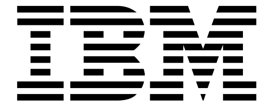
Debug Tool for VSE/ESA

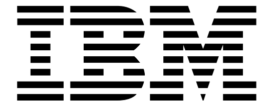


User's Guide and Reference

Release 1

SC26-8797-00

Debug Tool for VSE/ESA



User's Guide and Reference

Release 1

SC26-8797-00

Note!

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

First Edition (December 1996)

This edition applies to the Debug Tool feature of the following compilers:

- IBM C for VSE/ESA Version 1, Release 1 (Program Number 5686-A01)
- IBM COBOL for VSE/ESA Version 1, Release 1 (Program Number 5686-068)
- IBM PL/I for VSE/ESA Version 1, Release 1 (Program Number 5686-069)

and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, Department W92/H3
P.O. Box 49023
San Jose, CA, 95161-9023
United States of America

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

© **Copyright International Business Machines Corporation 1995, 1996. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . xii
Programming Interface Information . . . xii
Trademarks . . . xii

About This Book . . . xiii
IBM Language Environment for VSE/ESA . . . xiii
Debug Tool . . . xiv
Who Might Use This Book . . . xiv
How This Book Is Organized . . . xv
Using Your Documentation . . . xv
How to Read the Syntax Diagrams . . . xvi

Part 1. Getting Started . . . 1

Chapter 1. Before You Begin Debugging . . . 2
Debug Tool Debugging Environments . . . 2
Planning to Run Your Program with Debug Tool . . . 4
A Sample Interactive Debug Tool Session (COBOL) . . . 5

Chapter 2. Preparing to Debug Your Program . . . 12
Compiling a C Program with the Compile-Time TEST Option . . . 12
  Using #pragma to Specify Compile-Time TEST Option . . . 16
Compiling a COBOL Program with the Compile-Time TEST Option . . . 16
Compiling a PL/I Program with the Compile-Time TEST Option . . . 19
Debugging Multilanguage Programs . . . 22
  Debugging an Application Fully Supported by LE/VSE . . . 22
  Debugging an Application Partially Supported by LE/VSE . . . 23
Compiler Listings (and Program Source) . . . 23
  Debug Tool Compiler Print Exit . . . 23
  Assigning SYSLST to Disk . . . 26

Chapter 3. Beginning a Debugging Session . . . 27
Files Used By Debug Tool . . . 27
  Source and Listing Files . . . 27
  Preferences File . . . 29
  Commands File . . . 29
  Profile Settings File . . . 29
  The Log File . . . 30
  Determining the Default Userid . . . 30
  Specifying Files to Debug Tool . . . 31
Using the Run-Time TEST Option . . . 33
  Run-Time TEST Option Syntax . . . 33
  Run-Time TEST Option Considerations . . . 37
  Run-Time TEST Option Examples . . . 40
  Specifying Run-Time TEST Option with #pragma runopts in C . . . 41
  Specifying Run-Time TEST Option with PLIXOPT string in PL/I . . . 41
Invoking Your Program When Starting a Debugging Session . . . 42
  Invoking Your Program for a Debugging Session . . . 42
  Invoking Debug Tool under CICS . . . 43

© Copyright IBM Corp. 1995, 1996

Using Alternative Debug Tool Invocation Methods 43
Invoking Debug Tool with CEETEST 43
Invoking Debug Tool with the __ctest() Function 49
Invoking Debug Tool with PLITEST 51

Chapter 4. Debugging Your Programs in Full-Screen Mode 53
Preparing for Debugging 53
Invoking Your Program with Debug Tool 53
Ending a Debug Session 54
Basic Tasks of Debug Tool 54
Debug Tool Interface 54
Help 54
Window Control 54
Setting a Line Breakpoint 56
Stepping through or Running Your Program. 56
Displaying a Variable's Value 56
Continuously Displaying a Variable's Value 56
Setting a PF Key 56
Error Numbers for Messages in the log Window 57
Finding a Renamed Source File Using Debug Tool 57
Using a C Program to Demonstrate a Debug Tool Session 57
C Tasks 62
Using a COBOL Program to Demonstrate a Debug Tool Session 67
COBOL Tasks 71
Using a PL/I Program to Demonstrate a Debug Tool Session 76
PL/I Tasks 80

Chapter 5. Using the Debug Tool Interfaces 84
Customizing Debug Tool for Your Environment 84
Using the Debug Tool Session Panel 84
Session Panel Windows 85
Source Window 1 86
Monitor Window 3 87
Log Window 2 87
Using the Session Log File to Maintain a Record of Your Session 87
Entering Commands in a Debug Tool Session 89
Command Sequencing 90
Using the Command Line 90
Using Prefix Commands 90
Using Cursor Commands 91
Using Program Function (PF) Keys to Enter Commands 91
Defining PF Keys 91
Abbreviating Commands 91
Retrieving Commands 92
Retrieving Lines from the Session Log and Source Windows 92
Creating EQUATES and Using String Substitution 92
Navigating Through Debug Tool Session Panel Windows 93
Moving the Cursor 93
Scrolling the Windows 93
Positioning Lines at the Top of Windows 93
Searching for a Character or Character String 93
Customizing Your Session 94
Changing Session Panel Window Layout 95
Opening and Closing Session Panel Windows 96

Sizing Session Panel Windows . . . 96
Intersecting Windows . . . 97
Horizontal Windows . . . 97
Vertical Windows . . . 97
Zooming a Window . . . 97
Customizing Colors . . . 97
Customizing Settings . . . 99
Getting Help During Your Session . . . 102

Chapter 6. Multiple Enclaves . . . 103
Invoking Debug Tool within an Enclave . . . 103
Using the Source Window . . . 104
Retaining a Log File of your Debug Tool Session . . . 104
Processing Commands from a Commands File . . . 104
Using Breakpoints within Multiple Enclaves . . . 104
Ending a Debug Tool Session . . . 104
Using Debug Tool Commands within Multiple Enclaves . . . 104

Chapter 7. Using Debug Tool in Different Modes and Environments . . . 107
Using Debug Tool in Batch Mode . . . 107
Debugging CICS Programs . . . 107
Debug Modes under CICS . . . 108
Mechanisms for Invoking Debug Tool under CICS . . . 109
Preparing and Using DTCN to Invoke Debug Tool under CICS . . . 109
Preparing and Using CEEUOPT to Invoke Debug Tool under CICS . . . 114
Preparing and Using Compile-Time Directives To Invoke Debug Tool under CICS . . . 114
Restrictions When Debugging Under CICS . . . 114
Debugging DL/I Programs . . . 115
Programming Considerations . . . 115
Program Preparation . . . 115
Compile Requirements . . . 115
Link Requirements . . . 116
Using Debug Tool with DL/I Programs . . . 118
Batch Mode . . . 118
Interactive Mode . . . 118
Debugging SQL/DS Programs . . . 118
Programming Considerations . . . 119
Program Preparation . . . 119
Preprocessor Requirements . . . 119
Compile Requirements . . . 119
Link Requirements . . . 120
Using Debug Tool with SQL/DS Programs . . . 120
Batch Mode . . . 120
Interactive Mode . . . 121

Part 2. Language-Specific Information . . . 123

Chapter 8. Debug Tool Support of Programming Languages . . . 124
Multiple Enclaves and Interlanguage Communication (ILC) . . . 124
Compatible Attributes Mapped Between HLL Data Types . . . 124
Debug Tool Evaluation of HLL Expressions . . . 125
Debug Tool Interpretation of HLL Variables and Constants . . . 125

Debug Tool Variables (or Intrinsic Functions) 125
Interpretive Subsets 129
Qualifying Variables and Changing the Point of View 129
Qualification 130
Changing the Point of View 131
Debug Tool Handling of Conditions and Exceptions 131
Condition Handling in Debug Tool 132
Exception Handling within Expressions (C and PL/I only) 133
Requesting an Attention Interrupt During Interactive Sessions 133
Debug Tool's Built-in Functions 134
For Use with C, COBOL, and PL/I 134
For Use with C and PL/I 134
For Use with PL/I 135
Displaying Environmental Information 135
Low-Level Debugging 136

Chapter 9. Using Debug Tool with C Programs 138
Debug Tool Commands 138
Using C Variables with Debug Tool 138
Accessing Program Variables 138
Displaying Values of C Variables or Expressions 139
Declaring Temporary Variables 139
Assigning Values to C Variables 140
Using Debug Tool Variables in C 140
C Expressions 145
Using Debug Tool Functions with C 147
Debug Tool Evaluation of C Expressions 149
Using SET INTERCEPT with C Programs 150
Objects and Scopes 153
Storage Classes 154
Blocks and Block Identifiers for C 155
Displaying Environmental Information 156
Using Qualification for C 156
Using Qualifiers 157
Changing the Point of View 158

Chapter 10. Using Debug Tool with COBOL Programs 160
The Debugging Environment Provided for COBOL Programs 160
Debug Tool Commands 160
Restrictions on COBOL-like Commands 161
Using COBOL Variables with Debug Tool 163
Accessing Program Variables 164
Assigning Values to COBOL Variables 164
Declaring Temporary Variables 165
Displaying Values of COBOL Variables 166
Using DBCS Characters 166
Using Debug Tool Variables in COBOL 167
Debug Tool Evaluation of COBOL Expressions 172
Displaying the Results of Expression Evaluation 173
Using Constants in Expressions 173
Using Debug Tool Functions with COBOL 174
Using %HEX 174
Using the %STORAGE Function 174
Using Qualification for COBOL 174

