

  You cannot specify this parameter with the CKVOLFPRS parameter. If you specify both, the last one specified is used.

**SHOWLIB**
  Specifies that the include indicator in the EQASYSPF file is to be ignored so that all CSECTs are listed.

**SORTBY=sort_option**
  Specifies how to sort the names of the CSECTs in the output. Use one of the following values:

  **OFFSET**
  Sort by offset; the order shown in the linkage editor or AMBLIST output. (Default)
NAME
  Sorts by CSECT name.

PROGRAM
  Sort by the translator program ID.

LANGUAGE
  Sorts by the source language and by the translator program ID.

DATE
  Sorts by the translation date.

Description of EQASYSPF file format
This file contains a list of system prefixes. When Load Module Analyzer finds a
CSECT that has a name prefixed by a name in this list and the entry for that prefix
indicates that names beginning with that prefix are not to be included, Load
Module Analyzer does not display an individual entry for that CSECT. Instead, a
single line is displayed in the output for each prefix found that indicates that one
or more CSECTs with the specified prefix was found.

Debug Tool supplies data for this file in member EQALMPFX of the table library
(SEQATLIB). If you want to add entries to this file, create a data set containing the
new entries. Then, concatenate this data set to the one that ships with Debug Tool.
Concatenating the data sets prevents your entries from being deleted if an updated
table member is shipped in future releases of 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
    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
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, create a data set containing the new entries. Then, concatenate this data set to the one that ships with Debug Tool. Concatenating the data sets prevents your entries from being deleted if an updated table members is shipped in future releases of 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 EQASYSPF file format” on page 390 for a list of possible values.

`program_description`
A description of the program.

Description of program output

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. Prior to 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

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 CKVOLFPRS:
  - If a Language Environment prologue was detected, information is included in a string identified by LEINFO=(.... This string contains the Language Environment entry name or an asterisk to indicate that the name is the same as the external symbol, Language Environment linkage type, source language, and translation date, time, and translator version.
  - If no Language Environment prologue was detected, but the prologue appears to be that of a known, non-Language Environment compiler, one of the following is included: C/C++, COBOL, or PL/I.
  - Otherwise, ASSEMBLER is included to indicate that the program is likely to be an assembler program.

Example: Output for an OS/VS COBOL load module

The following is a fragment of output that might appear for an OS/VS COBOL load module:

<table>
<thead>
<tr>
<th>CSECT</th>
<th>Sg</th>
<th>Offset</th>
<th>Length</th>
<th>Program-ID</th>
<th>Trn-Date</th>
<th>Program-Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPRIV000010</td>
<td>28</td>
<td>C5B</td>
<td>5688216</td>
<td>1996/12/31</td>
<td>AD/Cycle C/370</td>
<td></td>
</tr>
<tr>
<td>$PRIV000011</td>
<td>D00</td>
<td>C80</td>
<td>5688216</td>
<td>1996/12/31</td>
<td>AD/Cycle C/370</td>
<td></td>
</tr>
<tr>
<td>00XINIT0</td>
<td>29E8</td>
<td>0</td>
<td>5688216</td>
<td>1995/08/03</td>
<td>ASSEMBLER H Version 1 Release 2, 3, OR 4</td>
<td></td>
</tr>
<tr>
<td>00C2CBL</td>
<td>3118</td>
<td>10</td>
<td>569623400</td>
<td>1996/12/31</td>
<td>High Level Assembler for MVS &amp; VM &amp; VSE Version 1</td>
<td></td>
</tr>
<tr>
<td>00FETCH</td>
<td>3138</td>
<td>10</td>
<td>569623400</td>
<td>1996/12/31</td>
<td>High Level Assembler for MVS &amp; VM &amp; VSE Version 1</td>
<td></td>
</tr>
<tr>
<td>MEMSET</td>
<td>3148</td>
<td>10</td>
<td>569623400</td>
<td>1996/12/31</td>
<td>High Level Assembler for MVS &amp; VM &amp; VSE Version 1</td>
<td></td>
</tr>
<tr>
<td>FPRINTF</td>
<td>3158</td>
<td>10</td>
<td>569623400</td>
<td>1996/12/31</td>
<td>High Level Assembler for MVS &amp; VM &amp; VSE Version 1</td>
<td></td>
</tr>
<tr>
<td>CS9403</td>
<td>316B</td>
<td>351B</td>
<td>566895807</td>
<td>1995/12/31</td>
<td>VS COBOL II Version 1 Release 3</td>
<td></td>
</tr>
<tr>
<td>STRLEN</td>
<td>7390</td>
<td>10</td>
<td>569623400</td>
<td>1995/08/03</td>
<td>High Level Assembler for MVS &amp; VM &amp; VSE Version 1</td>
<td></td>
</tr>
<tr>
<td>CEE*</td>
<td>5668962</td>
<td>(Multiple program ID's)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFF*</td>
<td>5668962</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDC*</td>
<td>5668962</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IGZ*</td>
<td>5668962</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 5655-R45 Debug Tool Version 7 Release 1.2 Load Module Analyzer 2006/08/28 Page 15
Appendix H. 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

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 formats of this document is accessible to visually impaired individuals who use a screen reader:

- PDF format when viewed with Adobe Acrobat Reader 5.0 or later
- BookManager® format when viewed with IBM BookManager BookServer (except for syntax diagrams)

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.

For BookManager users only: A screen reader might say the lines, symbols, and words in a diagram, but not in a meaningful way. For example, you might hear "question question dash dash MOVE dash dash plus dash dash literal-1 dash dash plus" for part of the MOVE statement. You can enter Say Next Paragraph to move quickly through syntax diagrams if your screen reader has that capability.
Appendix I. Support information

If you have a problem with your IBM software, you want to resolve it quickly. This section describes the following options for obtaining support for IBM software products:

- “Searching knowledge bases”
- “Obtaining fixes”
- “Receiving weekly support updates” on page 396
- “Contacting IBM Software Support” on page 396

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

IBM provides this documentation in an information center. You can use the search function of the information center to query conceptual information, instructions for completing tasks, and reference information.

Searching the Internet

If you cannot find an answer to your question in the information center, search the Internet for the latest, most complete information that might help you resolve your problem.

To search multiple Internet resources for your product, use the Web search topic in your information center. In the navigation frame, click Troubleshooting and support ▶ Searching knowledge bases and select Web search. From this topic, you can search a variety of resources, including the following:

- IBM technotes
- IBM downloads
- IBM Redbooks®
- IBM developerWorks®
- Forums and newsgroups
- Google

Obtaining fixes

A product fix might be available to resolve your problem. To determine what fixes are available for your IBM software product, follow these steps:

2. Click Downloads and drivers in the Support topics section.
3. Select the Software category.
4. Select a product in the Sub-category list.
5. In the Find downloads and drivers by product section, select one software category from the Category list.
6. Select one product from the Sub-category list.
7. Type more search terms in the Search within results if you want to refine your search.
8. Click Search.
9. From the list of downloads returned by your search, click the name of a fix to read the description of the fix and to optionally download the fix.

For more information about the types of fixes that are available, see the IBM Software Support Handbook at [http://techsupport.services.ibm.com/guides/handbook.html](http://techsupport.services.ibm.com/guides/handbook.html)

**Receiving weekly support updates**

To receive weekly e-mail notifications about fixes and other software support news, follow these steps:

2. Click My support in the upper right corner of the page.
3. If you have already registered for My support, sign in and skip to the next step. If you have not registered, click register now. Complete the registration form using your e-mail address as your IBM ID and click Submit.
4. Click Edit profile.
5. In the Products list, select Software. A second list is displayed.
6. In the second list, select a product segment, for example, Application servers. A third list is displayed.
7. In the third list, select a product sub-segment, for example, Distributed Application & Web Servers. A list of applicable products is displayed.
8. Select the products for which you want to receive updates, for example, IBM HTTP Server and WebSphere Application Server.
9. Click Add products.
10. After selecting all products that are of interest to you, click Subscribe to email on the Edit profile tab.
11. Select Please send these documents by weekly email.
12. Update your e-mail address as needed.
13. In the Documents list, select Software.
14. Select the types of documents that you want to receive information about.
15. Click Update.

If you experience problems with the My support feature, you can obtain help in one of the following ways:

**Online**
Send an e-mail message to erchelp@ca.ibm.com, describing your problem.

**By phone**
Call 1-800-IBM-4You (1-800-426-4968).

**Contacting IBM Software Support**

IBM Software Support provides assistance with product defects.
Before contacting IBM Software Support, your company must have an active IBM software maintenance contract, and you must be authorized to submit problems to IBM. The type of software maintenance contract that you need depends on the type of product you have:

- 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](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 IBM Software Support Web site at [http://techsupport.services.ibm.com/guides/contacts.html](http://techsupport.services.ibm.com/guides/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 [https://techsupport.services.ibm.com/ssr/login](https://techsupport.services.ibm.com/ssr/login)


- 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](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://techsupport.services.ibm.com/guides/contacts.html](http://techsupport.services.ibm.com/guides/contacts.html) and click the name of your geographic region for phone numbers of people who provide support for your location.

To contact IBM Software support, follow these steps:

1. “Determining the business impact”
2. “Describing problems and gathering information” on page 398
3. “Submitting problems” on page 398

**Determining the business impact**

When you report a problem to IBM, you are asked 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.

Describing problems and gathering information
When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these 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.

Submitting problems
You can submit your problem to IBM Software Support in one of two ways:

Online
Click Submit and track problems on the IBM Software Support site at [http://www.ibm.com/software/support/probsub.html](http://www.ibm.com/software/support/probsub.html). Type your information into the appropriate problem submission form.

By phone
For the phone number to call in your country, go to the contacts page of the IBM Software Support Handbook at [http://techsupport.services.ibm.com/guides/contacts.html](http://techsupport.services.ibm.com/guides/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 Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.
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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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

The following terms, denoted by an asterisk (*) on the first occurrence in this publication, are trademarks or service marks of International Business Machines Corporation in the United States or other countries:

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

LINUX is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT® are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a trademark of UNIX System Laboratories, licensed exclusively by X/Open Company, Ltd.
Bibliography

**Debug Tool publications**

Using CODE/370 with VS COBOL II and OS PL/I, SC09-1862

Debug Tool for z/OS

- *Debug Tool Summary of Commands and Built-in Functions*, SC19-1074
- *Debug Tool Customization Guide*, SC19-1075
- Program Directory for Debug Tool for z/OS, GI10-8726
- *Debug Tool Reference and Messages*, GC19-1073

Debug Tool Utilities and Advanced Functions for z/OS

- *Debug Tool Coverage Utility Users Guide*, SC19-1072
- Program Directory for Debug Tool Utilities and Advanced Functions for z/OS, GI10-8727

**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

Enterprise COBOL for z/OS and OS/390

- *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 and OS/390**

- *Diagnosis Guide*, 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 & VM**

- *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

© Copyright IBM Corp. 1992, 2006
Online publications can also be downloaded from the IBM Web site. Visit the IBM Web site for each product to find online publications for that product.