Using Qualifiers 174
Changing the Point of View 176

Chapter 11. Using Debug Tool with PL/I Programs 177
Debug Tool Commands 177
PL/I Language Statements 177
Using PL/I Variables with Debug Tool 178
Accessing Program Variables 178
Displaying Values of PL/I Variables or Expressions 178
Structures 179
Assigning Values to PL/I Variables 181
Using Debug Tool Variables in PL/I 181
PL/I Expressions 186
Using DBCS Characters - Freeform Input 186
PL/I Built-In Functions 187
Using SET WARNING Command with Built-Ins 187
Using Debug Tool Functions with PL/I 187
Using Qualification for PL/I 189
Using Qualifiers 189
Changing the Point of View 191

Part 3. Debug Tool Reference 193

Chapter 12. Using Debug Tool Commands 194
Command Modes and Language Support 194
Entering Commands 194
Command Format 194
Character Set and Case 194
Abbreviating Keywords 195
Continuation (Full-screen mode) 196
Significance of Blanks 197
Comments 198
Constants 198
Retrieving Commands from the Log and Source Windows 199
Online Command Syntax Help 199
Common Syntax Elements 200
Block_Name 200
Block_Spec 200
Compile_Unit_Name 201
CU_Spec 201
Expression 202
Phase_Name 202
Load_Spec 203
References 203
Statement_Id 203
Statement_Id_Range and Stmt_Id_Spec 204
Statement_Label 205

Chapter 13. Debug Tool Commands 206
ANALYZE Command (PL/I) 206
Assignment Command (PL/I) 207
AT Command 208
Every_Clause 209

AT ALLOCATE (PL/I) . . . 210
AT APPEARANCE . . . 211
AT CALL . . . 213
AT CHANGE . . . 215
AT CURSOR (Full-Screen Mode) . . . 218
AT DELETE . . . 219
AT ENTRY/EXIT . . . 220
AT GLOBAL . . . 221
AT LABEL . . . 222
AT LINE . . . 224
AT LOAD . . . 224
AT OCCURRENCE . . . 225
AT PATH . . . 228
AT Prefix (Full-Screen Mode) . . . 229
AT STATEMENT . . . 230
AT TERMINATION . . . 231
BEGIN Command (PL/I) . . . 232
block Command (C) . . . 233
break Command (C) . . . 233
CALL Command . . . 234
CALL %DUMP . . . 235
CALL entry_name (COBOL) . . . 239
CALL procedure . . . 240
CLEAR Command . . . 240
CLEAR Prefix (Full-Screen Mode) . . . 243
COMMENT Command . . . 244
COMPUTE Command (COBOL) . . . 245
CURSOR Command (Full-Screen Mode) . . . 246
Declarations . . . 246
Language Compatible Attributes . . . 246
Declarations (C) . . . 247
Declarations (COBOL) . . . 250
DECLARE Command (PL/I) . . . 251
DESCRIBE Command . . . 253
DISABLE Command . . . 255
DISABLE Prefix (Full-Screen Mode) . . . 255
do/while Command (C) . . . 256
DO Command (PL/I) . . . 256
ENABLE Command . . . 259
ENABLE Prefix (Full-Screen Mode) . . . 259
EVALUATE Command (COBOL) . . . 260
Expression Command (C) . . . 261
FIND Command . . . 262
for Command (C) . . . 263
GO Command . . . 265
GOTO Command . . . 266
GOTO LABEL Command . . . 267
if Command (C) . . . 268
IF Command (COBOL) . . . 269
IF Command (PL/I) . . . 269
IMMEDIATE Command (Full-Screen Mode) . . . 270
INPUT Command (C and COBOL) . . . 271
LIST Command . . . 272
LIST (blank) . . . 273

LIST AT . . . 273
LIST CALLS . . . 275
LIST CURSOR (Full-Screen Mode) . . . 276
LIST Expression . . . 276
LIST FREQUENCY . . . 277
LIST LAST . . . 278
LIST LINE NUMBERS . . . 279
LIST LINES . . . 279
LIST MONITOR . . . 279
LIST NAMES . . . 279
LIST ON (PL/I) . . . 281
LIST PROCEDURES . . . 281
LIST REGISTERS . . . 282
LIST STATEMENT NUMBERS . . . 282
LIST STATEMENTS . . . 283
LIST STORAGE . . . 283
MONITOR Command . . . 284
MOVE Command (COBOL) . . . 286
Null Command . . . 287
ON Command (PL/I) . . . 287
PANEL Command (Full-Screen Mode) . . . 289
PERFORM Command (COBOL) . . . 291
Prefix Commands (Full-Screen Mode) . . . 292
PROCEDURE Command . . . 293
QUERY Command . . . 294
QUERY Prefix (Full-Screen Mode) . . . 297
QUIT Command . . . 297
RETRIEVE Command (Full-Screen Mode) . . . 298
RUN Command . . . 299
SCROLL Command (Full-Screen Mode) . . . 299
SELECT Command (PL/I) . . . 301
SET Command . . . 302
SET CHANGE . . . 303
SET COLOR (Full-Screen Mode) . . . 304
SET COUNTRY . . . 306
SET DBCS . . . 306
SET DEFAULT LISTINGS . . . 307
SET DEFAULT SCROLL (Full-Screen Mode) . . . 307
SET DEFAULT WINDOW (Full-Screen Mode) . . . 308
SET ECHO . . . 308
SET EQUATE . . . 309
SET EXECUTE . . . 310
SET FREQUENCY . . . 310
SET HISTORY . . . 311
SET INTERCEPT (C and COBOL) . . . 311
SET KEYS (Full-Screen Mode) . . . 313
SET LOG . . . 313
SET LOG NUMBERS (Full-Screen Mode) . . . 314
SET MONITOR NUMBERS (Full-Screen Mode) . . . 314
SET MSGID . . . 315
SET NATIONAL LANGUAGE . . . 315
SET PACE . . . 316
SET PFKEY . . . 316
SET PROGRAMMING LANGUAGE . . . 317

SET QUALIFY . . . 318
SET REFRESH (Full-Screen Mode) . . . 320
SET REWRITE . . . 320
SET SCREEN (Full-Screen Mode) . . . 321
SET SCROLL DISPLAY (Full-Screen Mode) . . . 321
SET SOURCE . . . 322
SET SUFFIX (Full-Screen Mode) . . . 323
SET TEST . . . 323
SET WARNING (C and PL/I) . . . 325
SET Command (COBOL) . . . 326
SHOW Prefix Command (Full-Screen Mode) . . . 326
STEP Command . . . 327
switch Command (C) . . . 328
TRIGGER Command . . . 331
USE Command . . . 333
while Command (C) . . . 335
WINDOW Command (Full-Screen Mode) . . . 335
WINDOW CLOSE . . . 336
WINDOW OPEN . . . 336
WINDOW SIZE . . . 337
WINDOW ZOOM . . . 338

Part 4. Appendixes . . . 341

Appendix A. Coexistence . . . 342
Coexistence with Other Debug Tools . . . 342
Coexistence with Unsupported HLL Modules . . . 342

Appendix B. Using Debug Tool in a Production Mode . . . 343
Fine-Tuning Your Programs with Debug Tool . . . 343
Removing Hooks, Statement Tables, and Symbol Tables . . . 343
Using Debug Tool on Optimized Programs . . . 344

Appendix C. Using C Reference Information with Debug Tool . . . 346
Debug Tool Interpretive Subset of C Commands . . . 346
C Reserved Keywords . . . 346
Operators and Operands . . . 346
LE/VSE Conditions and Their C Equivalents . . . 347

Appendix D. Using COBOL Reference Information with Debug Tool . . . 349
Debug Tool Interpretive Subset of COBOL Commands . . . 349
COBOL Reserved Keywords . . . 349
Allowable Comparisons for the Debug Tool IF Command . . . 349
Allowable Moves for the Debug Tool MOVE Command . . . 351
Allowable Moves for the Debug Tool SET Command . . . 352

Appendix E. Using PL/I Reference Information with Debug Tool . . . 353
Debug Tool Interpretive Subset of PL/I Commands . . . 353
PL/I Reserved Keywords . . . 353
Conditions and Condition Handling . . . 353
Unsupported PL/I Language Elements . . . 354

Appendix F. Debug Tool Messages . . . 355

Bibliography . . . 391
Debug Tool Publications . . . 391
Language Environment Publications . . . 391
LE/VSE-Conforming Language Product Publications . . . 391
Related Publications . . . 391
Softcopy Publications . . . 392

Glossary . . . 393

Index . . . 399

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

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 Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Programming Interface Information

This book is intended to help with application programming. This book documents General-Use Programming Interface and Associated Guidance Information provided by Debug Tool for VSE/ESA.

General-Use programming interfaces allow the customer to write programs that obtain the services of Debug Tool for VSE/ESA.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

CICS	System/370
IBM	System/390
Language Environment	VSE/ESA
SQL/DS	VTAM

About This Book

Debug Tool for VSE/ESA (referred to throughout this book as Debug Tool) combines the richness of the System/370 and System/390 environments with the power of IBM Language Environment for VSE/ESA (LE/VSE) to provide a debug tool for programmers to isolate and fix their program bugs and test their applications. Debug Tool gives you the capability of testing programs in batch or using a nonprogrammable terminal in full-screen mode to debug your programs interactively.