Softcopy publications

Online publications are distributed on CD-ROMs and can be ordered through your IBM representative. Messages and Codes, GC18-7422, Reference for Remote DRDA* Requesters and Servers, SC18-7424, Release Planning Guide, SC18-7425, SQL Reference, SC18-7426, Utility Guide and Reference, SC18-7427 are distributed on the following collection kit:

SK3T-4269
Glossary

This glossary defines technical terms and abbreviations used in [Debug Tool User's Guide](http://www.ibm.com/ibm/terminology) 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-YYYYDDDD
  - YEAR-TO-YYYY
  - YEARWINDOW
- The conceptual data items DATE and DAY in the ACCEPT FROM DATE and ACCEPT FROM DAY statements, respectively.
- The result of certain arithmetic operations.

The term date field refers to both expanded date field and windowed date field. See also nondate.

**date processing statement.** A COBOL statement that references a date field, or an EVALUATE or SEARCH statement WHEN phrase that references a date field.

**DBCS.** See double-byte character set.

**debug.** To detect, diagnose, and eliminate errors in programs.

**DTCN.** 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.

data set.

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).

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.

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.”

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). (1) 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. (2) 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)**" on page 404.
Index

Special characters

%_ctest() function 93
%CONDITION variable for PL/I 249
%PATHCODE variable for C and C++ 260
#pragma 41
specifying TEST compiler option 41
specifying TEST run-time option with 84

A
ABEND 4038 340
abnormal end of application, setting breakpoint at 331
accessing PL/I program variables 251
ALLOCATE command managing file allocations 150
allocating example of 96
applications 315
assembler
assembling your program, requirements for 48
assigning values to variables 231, 259
AT commands
AT CALL
breakpoints, for C++ 276
AT ENTRY
breakpoints, for C++ 275
AT EXIT
breakpoints, for C++ 275
attention interrupt
effect of during Dynamic Debug 152
effect of during interactive sessions 152
how to initiate 152
required Language Environment run-time options 152
attributes of variables 334
automatic saving and restoring 142
automatic saving and restoring, disabling 143

B
batch mode 80
debugging DB2 programs in 299
debugging IMS programs in 306
description of 5
for non-Language Environment programs 98
starting Debug Tool in 96
using Debug Tool in 369
binding DB2 programs 53
blanks, significance of 226
blocks and block identifiers using, for C 270
breakpoint
clearing 16
implicit 80
setting, introduction to 14
skipping 16
using DISABLE and ENABLE 16
breakpoints
before calling a NULL function in C 190
in C++ 202
before calling an invalid program, in COBOL 164
before calling an undefined program, in PL/I 180
halting if a condition is true in C 185
in C++ 196
in COBOL 160
in OS/VS COBOL 171
in PL/I 177
halting when certain COBOL routines are called 158
halting when certain functions are called
in C 184
breakpoints (continued)
halting when certain functions are called (continued)
in C++ 195
in PL/I 176
halting when certain OS/VS COBOL routines are called 170
recording, using SET AUTOMONITOR 137
setting a line 137
setting, in C++ 275

C
C
debugging a program in full-screen mode
calling a C function from Debug Tool 187
capturing output to stdout 186
debugging a DLL 187
displaying raw storage 187
displaying strings 187
finding storage overwrite errors 189
finding uninitialized storage errors 189
getting a function traceback 188
halting on line if condition true 185
halting when certain functions are called 184
modifying value of variable 184
setting breakpoint to halt 190
tracing run-time path for code compiled with TEST 188
when not all parts compiled with TEST 186
sample program for debugging 181
text of TEST compiler option 375
C and C++
AT ENTRY/EXIT breakpoints 275
blocks and block identifiers 270
calls
commands
summary 257
equivalents for Language Environment conditions 264
function calls for 262
notes on using 224
reserved keywords 263
C/C++ file produced by DEBUG(FORMAT(DWARF)), how
Debug Tool locates 361
C/C++ source files, how Debug Tool locates 361
C++
AT CALL breakpoints 276
debugging a program in full-screen mode
calling a C++ function from Debug Tool 198