This book contains instructions and examples to help you use Debug Tool to debug C, COBOL, and PL/I applications running with LE/VSE. Topics covered include preparing your application for debugging, accomplishing basic debugging tasks, and Debug Tool's interaction with different programming languages. A complete command reference section is also included.

You can begin testing with Debug Tool after learning just a few concepts:

- How to invoke it
- How to set, display, and remove breakpoints
- How to step through your program

Debug Tool commands are similar to commands from the supported high-level languages (HLLs).

IBM Language Environment for VSE/ESA

Debug Tool can be used to debug application programs written in high-level languages that use the run-time environment and library of run-time callable services provided by LE/VSE.

LE/VSE establishes a common run-time environment and common run-time callable services for language products, user programs, and other products.

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that LE/VSE provides to your application include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, support for *interlanguage communication (ILC)* and *condition handling*.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help the execution, performance tuning, performance, and diagnosis of your application.
- Access to language-specific library routines.

© Copyright IBM Corp. 1995, 1996

Debug Tool

Debug Tool is distributed as part of the Full Function offering of the following IBM high-level language compilers:

- IBM C for VSE/ESA
- IBM COBOL for VSE/ESA
- IBM PL/I for VSE/ESA

Debug Tool is a powerful interactive source-level debug tool for application programs written in the above high-level languages. It allows programmers to address difficult problems at the source level where they are comfortable.

While a program is running, programmers can control and examine its execution with functions such as:

- Viewing the source listing and stepping through the source one statement at a time
- Setting dynamic break points, which can be simple or conditional based on other values in the program
- Monitoring the value of program variables
- Modifying program and variable storage
- Debugging mixed-language applications from a single debug session

The debug session can be recorded in a log file, so the programmer can replay the session. Programmers can use Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

Who Might Use This Book

This book is intended for application programmers using Debug Tool to debug high-level languages with LE/VSE under VSE/ESA. Throughout this book, these languages are referred to as C, COBOL, and PL/I.

The following environments are supported:

- Batch
- Customer Information Control System (CICS)
- Data Language/I (DL/I)
- Structured Query Language/Data System (SQL/DS)

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

How This Book Is Organized

Part 1 introduces you to Debug Tool. The first three chapters discuss the preparatory work you must complete before using it. Chapter 4 discusses accomplishing basic tasks with Debug Tool in full-screen mode by providing scenarios that help you begin using Debug Tool, including helpful hints when performing some basic debugging tasks. The last three chapters discuss how to debug programs that contain applications written in more than one HLL and information about using Debug Tool in a variety of environments, including batch mode and CICS. Also covered are debugging applications that contain SQL/DS statements and calls to DL/I.

Part 2 discusses Debug Tool's interaction with different programming languages. Debug Tool variables, functions, and expression evaluation are explained. Separate chapters provide more details on using Debug Tool with C, COBOL, and PL/I.

Part 3 is the Command Reference section. This section describes all of the Debug Tool commands, shows their syntax, and provides examples of their use.

Part 4 contains the appendixes. These include: a discussion of the coexistence of Debug Tool with HLL phases compiled with previous versions of compilers; information on how to optimize your programs while still retaining some debugging capability; tables of reference material for use with C, COBOL, and PL/I (for example, reserved keywords and Debug Tool interpretive subsets of HLL commands); and a complete list of Debug Tool messages.

Following the appendixes are a bibliography and a glossary of terms.

Using Your Documentation

The publications provided with Debug Tool and LE/VSE are designed to help you:

- Plan for, install, customize, and maintain Debug Tool.
- Debug problems in your LE/VSE-conforming applications.

Language programming information is provided in the high-level language programming manuals that provide language definition, library function syntax and semantics, and programming guidance information.

Each publication helps you perform a different task. For a complete list of publications you might need, see “Bibliography” on page 391.

Table 1. How to Use Debug Tool, LE/VSE and Language Publications

To...	Use...	
Evaluate Debug Tool	*Debug Tool for VSE/ESA Fact Sheet*	GC26-8925
Plan for, install, customize, and maintain Debug Tool	*Debug Tool for VSE/ESA Installation and Customization Guide*	SC26-8798
Use Debug Tool to debug LE/VSE-conforming applications	*Debug Tool for VSE/ESA User's Guide and Reference*	SC26-8797
Debug your LE/VSE-conforming application and get details on run-time messages	*LE/VSE Debugging Guide and Run-Time Messages*	SC33-6681
Diagnose problems that occur in your LE/VSE-conforming application	*LE/VSE Debugging Guide and Run-Time Messages*	SC33-6681
Understand the LE/VSE program models and concepts	*LE/VSE Concepts Guide* *LE/VSE Programming Guide*	GC33-6680 SC33-6684
Diagnose compiler problems that occur in your LE/VSE-conforming application	Your compiler Diagnosis Guide	

How to Read the Syntax Diagrams

The following rules apply to the syntax diagrams used in this book:

Arrow symbols

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

►►──	Indicates the beginning of a statement.
───►	Indicates that the statement syntax is continued on the next line.
►──	Indicates that a statement is continued from the previous line.
───►◄	Indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the ►── symbol and end with the ───► symbol.

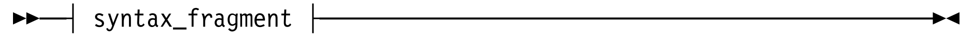
Conventions

- Keywords, their allowable synonyms, and reserved parameters, appear in uppercase. These items must be entered exactly as shown.
- Variables appear in lowercase italics (for example, *column-name*). They represent user-defined parameters or suboptions.
- When entering commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as given.
- Footnotes are shown by a number in parentheses, for example, (1).

Syntax Fragments

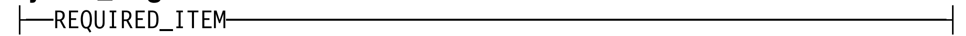
If a syntax diagram contains too many items to fit in the diagram, the syntax is shown by a main syntax diagram and one or more syntax fragments.

A syntax fragment is referred to in the main diagram by its fragment name between two vertical bars.



Each syntax fragment appears below the main syntax diagram, and is delimited by vertical bars. A heading above the fragment indicates the name of the fragment.

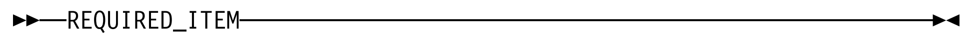
syntax_fragment:



Read each syntax fragment as though it were imbedded in the main syntax diagram.

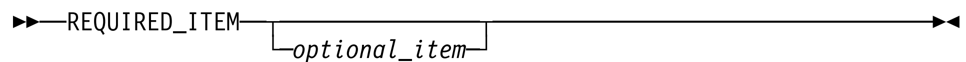
Required items

Required items appear on the horizontal line (the main path).

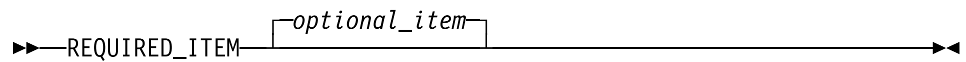


Optional Items

Optional items appear below the main path.

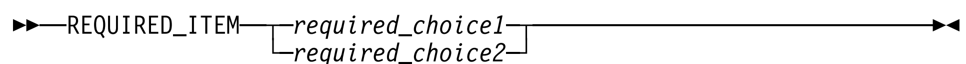


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

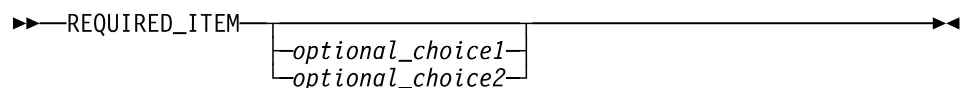


Multiple required or optional items

If you can choose from two or more items, they appear vertically in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

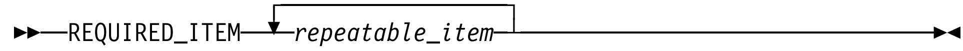


If choosing one of the items is optional, the entire stack appears below the main path.

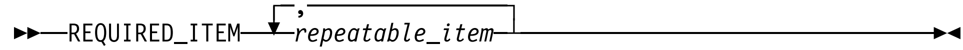


Repeatable items

An arrow returning to the left above the main line indicates that an item can be repeated.



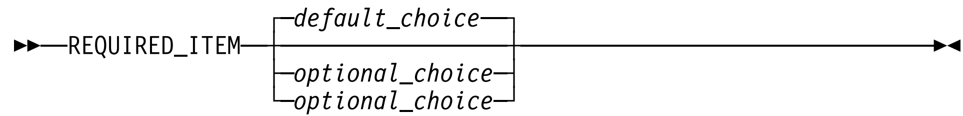
If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can specify more than one of the choices in the stack.

Default keywords

IBM-supplied default keywords appear above the main path, and the remaining choices are shown below the main path. In the parameter list following the syntax diagram, the default choices are underlined.



Part 1. Getting Started

© Copyright IBM Corp. 1995, 1996

Chapter 1. Before You Begin Debugging

Debug Tool is a program-testing and analysis aid that helps you examine, monitor, and control the execution of programs written in C, COBOL, or PL/I running with LE/VSE under VSE/ESA. Applications can include other languages, but Debug Tool does not debug those portions of your application.

This chapter provides an overview of the terminology used by the Debug Tool and some helpful hints you should consider before beginning.

It also provides a sample Debug Tool session to help you understand the features Debug Tool provides to assist you in debugging your programs.