© Copyright IBM Corp. 1992, 2006
C++ (continued)
calling a program in full-screen mode (continued)
capturing output to stdout 198
debbuging a DLL 199
displaying raw storage 199
displaying strings 199
finding storage overwrite errors 200
finding uninitialized storage errors 201
getting a function trace over 199
halting on an if condition true 196
modifying value of variable 195
setting a breakpoint to halt 195, 202
tracing the run-time path 200
viewing and modifying data members 197
when not all parts compiled with TEST 197
examing objects 277
overcaded operator 275
sample program for debugging 191
setting breakpoints 275
stepping through C++ programs 275
syntax of TEST compiler option 377
template in C++ 275
CADD
how to use 63
overview 106
CAF (call access facility), using to start DB2 program 300
call access facility (CAF), using to start DB2 program 300
capturing output to stdout in C 186
in C++ 198
CCCA 27
CEEBXITA
See EQADDCXT
CETEST
description 86
examples, for C 88
examples, for COBOL 89
examples, for PL/I 90
Starting Debug Tool with 85
using 306
CCEUOPT run-time options module 52
CCEUOPT to start Debug Tool under CICS, using 106
calling the value of a variable, introduction to 16
changing window layout in the session panel 214
character set 223
characters, searching 131
CICS
debbuging modes under 511
pseudo-conversational program 312
requirements for using Debug Tool in 311
restoring breakpoints 312
restrictions for debugging 313
saving breakpoints 312
starting Debug Tool under 104
CICS (continued)
starting the log file 313
CICS debugging
RLIM processing 313
CICS, starting Debug Tool under 103
closing automonitor section of Monitor window 146
closing Debug Tool session panel windows 215
COBOL 155
CCCA 27
column format 229
debbuging a program in full-screen mode
capturing I/O to system console 161
displaying raw storage 162
finding storage overwrite errors 164
generating a run-time paragraph trace 163
modifying the value of a variable 159
setting a breakpoint to halt 158
setting breakpoint to halt 164
stopping on line if condition true 160
tracing the run-time path 162
when not all parts compiled with TEST 161, 171
debbuging COBOL classes 239
debbuging VS COBOL II programs 240
finding listing 240
FACTORY 239
Load Module Analyzer 27
note on using H constant 227
notes on using 224
OBJECT 239
optimized programs, debugging 319
paragraph trace, generating a COBOL run-time 163
preparing, programs to debug 25
reserved keywords 230
restrictions on accessing, data 140
run-time options 83
sample program for debbuging 155, 167
separate debug file 25
syntax of TEST compiler option 358
variables, using with Debug Tool 231
COBOL listing, data set 355
coexistence of Debug Tool with other debbugers 336
coexistence with unsupported HLL modules 336
colors
changing in session panel 216
command
syntax diagrams xvi
command format for COBOL 229
command line, Debug Tool 125
command sequencing, full-screen mode 126
collections
abbreviating 224
commands (continued)
DTSU, using to debug DB2 program 300
for C and C++, Debug Tool subset 257
for PL/I, Debug Tool subset 247
getting online help for 227
interpreter subset description 328
list of Debug Tool Advanced Functions 7
multiline 225
PLAYBACK 17
prefix, using in Debug Tool 127
truncating 224
TSO, using to debug DB2 program 300
commands, Debug Tool
COBOL compiler options in effect 230
entering on the session panel 125
entering using program function keys 128
order of processing 126
retrieving with RETRIEVE command 129
that resemble COBOL statements 229
comments, inserting into command stream 226
common parameters, when to use 10
compile unit 126
general description 329
name area, Debug Tool 126
qualification of, for C and C++ 272
compiling a C program on an HFS file system 40
a C++ program on an HFS file system 45
an OS/VS COBOL program 27
considerations 21
Enterprise PL/I program on HFS file system 34
IMS programs 65
OS PL/I 35
PL/I for MVS & VM programs 35
programs, introduction to 11
condition handling of 249, 332
Language Environment, C and C++ equivalents 264
considerations before compiling and debugging 21
when using the TEST run-time option 79
constants
Debug Tool interpretation of HLL 328
entering 227
HLL 328
PL/I 252
debug session (continued)
starting your program 109

Debug Tool
C and C++ commands, interpretive subset 257
COBOL commands, interpretive subset 229
commands, subset 328
condition handling 332
data sets 355
enhancing performance of 52
evaluation of HLL expressions 327
exception handling, for C and C++
and PL/I 333
interfaces 4
interpretation of HLL variables 328
list of supported compilers 3
list of supported subsystems 3
multilanguage programs, using 333
PL/I commands, interpretive subset 247
starting at different points 80
starting under CICS 103, 104
starting under IMS 306
starting under MVS in TSO 95
starting your program with 109
starting by using Debug Tool Utilities 75
stopping, session 18
terminology xx
using in batch mode 369
using in remote debug mode 371
Debug Tool Advanced Functions, list of 7
Debug Tool Setup Utility 75

Debug Tool Utilities
brief description of Load Module Analyzer 9
brief description on preparing assembler 9
creating and managing setup files 8
creating private message region for IMS program 66
creating setup file for IMS program 66
how to use, to link-edit 50
overview of code coverage tasks 9
overview of IMS program preparation tasks 9
overview of program preparation tasks 8
preparing an IMS program by using 65
specifying TEST run-time options for IMS program 66
starting your program 78
using to assemble and create 49

Debug Tool Utilities and Advanced Functions
introduction 7
tasks it helps you with 7
Debug Tool Utilities, list of 7
debbugers, coexistence with other 336
debugging
CICS programs 311
COBOL classes 239
considerations 21
db2 (continued)
debugging 299
DB2 programs 299
DB2 stored procedures 303
DLL in C 187
in C++ 199
IMS programs 305
in full-screen mode 117
ISPF applications 315
multithreading programs 337
non-Language Environment programs 323
optimized programs, prevention 26
UNIX System Services programs 321
declaring session variables for C 260
for COBOL 234
defering an LDD command 170
defered, description of 207
DESCRIBE ALLOCATIONS command managing file allocations 150
DESCRIBE command using 272
destructor, stepping through 275
diagnostics, expression, for C and C++ 265
disassembly changing program in disassembly view 294
differences between SET ASSEMBLER and SET DISASSEMBLY 281, 291
displaying registers 294
displaying storage 294
modifying registers 294
modifying storage 294
performing single-step operations 293
restrictions on what you can debug 294
self-modifying code, restrictions 293
setting breakpoints 293
what you can do is disassembly view 291
disassembly view, description of 292
disassembly view, how to start 292
displaying environment information 272
halted location 134
lines at top of window, Debug Tool 131
raw storage in C 187
in C++ 199
in COBOL 162
in PL/I 178
source or listing file in full-screen mode 124
strings in C 187
in C++ 199
value of variable one time 144
values of COBOL variables 232
variable value 144
variables or storage in OS/VS COBOL 170
displaying prefixes 347
displaying the value of a variable, introduction to 15
displaying variable value
  See LIST commands
DDE debugging
  in C 187
  in C++ 199
documents, licensed xi
DOWN, SCROLL command 130
DTCN
creating a profile 58
data entry verification 62
description 104
modifying Language Environment options 62
overview 105
preparing to start Debug Tool 57
using repository profile items 105
DTSU
  See Debug Tool Setup Utility
dual terminal mode (CICS) 311
Dynamic Debug
activating 22, 23
attention interrupts, support for 152

E
editing
  setup file using Debug Tool Setup Utility
  75
  elements, unsupported, for PL/I 254
e nclave
  multiple, debugging interlanguage communication application in 345
  starting 339
ending
  debug session 153
  Debug Tool within multiple enclaves 340
entering
  commands on session panel 125
  file allocation statements into setup file 76
  program parameters into setup file 76
  run-time option into setup file 76
entering multiline commands without continuation 226
entering PL/I statements, freeform 250
Enterprise PL/I
  restrictions 255
Enterprise PL/I, definition of xvi
EQADCCXT 57
EQADCCXT user exit 81
EQADTCN2 63
EQALANGX
  creating 29
EQALANGX file
  how to create 48
EQALANGX files, how Debug Tool locates 359, 360
EQANMDBG
  passing parameters to 99, 100
  using only EQANMDBG DD statement 100
  using only PARM 99
EQASET 308
EQASTART, entering command 10
EQUATE, SET command
  description 214
error numbers in Log window 151
evaluating expressions
  COBOL 235
  HLL 327
evaluation of expressions
  C and C++ 265
examining C++ objects 277
examples
  assembler
  sample program for debugging 203
  C
  sample program for debugging 181
C and C++
  assigning values to variables 259
  blocks and block identifiers 272
  expression evaluation 262
  monitoring and modifying
    registers and storage 278
  referencing variables and setting breakpoints 271
  scope and visibility of objects 271
C++
  displaying attributes 277
  sample program for debugging 191
  setting breakpoints 276
CEDF procedure 107
CEETEST calls, for PL/I 90
CEETEST function calls, for C 88
CEETEST function calls, for COBOL 89
changing point of view, general 331
COBOL
  %HEX function 237
  %STORAGE function 237
  assigning values to COBOL variables 231
  changing point of view 239
  displaying results of expression evaluation 236
  displaying values of COBOL variables 232
  qualifying variables 238
  sample program for debugging 155
  using constants in expressions 236
  declaring variables, for COBOL 234
  displaying program variables 259
  modifying setup files by using Debug Tool Utilities 363
OS/VS COBOL
  sample program for debugging 167
PL/I
  in PL/I 176
  sample program for debugging 173
PLITEST calls for PL/I 92
preparing programs by using Debug Tool Utilities 363
remote debug mode 83
examples (continued)
  specifying TEST run-time option with #pragma 84
  TEST run-time option 82
  using #pragma for TEST compiler option 41
  using constants 227
  using continuation characters 225
  using qualification 273
exception handling for C and C++ and PL/I 333
EXEC CICS RETURN
  under CICS 312
executing
  See running
expressions
diagnostics, for C and C++ 265
displaying values, for C and C++ 258
displaying values, for COBOL 236
evaluation for C and C++ 261, 265
evaluation for COBOL 235
evaluation of HLL 327
evaluation, operators and operands for C 264
for PL/I 252
using constants in, for COBOL 236

F
FIND command
  using with windows 131
finding
  characters or strings 131
  renamed source, listing or separate debug file 151
storage overwrite errors
  in assembler 211
  in C 189
  in C++ 200
  in COBOL 164
  in OS/VS COBOL 172
  in PL/I 180
uninitialized storage errors
  in C 189
  in C++ 201
fixes, obtaining 395
FREE command
  managing file allocations 150
freeform input, PL/I statements 250
full-screen mode
  continuation character, using in 225
CURSOR 128
CURSOR command 130
debugging in 117
description of 4
introduction to 11
PANEL COLORS 216
PANEL LAYOUT 214
PANEL PROFILE 217
screen 12
SCROLL 130
WINDOW CLOSE 215
WINDOW OPEN 215
WINDOW SIZE 215
WINDOW ZOOM 216
full-screen mode through a VTAM terminal
description of 4
starting a debugging session 111
function calls, for C and C++ 262
function, calling C and C++ from Debug Tool
C 187
C++ 198
function, unsupported for PL/I 254
functions
PL/I 252
functions, Debug Tool
%HAX
using with COBOL 237
%STORAGE
using with COBOL 237
using with COBOL 237