Debug Tool Debugging Environments

Debug Tool provides several debugging environments. The number of modes of operation and languages supported by Debug Tool has necessitated that certain terms and conventions be adopted for use throughout this manual to reduce possible conflict between references to the different languages.

The terms *full-screen mode*, and *batch mode* are used to describe the types of debugging sessions or interfaces Debug Tool provides. Included in the following sections are definitions of these terms.

Debug Tool Sessions

Full-Screen Session
Debug Tool provides an interactive full-screen interface on a 3270 device. The full-screen interface is made up of session panel windows containing information about your debugging session.

Batch Session
Debug Tool commands files provide a mechanism to predefine a series of Debug Tool commands to be performed on an executing batch application. Neither terminal input nor user interaction is available for batch debugging of an application.

Full-Screen Session Interface

As Figure 1 on page 3 shows, in a full-screen session Debug Tool provides three windows:

- A Source window (**1**) in which to view your program source or listing
- A Log window (**2**), which records commands and other interactions between Debug Tool, and your program
- A Monitor window (**3**) in which to monitor changes in your program

You can adjust the sizes of the windows with the cursor, and change the relative locations of the windows by typing your preferences on a template.

© Copyright IBM Corp. 1995, 1996

```
COBOL    LOCATION: MULTCU :> 75.1
Command ===>                                                    Scroll ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 1 OF 2
******************************* TOP OF MONITOR ********************************
0001  1 01 MULTCU:>PROGRAM-USHORT-BIN    00000      3
0002  2 01 MULTCU:>PROGRAM-SSHORT-BIN    +00000
****************************** BOTTOM OF MONITOR ******************************

SOURCE: MULTCU ---1----+----2----+----3----+----4----+----5----+ LINE: 66 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    1       MOVE -25 TO PROGRAM-SSHORT-BIN.                           .
      78            PERFORM TEST-900.                                         .
      79            PERFORM TEST-1000.                                        .
      80            DISPLAY "MULTCU COBOL SOURCE ENDED." UPON CONSOLE.        .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6 LINE: 6 OF 14
0007  MONITOR
0008    LIST PROGRAM-USHORT-BIN ;
0009  MONITOR                            2
0010    LIST PROGRAM-SSHORT-BIN ;
0011  AT 75 ;
0012  AT 77 ;
0013  AT 79 ;
0014  GO ;
```

Figure 1. The Debug Tool Windows

For an explanation of all the windows, see Chapter 5, "Using the Debug Tool Interfaces" on page 84.

Denoting Environmental Differences

Certain aspects of Debug Tool usage can differ depending on the environment in which it is executing. Within this manual when explanations on these differences occur, the differences are marked in the text in the following fashion:

For CICS Only

CICS-specific information.

End of For CICS Only

Special language-specific information about accomplishing a task or using a particular procedure might also be marked the same way. More extensive differences are usually discussed in separate sections.

Terminology

Because of differing terminology among the various languages supported by Debug Tool, a group of common terms has been established. Table 2 on page 4 lists these terms and their equivalent in each language.

Table 2. Terminology

Debug Tool Term	C Equivalent	COBOL Equivalent	PL/I Equivalent
Compile Unit	C source file	Program	Program
Block	Function or Compound Statement	Program, Nested Program or PERFORM Group of Statements	Block
Label	Label	Paragraph Name or Section Name	Label

Planning to Run Your Program with Debug Tool

Before you can test your program using Debug Tool, you need to plan how you want to conduct your debugging session.

- Do you want to compile your program with hooks?

  Hooks are instructions inserted in a program by a compiler at compile time. Using hooks, you can set breakpoints that instruct Debug Tool to gain control at selected points during program execution.

  You can choose where to place the hooks. For example, you can place them at statements, or only at entry to and exit from blocks.

  For more information on placing hooks, see Chapter 2, “Preparing to Debug Your Program” on page 12.

- Do you want to be able to reference variables during your Debug Tool session?

  If yes, you need to instruct the compiler to create a symbol table. The symbol table contains descriptions of variables, their attributes, and their location in storage. These descriptions are used by Debug Tool when referencing variables.

  For more information on creating symbol tables, see Chapter 2, “Preparing to Debug Your Program” on page 12.

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

  Removing hooks, statement tables, or symbol tables can improve your application's performance and/or decrease its size. See Appendix B, “Using Debug Tool in a Production Mode” on page 343 for a complete discussion.

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

  There are a variety of ways to invoke Debug Tool:

  – Using the run-time TEST option—this option gives you the choice of invoking Debug Tool either prior to the execution of the application, or at the occurrence of an HLL condition during the execution of your application. For more information, see “Using the Run-Time TEST Option” on page 33.

  – Using the LE/VSE CEETEST callable service—with CEETEST, you can invoke Debug Tool directly from any HLL program. For more information, see “Using Alternative Debug Tool Invocation Methods” on page 43.

- Using the C `__ctest()` function—with `__ctest()`, you can invoke Debug Tool directly from a C program. For more information, see “Using Alternative Debug Tool Invocation Methods” on page 43.
- Using the PL/I PLITEST built-in subroutine—with PLITEST, you can invoke Debug Tool directly from a PL/I program. For more information, see “Using Alternative Debug Tool Invocation Methods” on page 43.

After Debug Tool is invoked, it can gain control of your program and suspend execution to allow you to take such actions as checking the value of a variable or examining the contents of storage.

- Do you want to use Debug Tool interactively, or in batch mode?

  Debug Tool gives you the option of conducting an interactive debugging session using a terminal with full-screen capabilities. This allows you to debug your batch as well as CICS applications interactively.

- Do you need to reduce file I/O?

  If the source (for C) and the source listing (for COBOL and PL/I) files are held as SAM ESDS files or sequential disk files, they should be defined with a suitable block size to minimize file I/O when using Debug Tool. Debug Tool will handle block sizes up to 8192 bytes for these types of files.

A Sample Interactive Debug Tool Session (COBOL)

This section provides a sample of a full-screen Debug Tool session to help you understand the support provided for debugging your programs.

The sample job, EQAWIVC2, is an example of a Debug Tool session for a COBOL program. This job is provided with Debug Tool to verify installation of the product. See your Systems Programmer to find where the source of the sample job is installed on your system. If your programming language is C or PL/I you can obtain other installation verification jobs in those languages. These are documented in *Debug Tool for VSE/ESA Installation and Customization Guide*.

The sample job EQAWIVC2 compiles, link-edits, and runs a COBOL/VSE program, IVPCOB2, that invokes Debug Tool interactively. Modify the sample job control to meet your requirements before submitting the job. Figure 2 on page 6 shows an extract from EQAWIVC2; tailoring requirements for each statement are discussed in the notes following Figure 2.

```
* $$ JOB JNM=EQAWIVC2,CLASS=Z                  <---- 1
// JOB EQAWIVC2                                <---- 2
*                                              <---- 3
*                                                    4
*                                                    5
*
// LIBDEF *,SEARCH=(lib.userlib,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=lib.userlib                  6
// OPTION CATAL                                      4
*
*       Step 1: Compile IVPCOB2
*
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM='EXIT(PRTEXIT(EQALIST))'
 CBL QUOTE TEST NAME
   ⋮
COBOL/VSE program source
   ⋮
/*
*       Step 2: Link-Edit IVPCOB2
*
// EXEC LNKEDT
/*
*       Step 3: Run the IVP program invoking Debug Tool
*               for Interactive Debug
*                                                    7
*                                                    8
*                                                    9
*                                                    10
*
// EXEC IVPCOB2,SIZE=AUTO,PARM='/TEST(,SYSIPT,,MFI%vtamluid:*)'
      *
      * Command file Input
      * Note: Commands conform to COBOL syntax
      *
      * Set animated execution at 1 command per second
        SET PACE 1;
      * Set a breakpoint at line 86
        AT 86 LIST ( "At the breakpoint for line", %LINE );
      * Start execution of the program
        GO;
      * Monitor VARBL1. It will appear in the Monitor window
        MONITOR LIST VARBL1;
      * List the contents of STR1
        LIST STR1;
      * Step through the next 6 statements to be executed.
      * These will run in animated mode
        STEP 6;
/*
/&
* $$ EOJ
```

Figure 2. Job Control Extract for EQAWIVC2

1 Modify the VSE/POWER job control statements for your site.

2 Modify the job card as appropriate for your site.

3 The statements beginning with an asterisk (*) will appear as comments on the system console. You may want to modify or remove them (if you replace the asterisk with `/. C`, a job control label, these will not appear on the console).

4 Change *lib.userlib* to your temporary user sublibrary:

- As invoked in this example, the compiler listing exit EQALIST, supplied with Debug Tool, writes the compiler listing to sublibrary member IVPCOB2.LIST in the first sublibrary in the LIBDEF SOURCE search chain. If you put a sublibrary other than *lib.userlib* as the first sublibrary in

the SOURCE search chain, then this is where the compiler listing exit writes the compiler listing file.

For more information about EQALIST, see "Debug Tool Compiler Print Exit" on page 23.

- The phase IVPCOB2 created by the link-edit in step 2 is written to this sublibrary.
- Debug Tool writes a profile settings file (*userid*.DTSAFE) to the first sublibrary in the LIBDEF SOURCE search chain. The DTSAFE member is used to save the Debug Tool session settings when your debug session terminates. These settings are used as your defaults for future debugging sessions.

  If you put a sublibrary other than *lib.userlib* as the first sublibrary in the SOURCE search chain, then this is where Debug Tool writes the profile settings file.