G
global data 278
global preferences file 357
global scope operator 278

H
H constant (COBOL) 227
halted location, displaying 134
header fields, Debug Tool session panel 118
help, online
for command syntax 227
hexadecimal format, displaying values in 147
hexadecimal format, displaying variable values in 144
hexadecimal format, how to display value of variable 147
hexadecimal format, how to monitor value of variable 148
hexadecimal format, monitoring values in 148
HFS, compiling a C program on 40
HFS, compiling a C++ program on 45
HFS, compiling Enterprise PL/I program on 34
highlighting, changing in Debug Tool session panel 216
history, Debug Tool command 129
retrieving previous commands 129
HOOK C++ TEST suboption 378
hooks
compiling with 21
compiling with, C 38
compiling with COBOL 26
compiling with, PL/I 33
compiling without, COBOL 21
removing from application 317, 318
removing, implications of 22
rules for placing 40, 41, 45
improving Debug Tool performance 317
IMS
compiling and linking 65
preparing
by using Debug Tool Utilities 65
programs, compiling 65
programs, debugging interactively 306
programs, linking 67
using Debug Tool with 305
IMS MFP
debugging 307
preparing to debug 68, 307
information centers, searching for problem resolution 395
information, displaying environmental 272
initial programs, non-Language Environment 323
CICS assembler 323
OS/VS COBOL 323
input areas, order of processing, Debug Tool 126
INSPLOG
creating the log file 135
DD name to store log file 357
example of using 96
INSPREP
example of using 96
INSPSAFE
example of using 96
interfaces
batch mode 5
full-screen mode 4
full-screen mode through a VTAM terminal 4
remote debug mode 5
interlanguage communication (ILC)
application, debugging 345
interlanguage programs, using with Debug Tool 333
Internet
searching for problem resolution 395
interpretive subset
general description 328
of C and C++ commands 257
of COBOL statements 229
of PL/I commands 247
INTERRUPT, Language Environment
run-time option 152
invoking
ISPF starting 127
J
JCL to create EQALANGX file 48
K
keywords, abbreviating 224
knowledge bases, searching for problem resolution 395
L
Language Environment
conditions, C and C++ equivalents 264
EQADCCX user exit 81
run-time options, precedence 81
LDD command, example 281
LEFT, SCROLL command 130
licensed documents xi
line breakpoint, setting 137
line continuation
for C 225
for COBOL 226
link-edit assembler program
how to, by using Debug Tool Utilities 50
linking
DB2 programs 52
EQADCCX 57
IMS programs 65, 67
LIST %HEX command 147
LIST command
use to display value of variable one time 145
LIST commands
LIST STORAGE using with PL/I 250
listing
file, finding renamed 151
find, OS PL/I 254
find, VS COBOL II 240
listing files, how Debug Tool locates 359, 360
literal constants, entering 227
Load Module Analyzer 27, 387
log file 135, 357
creating 135
default names 135
using 135
using as a commands file 135
Log window
description 122
error numbers in 151
retrieving lines from 129
log, session 81
LookAt message retrieval tool xii
low-level debugging 278
M
managing file allocations 150
manual restoring 143
message retrieval tool, LookAt xii
modifying
value of variable by typing over 148
value of variable by using command 148
modifying value of a C variable 184
MONITOR command
viewing output from, Debug Tool 121
MONITOR LIST command, using to monitor variables 145
Monitor window
description 121
opening and closing 149, 215
monitoring 145
monitoring storage in C++ 278
more than one language, debugging
programs with 333
moving around windows in Debug Tool 130
moving the cursor, Debug Tool 130
multilanguage programs, using with
Debug Tool 333
multiline commands
continuation character, using in 225
without continuation character 226
multiple enclaves
ending Debug Tool 340
interlanguage communication
application, debugging 345
starting 339
multithreading 337
restrictions 337
MVS
starting Debug Tool using TEST
run-time option 109
MVS POSIX programs, debugging 321
MVS, starting Debug Tool under 95

N
NAMES 347
NAMES command 350
using EQAOPTS 351
NAMES EXCLUDE command 349
restrictions of 349
NAMES INCLUDE command 348
restrictions for 348
naming conflicts 347
navigating session panel windows 130
NOMACGEN 285
non-Language Environment
CICS
passing run time parameters 64
Starting Debug Tool 63
restrictions 246
non-Language Environment initial
programs 323
CICS assembler 323
OS/VS COBOL 323
donnon-Language Environment programs
debugging 323
starting Debug Tool 98
NOTEST suboption of TEST run-time
option 80

O
objects
C and C++, scope of 268
opening Debug Tool session panel
windows 215
operators and operands for C 264
optimized programs, debugging
COBOL 319
optimized programs, debugging
large 350
options module, CEEUOPT run-time 52
OS PL/I programs, debugging 254
OS PL/I, finding list for 254

OS/VS COBOL 167
%PATHCODE values 246
compiler options to use 29
debugging a program in full-screen
mode 167
displaying raw storage 170
finding storage overwrite
errors 172
setting a breakpoint to halt 170
stopping on line if condition
true 171
defining as 169
how to prepare a 29
loading debug information 169
loading debug information for
restrictions 245
session panel's appearance 244
output
C, capturing to stdout 186
C++, capturing to stdout 198
overloaded operator 275
overwrite errors, finding storage
in assembler 211
in C 189
in C++ 200
in COBOL 164
in OS/VS COBOL 172
in PL/I 180

P
panel
header fields, session 118
Profile 217
PANEL command (full-screen mode)
changing session panel colors and
highlighting 216
paragraph trace, generating a COBOL
run-time 163
performance
enhancing Debug Tool's 52
performance, improving Debug Tool 317
PF keys
defining 213
using 128
PF4 key, using 145
PL/I 173
built-in functions 252
calling handling 249
constants 252
debugging a program in full-screen
mode 178
displaying raw storage 178
finding storage overwrite
errors 180
getting a function traceback 178
halting on line if condition is
true 177
modifying value of variable 176
setting a breakpoint to halt 176
setting breakpoint to halt 180
tracing run-time path for code
compiled with TEST 179
when not all parts compiled with
TEST 177
debugging OS PL/I programs 254
finding list for 254

PL/I (continued)
Enterprise, restrictions 255
expressions 252
notes on using 224
preparing a program for
debugging 33
run-time options 83
sample program for debugging 173
session variables 250
statements 247
syntax of TEST compiler option 382
PL/I listing, data set 356
PL/I source, data set 356
PL/I definition of 226
PLAYBACK commands
introduction to 17
PLAYBACK BACKWARD
using 140
PLAYBACK DISABLE
using 140
PLAYBACK ENABLE
using 138
PLAYBACK FORWARD
using 140
PLAYBACK START
using 139
PLAYBACK STOP
using 140
PLTEST 92
point of view, changing
description 331
for C and C++ 273
with COBOL 239
positioning lines at top of windows 131
precompiling DB2 programs 51
preference file 62, 79
preferences file 357
customizing with 219
prefix area
Debug Tool 126
prefix commands
prefix area on session panel 126
using in Debug Tool 127
prepare an assembler program, steps
to 47
preparing
a PL/I program for debugging 33
COBOL programs for debugging 25
DB2 stored procedures 55
to replay recorded statements using
PLAYBACK START command 139
previous commands, retrieving 129
problem determination
describing problems 398
determining business impact 397
submitting problems 398
profile settings, changing in Debug Tool
217
program
CICS, debugging 311
compiling an IMS 65
DB2, debugging 299
hook
compiling with, C 38
compiling with, COBOL 26
compiling with, PL/I 33
Index 415