  The *userid* of the profile settings file depends on your VSE/ESA system options. This is described in "Determining the Default Userid" on page 30

5 If necessary, change the sublibrary to match the sublibrary where Debug Tool and COBOL/VSE are installed (Debug Tool and COBOL/VSE might not be installed in the same sublibrary).

6 If necessary, change the sublibrary to match the sublibrary where LE/VSE is installed.

7 Any messages produced by EQAWIVC2 will be in your LE/VSE installation default national language.

If you want messages to be produced in a national language other than your LE/VSE installation default, you can use the LE/VSE NATLANG run-time option. To do this, following the slash (/) at the beginning of the options string, add `NATLANG(xxx)`, followed by a blank, where *xxx* is the three character representation of the language you want to use (such as JPN, ENU, or UEN).

8 SYSIPT indicates to Debug Tool that the primary commands file for this session is the system input device. That is, the commands are included in the job stream (immediately after the EXEC statement).

Other possible options for the primary commands file are a filename, a file-id, a member of a sublibrary, or no commands file at all. These are explained in "Run-Time TEST Option Syntax" on page 33.

9 Change the *vtamluid* in the `TEST` option to the VTAM LU name of the terminal where your interactive debug session will run. For example, if your VTAM LU name is D0800001, change the `TEST` option to `TEST(,SYSIPT,,MFI%D0800001:*)`.

Note: The VTAM LU must not be in session with another application when Debug Tool attempts to acquire it.

10 The asterisk (*) indicates to Debug Tool that there is no preferences file for this session.

Other possible options for the preferences file are a filename, a file-id, a member of a sublibrary, or the system input device (SYSIPT). These are explained in "Run-Time TEST Option Syntax" on page 33.

Note: If you have both a primary commands file and a preferences file, the commands in the preferences file will be executed first. If both files are specified as SYSIPT, you need to separate the two input streams by an end of input (/*) job control statement.

After you modify the EQAWIVC2 job, submit it. Step 1 compiles the sample program and uses the print exit EQALIST supplied with Debug Tool to write a blank-compressed copy of the compiler listing to member IVPCOB2.LIST in the first sublibrary of the LIBDEF SOURCE search chain (see **4** in the notes for Figure 2 on page 6). In the sample job this is *lib.userlib*. This member is used during the interactive debug session to display the Source window.

Step 2 link-edits the phase IVPCOB2, and step 3 invokes IVPCOB2. When the execution of the program IVPCOB2 starts, following successful completion of the link-edit step, an interactive debug session starts on the terminal you designated in the TEST option at **7**. The Debug Tool commands specified in the primary commands file (SYSIPT) are then executed before you enter commands interactively. After the commands in the command file have executed the debug screen looks like the example shown in Figure 3.

Note: The example screen is for a 32-line display and assumes the default settings are being used for Debug Tool. If your display has a different number of lines or is not using the default settings, the display you see will be slightly different to the example.

```
COBOL    LOCATION: IVPCOB2 :> 92.1
COMMAND ===>                                                   SCROLL ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 1 OF 1
******************************* TOP OF MONITOR ********************************
0001  1 VARBL1   11
****************************** BOTTOM OF MONITOR ******************************








SOURCE: IVPCOB2 --1----+----2----+----3----+----4----+----5----+ LINE: 87 OF 96
      87             MOVE "BEG" TO STR2                                       .
      88             MOVE "UP" TO STR3                                        .
      89             ADD 1 TO VARBL1                                          .
      90             SUBTRACT 2 FROM VARBL2                                   .
      91             ADD 1 TO R                                               .
      92             MOVE "BOT" TO STR1                                       .
      93             MOVE "END" TO STR2 MOVE "DOW" TO STR3                    .
      94           END-PERFORM.                                               .
      95           MOVE "DONE" TO STR1. MOVE "END" TO STR2. MOVE "FIN" TO ST .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6 LINE: 9 OF 14
0009 %LINE = 86.1
0010  MONITOR
0011    LIST VARBL1 ;
0012  LIST STR1 ;
0013 STR1 = 'ONE  '
0014  STEP 6 ;
PF  1:?         2:STEP       3:QUIT       4:LIST       5:FIND       6:AT/CLEAR
PF  7:UP        8:DOWN       9:GO        10:ZOOM      11:ZOOM LOG  12:RETRIEVE
```

Figure 3. Debug Session Screen for IVPCOB2 after Command File Execution

You are now ready to enter commands in your interactive debug session. The following script leads you through the session. At the command line in your interactive debug session enter the commands shown following the ===> and press Enter.

1. The initial Source window is pointing at statement 92 and the value of VARBL1 in the Monitor window is 11. Display the contents of STR1:

   ===> list str1

   The Source window remains at statement 92, the Monitor window remains unchanged, and the Log window shows:

   ```
   0015  LIST STR1 ;
   0016 STR1 = 'TOP  '
   ```

2. Describe the attributes of STR1:

   ===> describe attributes str1

   The Source window remains at statement 92, the Monitor window remains unchanged, and the Log window shows:

   ```
   0017  DESCRIBE ATTRIBUTES STR1 ;
   0018    ATTRIBUTES FOR STR1
   0019      ITS LENGTH IS 5
   0020      ITS ADDRESS IS 00521DE8
   0021      77 IVPCOB2:>STR1   X(5) DISP
   ```

3. Cause the next statement to be executed:

   ===> step 1

   The Source window moves to statement 93, the Monitor window remains unchanged, and the Log window shows:

   ```
   0022  STEP 1 ;
   ```

4. Define a temporary variable TEMP, set it to the value of STR1, and display the contents of TEMP:

   ===> 77 temp pic x(5)
   ===> move str1 to temp
   ===> list temp

   The Source window remains at statement 93, the Monitor window remains unchanged, and the Log window shows:

   ```
   0023  77 TEMP PIC X(5) ;
   0024  MOVE STR1 TO TEMP ;
   0025  LIST TEMP ;
   0026 TEMP = 'BOT  '
   ```

5. Cause execution to continue until the next breakpoint is encountered:

   ===> go

   The Source window moves to statement 86, the Monitor window remains unchanged, and the Log window shows:

   ```
   0027  GO ;
   0028 At the breakpoint for line
   0029 %LINE = 86.1
   ```

6. Clear the breakpoint at line 86, set a breakpoint at exit of the program, and cause execution to continue:

   ===> clear at 86
   ===> at exit * list ("Exiting ",%cu)
   ===> go

   The Source window moves to statement 1, the value of VARBL1 in the Monitor window changes to 18, and the Log window shows:

   ```
   0030  CLEAR AT 86 ;
   0031  AT EXIT *
   0032    LIST ( "Exiting ", %CU ) ;
   0033  GO ;
   0034 Exiting
   0035 %CU = IVPCOB2
   ```

7. Exit Debug Tool:

   ===> quit

   The Source window remains at statement 1, the Log window shows:

   ```
   0036  QUIT ;
   ```

   and the Monitor window is replaced with:

```
COBOL    LOCATION: IVPCOB2 EXIT
COMMAND ===>                                                  SCROLL ===> PAGE
*******************************************************************************

DO YOU REALLY WANT TO TERMINATE THIS SESSION? N
ENTER     Y FOR YES   AND   N FOR NO

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

   Overtype the N with Y and press Enter. Your interactive debug session now terminates and the terminal returns to your default VTAM display.

If successful, the job finishes with return code 2 indicating there were unresolved weak external references during the link-edit step. This return code is normal and does not indicate a problem.

Your interactive debug session creates a log on SYSLST that contains the results of your interactive debug session (Figure 4 on page 11).