program (continued)
hook (continued)
replaying 317, 318
removing 305
rules for placing 40, 41, 45
IMS, debugging 67
linking an IMS 67
multithreading, debugging 337
preparation
considerations, size and performance 317, 318
TEST compiler option, for C 37
TEST compiler option, for C++ 43
TEST compiler option, for COBOL 25
TEST compiler option, for PL/I 33
TEST compiler option, for VS COBOL II 26
reducing size 317
source, displaying with Debug Tool 120
starting for a debug session 103
stepping through 137
UNIX System Services, debugging 321
variables
accessing for C and C++ 258
variables, accessing for COBOL 231
pseudo-conversational program, saving settings 312
PX constant (PL/I) 227
Q
qualification
description, for C and C++ 272
general description 329
qualifying variables
with COBOL 237
R
recording
breakpoints using SET AUTOMONITOR 137
number of times each source line runs 136
restrictions on statements 140
session with the log file 135
statements, introduction to 17
statements, using PLAYBACK ENABLE command 138
stepping, using PLAYBACK DISABLE command 140
recording a debug session 122
referencing variables, implications of remote debug mode
description of 5
examples of 83
using Debug Tool in 371
removing statement and symbol tables 318
replaying statements, introduction to 17
replaying recorded statements 139
RUNTO command
using, to replay recorded statements 139
S
save breakpoints file 358
save monitor specifications file 358
save settings file 358
saving
breakpoints 141
monitor specifications 141
settings 141
setup file using Debug Tool Utilities 78
saving and restoring customizations 219
scope of objects in C and C++ 268
scroll area, Debug Tool 126
SCROLL command
using 130
search string, syntax of 132
searching for characters or strings 131
self-modifying code, restrictions for debugging 293
separate debug file 25
separate debug file files, how Debug Tool locates 359
separate debug file, data set 356
separate debug file, finding
renamed 151
session
variables, for PL/I 250
session panel
changing colors and highlighting in 216
changing window layout 214
command line 125
description 117
header fields 118
navigating 130
opening and closing windows 215
order in which Debug Tool accepts commands from 126
PF keys
initial settings 128
using 128
while debugging OS/VS COBOL 244
windows
scrolling 130
session panel, while debugging assembler 282
session settings
changing in Debug Tool 213
session variables
declaring, for COBOL 234
SET AUTOMONITOR ON command, example 147
SET commands
SET AUTOMONITOR
using to record breakpoints 137
viewing output from 121
SET AUTOMONITOR ON monitoring values of variables 145
SET DEFAULT SCROLL using 119
starting Debug Tool
\_ctest(), using 93
batch mode 96
DB2 program with TSO 300
from a Language Environment
program 85
starting your program for a debug
session 103
under CICS 103, 104, 107
under CICS, using CEEUOPT 106
under IMS 306
under MVS in TSO 95
using the TEST run-time option 79
with PLITEST 92
with the CEETEST function call 85
within an enclave 339
Starting Debug Tool
at different points 80
starting Debug Tool Utilities 10
starting interactive function calls
in C 187
starting your program 109
statement tables, removing 318
statements
PL/I 247, 250
recording and replaying, introduction
to 17
stdout, capturing output to
in C 186
in C++ 198
STEP command
using, to replay recorded
statements 139
stepping
through a program 137
through C++ programs 275
stepping, introduction to 14
stopping
Debug Tool session 18
storage
classes, for C 269
OS/VS COBOL, displaying 170
storage errors, finding
overwrite
in assembler 211
in C 189
in C++ 200
in COBOL 164
in OS/VS COBOL 172
in PL/I 180
uninitialized
in C 189
in C++ 201
STORAGE run-time option,
specifying 83
storage, raw
C, displaying 187
C++, displaying 199
COBOL, displaying 162
PL/I, displaying 178
stored procedures
DB2, debugging 303
string
syntax for searching 132
string substitution, using 214
strings
C, displaying 187
strings (continued)
C++, displaying 199
searching for in a window 131
substitution, using string 214
symbol tables, removing 318
syntax diagrams
how to read xvi
SYSDEBUG 357
SYSTCPD 371
system commands, issuing, Debug
Tool 127
T
template in C++ 275
Terminal Interface Manager
example of 98
how to start 111
terminology, Debug Tool xv
TEST compiler option
benefit of using 375
debugging C when only a few parts
are compiled with 186
debugging C++ when only a few
parts are compiled with 197
debugging COBOL when only a few
parts are compiled with 161
debugging OS/VS COBOL when only
a few parts are compiled with 171
debugging PL/I when only a few
parts are compiled with 177
for C 37
for C++ 43
for COBOL 25
for PL/I 33
restrictions 376
specifying NUMBER option with 26
suboptions 25
syntax of C 375
syntax of C++ 377
syntax of COBOL 378
syntax of PL/I 382
using #pragma statement to
specify 41
using for IMS 67
TEST run-time option
as parameter on RUN
subcommand 300
example of 82
for PL/I 250
specifying with #pragma 84
suboption processing order 79
this pointer, in C++ 197
trace, generating a COBOL run-time
paragraph 163
traceback, COBOL routine 162
traceback, function
in assembler 210
in C 188
in C++ 199
in PL/I 178
traceback, OS/VS COBOL routine 171
tracing run-time path
in C 188
in C++ 200
in COBOL 162
in PL/I 179
TRAP, Language Environment run-time option 152, 331
TRAP(ON) run-time option,
  specifying 84
trigraph 224
trigraphs
  using with C 224
TSO
  starting Debug Tool using TEST
  run-time option 109
TSO command
  using to debug DB2 program 300
TSO, starting Debug Tool under 95

U
uninitialized storage errors, finding
  in C 189
  in C++ 201
UNIX System Services
  compiling a C program on 40
  compiling a C++ program 45
  compiling a Enterprise PL/I program
    on 34
  using Debug Tool with 321
unsupported
  HLL modules, coexistence with 336
  PL/I language elements 254
UP, SCROLL command 130
USE file 80
using Debug Tool
  finding renamed source, listing, or
    separate debug file 151

V
values
  assigning to C and C++
    variables 259
  assigning to COBOL variables 231
variable
  automonitor 15
  changing value of 16
  continuous display 15
  displaying value of 15
  modifying value
    in C 184
    in C++ 195
    in COBOL 159
    in PL/I 176
  one-time and continuous display 15
  one-time display 15
  using SET AUTOMONITOR ON
    command to monitor value of 145
    value, displaying 144
variables
  accessing program, for C and
    C++ 258
  accessing program, for COBOL 231
  assigning values to, for C and
    C++ 259
  assigning values to, for COBOL 231
  compatible attributes in multiple languages 334
  displaying, for C and C++ 258
  displaying, for COBOL 232

W
warning, for PL/I 254
what you need to debug 24
window id area, Debug Tool 126
window, error numbers in 151
windows, Debug Tool session panel
  changing configuration 214
  opening and closing 215
  resizing 215
  zooming 216
workstation debugging
  See remote debug mode

X
XPLINK
  restriction on applications that
    use 294

Z
zooming a window, Debug Tool 216
Readers’ Comments — We’d Like to Hear from You

Debug Tool for z/OS
Debug Tool Utilities and Advanced Functions for z/OS
User’s Guide
Version 7.1

Publication No. SC19-1071-02

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.
Submit your comments using one of these channels:
• Send your comments to the address on the reverse side of this form.
• Send your comments via e-mail to: COMMENTS@US.IBM.COM

If you would like a response from IBM, please fill in the following information:

Name
Address

Company or Organization

Phone No.
E-mail address