```
* DEBUG TOOL FOR VSE/ESA Version 1 Release 1 Mod 0
* 12/13/96 11:30:29 AM
* (C) COPYRIGHT IBM CORP. 1992, 1996
  SET PACE 1 ;
  AT 86
    LIST ( "At the breakpoint for line", %LINE ) ;
  GO ;
* At the breakpoint for line
* %LINE = 86.1
  MONITOR
    LIST VARBL1 ;
  LIST STR1 ;
* STR1 = 'ONE  '
  STEP 6 ;
  LIST STR1 ;
* STR1 = 'TOP  '
  DESCRIBE ATTRIBUTES STR1 ;
*   ATTRIBUTES FOR STR1
*     ITS LENGTH IS 5
*     ITS ADDRESS IS 00521DE8
*     77 IVPCOB2:>STR1   X(5) DISP
  STEP 1 ;
  77 TEMP PIC X(5) ;
  MOVE STR1 TO TEMP ;
  LIST TEMP ;
* TEMP = 'BOT  '
  GO ;
* At the breakpoint for line
* %LINE = 86.1
  CLEAR AT 86 ;
  AT EXIT *
    LIST ( "Exiting ", %CU ) ;
  GO ;
* Exiting
* %CU = IVPCOB2
*  QUIT ;
```

Figure 4. SYSLST Log Content after Execution of IVPCOB2

Chapter 2. Preparing to Debug Your Program

This chapter describes how to prepare your programs for debugging with Debug Tool. It discusses how to compile your programs using the TEST compile-time option to furnish Debug Tool with the necessary debugging information.

Information for using the TEST option with each language compiler and debugging multilanguage programs is discussed separately in the following sections:

- "Compiling a C Program with the Compile-Time TEST Option"
- "Compiling a COBOL Program with the Compile-Time TEST Option" on page 16
- "Compiling a PL/I Program with the Compile-Time TEST Option" on page 19
- "Debugging Multilanguage Programs" on page 22

Compiling a C Program with the Compile-Time TEST Option

Before testing your C program with Debug Tool, you must compile it with the C compile-time TEST option. This option causes the compiler to retain information about your application program that Debug Tool uses.

The TEST suboptions BLOCK, LINE, and PATH define the points where the compiler inserts program hooks. When you set breakpoints, they are associated with the hooks which are used to instruct Debug Tool where to gain control of your program.

The symbol table suboption SYM controls 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 using the C compile-time TEST option, be aware that:

- The C compile-time TEST option always generates entry and exit hooks for functions. If you specify TEST, these hooks are generated regardless of the suboptions you specify with TEST.
- The C compile-time TEST option implicitly specifies the GONUMBER option, which causes the compiler to generate line number tables corresponding to the input source file. You can explicitly remove this option by specifying NOGONUMBER.
- Programs compiled with both the TEST and OPT options do not have line hooks, block hooks, path hooks, or a symbol table generated, regardless of the TEST suboptions specified. Only function entry and exit hooks 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 specified take effect. For example, if you specify TEST(BLOCK, NOBLOCK, BLOCK, NOLINE, LINE), what takes effect is TEST(BLOCK, LINE) since 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

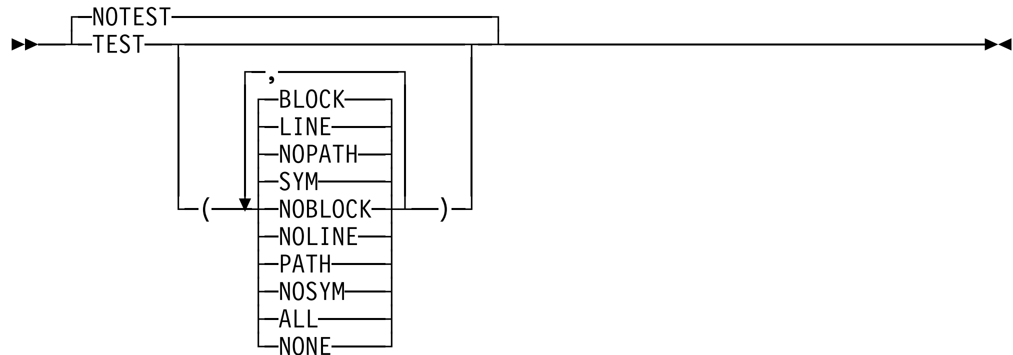
© Copyright IBM Corp. 1995, 1996

suboption causes the generation of entry and exit hooks. The PATH suboption also causes the generation of entry and exit hooks. However, only one hook is generated at each entry and exit.

See *IBM C for VSE/ESA User's Guide* for more information on the compile-time TEST option.

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

The syntax for the C compile-time TEST option is:



The compile-time TEST suboptions control the generation of symbol tables and program hooks Debug Tool needs to debug your program. The choices you make when compiling your program affect the amount of Debug Tool function available during your debugging session. When a program is under development, you should compile the program with TEST(ALL) to get the full capability of Debug Tool.

The following list explains what is produced by each option and suboption and how Debug Tool uses them when debugging your program:

TEST
Produces debugging information for Debug Tool to use during batch and interactive debugging. The extent of the information provided depends on which of the following 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 which have executable statements cannot exceed 1024.

If you do exceed these limits, the results from Debug Tool are undefined. Also, an LE/VSE dump generated from a program compiled with the TEST option yields incorrect line numbers and source file information.

NOTEST
Specifies that no debugging information is to be generated. That is, no statement hooks or path hooks are compiled into your program, no dictionary 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.

BLOCK
Inserts only block entry and exit hooks into your program's object output. A block is any number of data definitions, declarations, or statements enclosed within a single set of braces. BLOCK also creates entry and exit hooks for nested blocks. If SYM is enabled, symbol tables are generated for variables local to these nested blocks.

- You can only gain control at entry and exit of blocks.
- Issuing a command such as STEP causes your program to run, until it reaches the exit point.

NOBLOCK
Prevents symbol information and entry and exit 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
- Statements that begin in an #include file.

NOLINE
Suppresses the generation of statement (line number) hooks.

PATH
Hooks are generated at all path points.

- This option does not influence the generation of entry and exit hooks 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 path hooks are generated.

SYM
Generates symbol tables in the program's object output that gives 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
Block and line 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 all compiled-in hooks only at function entry and exit points. Block and line hooks are not inserted, and the symbol tables are suppressed.

TEST(NONE) is equivalent to TEST(NOLINE, NOBLOCK, NOPATH, NOSYM).

Placing Compiled-in Hooks for Functions and Nested Blocks

The following rules apply to the placement of compiled-in 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.

Placing Compiled-in Hooks for Statements and Path Points

The following rules apply to the placement of compiled-in 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.

Using #pragma to Specify Compile-Time TEST Option

The compile-time TEST/NOTEST option can be specified either when you invoke the compiler or directly in your program, using the `#pragma options` compiler directive.

The `#pragma` directive must appear before any executable code in your program.

Any options specified in the PARM parameter when invoking the compiler override those specified in the `#pragma`.

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

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

This is equivalent to TEST(SYM,BLOCK,LINE,NOPATH). The default LINE means that the LINE breakpoint will be triggered for a program containing the following statement:

```
#pragma options(test)
```

You can also use a `#pragma` to specify run-time options. This is explained, with examples, in "Specifying Run-Time TEST Option with #pragma runopts in C" on page 41.

For more information about `#pragma` options, refer to *IBM C for VSE/ESA Language Reference*.

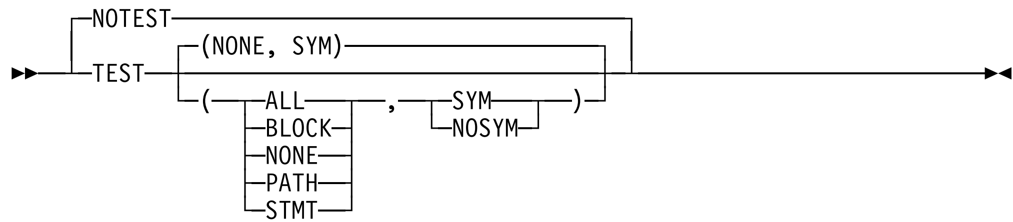
Compiling a COBOL Program with the Compile-Time TEST Option

When you compile with the TEST option, the compiler creates the dictionary tables that Debug Tool uses to obtain information about program variables, and inserts program hooks at selected points in your program. Your source is not modified. These points can be 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. Using these hooks, you can set breakpoints to instruct Debug Tool to gain control of your program at selected points during its execution.

When using the COBOL compile-time TEST option, be aware that:

- If you specify NUMBER with TEST, make sure the sequence fields in your source code all contain numeric characters.
- Usually, when you specify TEST, the compile-time options NOOPTIMIZE and OBJECT automatically go into effect, preventing you from debugging optimized programs. However, TEST(NONE, SYM) does not conflict with OPT, allowing limited debugging of optimized programs. See Appendix B, "Using Debug Tool in a Production Mode" on page 343 for more information on debugging production programs.
- The compile-time TEST option and the run-time DEBUG 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 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.

The syntax for the COBOL compile-time TEST option is:



Accepted abbreviations are TES and NOTES.

The compile-time TEST suboptions control the production of such debugging aids as dictionary tables and program hooks that Debug Tool needs to debug your program. The choices you make when compiling your program can affect the amount of Debug Tool function available during your debugging session. When a program is under development, you should compile the program with TEST(ALL) to get the full capability of Debug Tool. 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 to be generated. That is, no statement hooks or path hooks are compiled into your program, no dictionary 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 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 will be available during a debugging session.
- 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 batch and interactive debugging. The extent of the information provided depends on which of the following suboptions are selected.

ALL

Generates all compiled-in hooks, which includes 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.

- 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, nested programs, and PERFORM group of statements.
- Debug Tool can still be explicitly invoked 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.

- The GOTO command is valid for some statements and labels coinciding with path points.
- A call to CEETEST can still be used at any point to invoke Debug Tool.

PATH
Hooks are inserted at all path points.

- 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 still be used at any point to invoke 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, and at all entry and exit points.

- 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 dictionary tables in the program's object output (including the symbol table), that gives 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.
- SYM is required to support labels (paragraph or section names) as GOTO targets.

NOSYM
Suppresses the generation of dictionary 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).

Specifying TEST with no suboptions is equivalent to TEST(ALL, SYM).

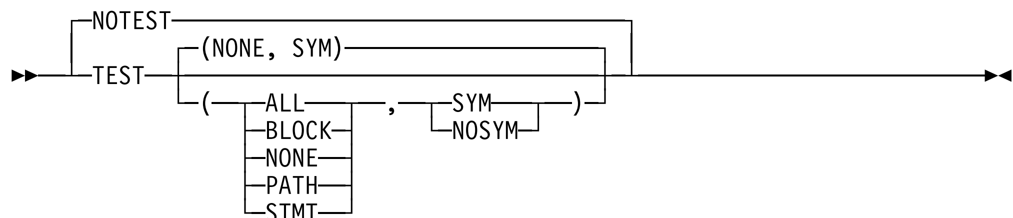
See *IBM COBOL for VSE/ESA Programming Guide* for more information about the compile-time TEST option.

Note: To be able to view your source code while debugging in interactive mode, you must direct the listing to a nontemporary file that is available during the debugging session. "Compiler Listings (and Program Source)" on page 23 describes how to make the listing file available for Debug Tool.

Compiling a PL/I Program with the Compile-Time TEST Option

The PL/I compiler provides support for Debug Tool under control of the TEST option and its hook location and symbol table suboptions. The hook location suboptions (BLOCK, STMT, PATH, ALL, and NONE) define the points at which the compiler inserts hooks. These program hooks allow Debug Tool to gain control at select points in a program during execution. The symbol table suboptions (SYM or NOSYM) controls the insertion of symbol tables into the program. Debug Tool uses the symbol tables to obtain information about program variables.

The syntax for the PL/I compile-time TEST option is:



Accepted abbreviations are TES and NOTES.

The choices you make when compiling your program can affect the amount of Debug Tool function available during your debugging session. When a program is under development, you should compile the program with TEST(ALL) to get the full capability of the Debug Tool. 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 to be generated. That is, no statement hooks or path hooks are compiled into your program, no symbol

tables are created, and Debug Tool does not have access to any symbol information.

- You can LIST storage and registers.
- You can include calls to PLITEST or CEETEST in your program to allow you to suspend program execution and issue Debug Tool commands.
- 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.
- Because statement hooks are not available, you cannot set any statement breakpoints or use commands such as GOTO or QUERY location. A statement table is available if compiled with STMT or GOSTMT.

TEST
Produces debugging information for Debug Tool to use during batch and interactive debugging. The extent of the information provided depends on which of the following suboptions are selected.

ALL
Generates all compiled-in hooks, which includes 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 of your program and all entry and exit points of internal program blocks.
- A call to PLITEST or CEETEST can still be used to invoke 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.
- Block hooks are not inserted into a NULL ON-unit or an ON-unit consisting of a single GOTO statement.

NONE
No hooks are inserted in the program.

- A call to PLITEST or CEETEST can still be used to invoke Debug Tool at any point in your program.

PATH
Causes hooks to be inserted:

- 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 to be compiled into the program. 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.
- SYM is required to support labels as GOTO targets.

NOSYM
Suppresses the generation of a symbol table. 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 (procedure or block name).

See *IBM PL/I for VSE/ESA Programming Guide* for more information about the compile-time TEST option.

Compiling with TEST(STMT), TEST(PATH), or TEST(ALL) also causes a statement number table to be generated. If the compile-time STMT option is in effect, TEST

causes GOSTMT to apply. If the compile-time NUMBER option is in effect, TEST causes GONUMBER to apply.

Note: To be able to view your source code while debugging in interactive mode, PL/I programs must be compiled using the PL/I SOURCE compile-time option. You must also direct the listing to a nontemporary file that is available during the debugging session. "Compiler Listings (and Program Source)" on page 23 describes how to make the listing file available for Debug Tool.

Debugging Multilanguage Programs

This section discusses strategies you can employ when debugging programs written in more than one language.

The process of debugging multilanguage programs is simplified by the introduction of LE/VSE. LE/VSE supports the creation of application programs written in more than one HLL by providing a single environment to run such programs using interlanguage communication (ILC).

When the need to debug a multilanguage program 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 LE/VSE.
- You need to debug an application written in more than one language, where not all of the languages are supported by LE/VSE.

When writing a multilanguage application, a number of special considerations arise because you must work outside the scope of any single language. The LE/VSE initialization process establishes an environment tailored to the set of HLLs constituting the main phase of your application program. This removes the need to make explicit calls to manipulate the environment. Also, termination of the LE/VSE environment is accomplished in an orderly fashion, regardless of the mixture of HLLs present in the application.

Debugging an Application Fully Supported by LE/VSE

If you are debugging an application written in a combination of languages supported by LE/VSE 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. For details, see "SET PROGRAMMING LANGUAGE" on page 317.

When defining session variables you want to access from compile units of different languages, you must define them with compatible attributes."Language Compatible Attributes" on page 246 contains a table showing compatible attributes for variables declared in the supported languages.

For more information on creating multilanguage applications, see *LE/VSE Writing Interlanguage Communication Applications*.

Debugging an Application Partially Supported by LE/VSE

Sometimes you might find yourself debugging applications that contain compile units written in languages not supported by either Debug Tool or LE/VSE. For example, you can run programs containing mixtures of Assembler, C, COBOL, FORTRAN, and PL/I source code with the Debug Tool. You can invoke Debug Tool and perform testing only for the sections of a multilanguage program written in a supported language and compiled with an LE/VSE-enabled compiler, or relink-edited to take advantage of LE/VSE library routines. If you are debugging a compile unit written in a supported language and the compile unit calls another unsupported language, a breakpoint set with AT CALL is triggered. Debug Tool can determine the name of the compile unit, but little else. Your compile unit runs unhindered by Debug Tool. When program execution returns to a compile unit of a known HLL, Debug Tool can once again gain control and execute commands.

Compiler Listings (and Program Source)

In order for you to view the program you are debugging, Debug Tool must have access to a file containing the program source statements. For C, Debug Tool must have access to the program source. For COBOL or PL/I, Debug Tool must have access to the listing generated by the compiler.

The following sections describe two ways in which you can store COBOL and PL/I compiler listings so Debug Tool can access them.

Debug Tool Compiler Print Exit

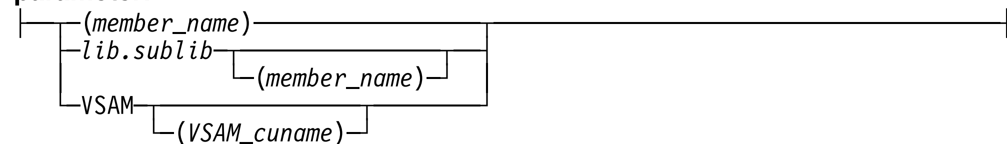
Debug Tool supplies the compiler print exit program EQALIST to assist you in storing the compiler listing for your COBOL and PL/I programs to disk files. EQALIST is invoked by using the COBOL or PL/I EXIT compile-time option, as follows:

```
// EXEC compiler,SIZE=compiler,                                   X
              PARM='EXIT(PRTEXIT([''parameter'',]EQALIST))'
```

The syntax of the EXIT compile-time option to invoke EQALIST is:

```
►►──EXIT(PRTEXIT(──┬──────────────────────────┬──EQALIST))──────────────►◄
                   └─''─┤ parameter ├─'',─┘
```

parameter:



The parameter passed to EQALIST indicates the type of file the compiler listing will be written to.

Note: As well as writing the listing to the file indicated, EQALIST will output it to SYSLST.

The following types of parameter can be passed to EQALIST:

(none)

If no parameter is passed to EQALIST, the compiler listing is written to a sublibrary member. The name of this member has the form *cuname*.LIST, where *cuname* is the name of the compile unit. This corresponds to the default sublibrary member name that Debug Tool searches for to obtain the source listing for *cuname*. The listing is written to the first sublibrary in the SOURCE search chain (specified using the LIBDEF JCL statement). If the member already exists, it will be overwritten.

The following is an example of invoking the PL/I compiler with the Debug Tool print exit to write the compiler source listing to a member in the first sublibrary in the SOURCE search chain:

```
// LIBDEF SOURCE,SEARCH=(LIB.SUBLIB,... )
// EXEC IEL1AA,SIZE=IEL1AA,PARM='EXIT(PRTEXIT(EQALIST))'
*PROCESS TEST(ALL);
 PLISAMP: PROC OPTIONS(MAIN);
```

The name of the member will be PLISAMP.LIST in the sublibrary LIB.SUBLIB.

member_name

If a parameter in the form (*member_name*) is passed to EQALIST, the compiler listing is written to a sublibrary member. The name of this member has the form *member_name*.LIST. The listing is written to the first sublibrary in the SOURCE search chain (specified using the LIBDEF JCL statement). For example, to write a PL/I compiler source listing to the member MYNAME.LIST in the first sublibrary in the SOURCE search chain, you would use the following statement to invoke the compiler:

```
// EXEC IEL1AA,SIZE=IEL1AA,PARM='EXIT(PRTEXIT(''(MYNAME)'',EQALIST))'
```

lib.sublib

If a parameter of the form *lib.sublib* is passed to EQALIST, the compiler listing is written to the sublibrary *lib.sublib*. The member name has the form *cuname*.LIST, where *cuname* is the name of the compile unit.

The following is an example of invoking the COBOL compiler with the Debug Tool print exit to write the compiler source listing to a member in the sublibrary MYLIB.MYSUBLIB:

```
// EXEC IGYCRCTL,SIZE=IGYCRCTL,                                       X
               PARM='EXIT(PRTEXIT(''MYLIB.MYSUBLIB'',EQALIST))'
 CBL TEST(ALL,SYM)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBPROG.
```

The name of the member will be COBPROG.LIST.

member_name

You can also specify the compile unit part of the sublibrary member name by including the preferred name in parentheses as part of the print exit parameter. For example, to write a PL/I compiler source listing to the member MYNAME.LIST in the sublibrary MYLIB.MYSUBLIB, you would use the following statement:

```
// EXEC IEL1AA,SIZE=IEL1AA,                                           X
               PARM='EXIT(PRTEXIT(''MYLIB.MYSUBLIB(MYNAME)'',EQALIST))'
```

VSAM

If the parameter VSAM is passed to EQALIST, the compiler listing is written to a SAM ESDS file in VSAM-managed space. The file-id of this file has the form DTVSE.*cuname*.LIST, where *cuname* is the name of the compile unit. If the filename of a VSAM user catalog is specified in the CAT parameter of the DLBL job control statement, the file will be cataloged in the specified VSAM user catalog. If the CAT parameter is not specified, the file will be cataloged in the job catalog (the VSAM catalog specified in the label information for IJSYSUC) or, if no job catalog is available, the VSAM master catalog. If the file already exists, it will be overwritten.

The following is an example of invoking the COBOL compiler with the Debug Tool print exit to write the compiler source listing to a SAM ESDS file:

```
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM='EXIT(PRTEXIT(''VSAM'',EQALIST))'
 CBL TEST(ALL,SYM)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBMAIN.
```

The name of the VSAM file will be DTVSE.COBMAIN.LIST.

VSAM_cuname

You can also specify the compile unit part of the file-id by including the preferred name in parentheses as part of the print exit parameter. For example, to write a PL/I compiler source listing to the file-id DTVSE.MYNAME.LIST you would use the following statement to invoke the compiler:

```
// EXEC IEL1AA,SIZE=IEL1AA,PARM='EXIT(PRTEXIT(''VSAM(MYNAME)'',EQALIST))'
```

For COBOL Only

The “batch compile” process in COBOL only produces one open/close call for the listing file, no matter how many END PROGRAM units there are. Thus, all compiler listings for the batch compile will be written to a file named after the first compile unit name.

The use of batch compiles using the EQALIST exit is not recommended, as a single output file for each compile unit will not be produced.

End of For COBOL Only

For PL/I Only

EQALIST can automatically generate a name for the PL/I compiler listing file by using the leftmost statement label of the first PROCEDURE statement in your source program. However, if you want EQALIST to automatically generate a name for the listing file, the PROCEDURE statement label must be in the first line of your program source (after any *PROCESS statement, but before any comments).

Batching multiple compilations (that is, multiple *PROCESS statements) in the one step with EQALIST active is not supported for storage reasons.

End of For PL/I Only

Note: EQALIST is an LE/VSE-enabled assembler program which creates an LE/VSE pre-initialized environment and then invokes the C program EQALISTC. Consequently the LE/VSE run-time library (including the C run-time support) must be included in the LIBDEF PHASE search chain when invoking the compiler with the EQALIST exit.

This will also require a much larger partition size, up to 1.5 MB more, than may be required for compilation without the print exit.

Assigning SYSLST to Disk

The source listing generated by the COBOL and PL/I compilers can also be saved on disk by assigning SYSLST to a sequential disk file prior to invoking the compiler. The file-id specified for the file should be of the form DTVSE.*cuname*.LIST, where *cuname* is the name of the compile unit.

The sequential disk file must be defined in the JCL used to initiate the Debug Tool full-screen session.

See “Determining the Default Userid” on page 30 for further details on files referenced by Debug Tool.

Chapter 3. Beginning a Debugging Session

This chapter explains how to begin a debugging session with Debug Tool. It describes:

- The files used by Debug Tool during a debugging session.
- The use and syntax of the run-time TEST option. The run-time TEST option gives you several alternatives for beginning a debugging session when specified during the invocation of your program.
- How to use the `#pragma runopts` to specify the run-time TEST option in C programs.
- How to use the PLIXOPT string to specify the run-time TEST option in PL/I programs.
- How to invoke Debug Tool from your program using the LE/VSE callable service CEETEST, the C function `__ctest()`, or the PL/I built-in subroutine PLITEST.

When Debug Tool is invoked using one of the methods described in this chapter, it interrupts the execution of your program to allow you to take appropriate actions. Debug Tool returns control to your program at the point of its interruption as the result of a GO or STEP command. You can also specify that control return to some other point in your program with the GOTO or GO BYPASS command. You can even specify that control be given to another program with the CALL command or a C function invocation.

If Debug Tool gains control because of a program condition, when control is returned to the program, the condition is raised in the program unless explicitly prevented (see “GO Command” on page 265).

Files Used By Debug Tool

Debug Tool uses a number of files to assist in the execution of a debugging session and to record the outcomes of that session.

Source and Listing Files

In order to display the source code for a compile unit in the Source window of a full-screen debugging session, Debug Tool must have access to a file containing either the source code or the compiler listing for that compile unit.

© Copyright IBM Corp. 1995, 1996

For C Only

For C compile units, Debug Tool requires a file containing the source code. By default, when Debug Tool encounters a new C compile unit, it looks for the source file or sublibrary member that was specified in the compile-time INFILE option when the source code was compiled. If a file (not a sublibrary member) was specified in the INFILE option, the necessary label information must be available for Debug Tool to read the file. If a sublibrary member was specified in the INFILE option, Debug Tool expects to find the member in the same sublibrary as it was in when it was compiled. If the INFILE option was not specified at compilation time, or INFILE(SYSIPT) was specified, Debug Tool does not look for a source file. Instead, you need to use the SET SOURCE command to specify the name of a sublibrary member where the source file can be found. For more information, see “SET SOURCE” on page 322.

End of For C Only

For COBOL and PL/I Only

For COBOL and PL/I compile units, Debug Tool requires a file containing the compiler listing. By default, when Debug Tool encounters a new COBOL or PL/I compile unit (*cuname*), it first looks for a sublibrary member named *cuname*.LIST in the source search chain. If this sublibrary member is not found, Debug Tool looks for a DLBL of a SAM ESDS file or a sequential disk file with a file-id of DTVSE.*cuname*.LIST. If a DLBL with a matching file-id is not found, Debug Tool looks for VSAM catalog entry for a SAM ESDS file with a file-id of DTVSE.*cuname*.LIST.

For information about how to create a compiler listing file, see “Compiler Listings (and Program Source)” on page 23.

End of For COBOL and PL/I Only

If the Source window is empty when Debug Tool starts, press LIST (PF4) with the cursor on the command line to display the Debug Tool Listing panel. The Listing panel indicates the name of the source or listing file that Debug Tool expected to find. With this panel you can overtype the displayed filename with the actual name of your source or listing file, and press QUIT (PF3). This causes Debug Tool to issue a SET SOURCE command specifying the source filename you entered. Alternatively, you can quit the Listing panel and, on the command line, enter the SET SOURCE command to specify the name of the source or listing file.

For more information, see “SET SOURCE” on page 322, as well as “SET DEFAULT LISTINGS” on page 307.

Preferences File

The preferences file is specified as a suboption of the TEST run-time option. It is processed by Debug Tool before any primary commands file and is useful for setting up the Debug Tool environment. This file will usually contain Debug Tool commands that you would use during every Debug Tool session. For example, you can use it to set up your preferred layout of the Debug Tool windows and the preferred screen colors for a full-screen session.

If no preferences file is specified in the TEST run-time option, Debug Tool looks for a default preferences file in the sublibrary member *userid*.DTPREF (for information on how *userid* is determined, see “Determining the Default Userid” on page 30). Debug Tool searches for this member in the sublibraries specified in the SOURCE search chain (specified using the LIBDEF JCL statement).

It is possible to use the standard input device, SYSIPT, as a preferences file. See page 41 for an example of this.

Note: If you do create a preference file you must remember to conform to the source format for the language of any of the main compile units you wish to debug.

Commands File

The commands file can be specified to Debug Tool as a suboption of the TEST run-time option or by the Debug Tool USE command.

- If a commands file is specified as part of the *commands_file* suboption of the TEST run-time option it is termed a primary commands file. The commands in a file specified in this way are executed to end of file.
- If a commands file is specified by the USE command, it is termed a USE file. Commands in a USE file (unless the USE command is included as part of a primary commands file) are executed until a “non-returning” command, such as GO, is encountered; any further commands in the USE file are ignored.

A commands file is the most convenient way of passing commands to Debug Tool for a batch debugging session. You can also use it to pass commands to multiple invocations of a particular application program.

It is possible to use the standard input device, SYSIPT, as a commands file. Note however that a commands file assigned to SYSIPT cannot be reused as a USE file. For an example of using SYSIPT as a commands file, see page 41.

Note: If you do create a commands file you must remember to conform to the source format for the language of any of the main compile units you wish to debug.

Profile Settings File

Debug Tool preserves debugging session profile settings and color customization in the profile settings file. The profile settings file is stored in a sublibrary member with the name *userid*.DTSAFE (for information on how *userid* is determined, see “Determining the Default Userid” on page 30).

When initializing a debugging session, Debug Tool searches the sublibraries specified in the SOURCE search chain (specified using the LIBDEF JCL statement) for member *userid*.DTSAFE, and reads the first occurrence of the member it finds

in the SOURCE search chain. It then uses the information stored in this member to set up the environment for the debugging session. If Debug Tool subsequently needs to update the profile settings file during a debugging session, it writes the revised settings to the same member in the same sublibrary.

If, when initializing a debugging session, Debug Tool does not find member *userid*.DTSAFE, at the end of the debugging session it creates the member in the first sublibrary in the SOURCE search chain.

The Log File

Debug Tool can record commands and their generated output in a session log file.

By default, Debug Tool writes the session log to the system output device, SYSLST. The default under CICS is no log file.

The results of command execution are logged as comments in the session log while the actual commands themselves are logged in the syntax of the current programming language. This means the session log can be used as a commands file for a later Debug Tool session without having to edit out the results from a previous run.

The SET LOG command can be used to change the log file from the default. Issuing the command SET LOG OFF suppresses output to the log file.

Determining the Default Userid

Unless otherwise specified, Debug Tool uses default names for preference and profile settings files, where the default is based on a userid. How the userid is determined depends upon whether you are running your debugging session in the batch environment or the CICS environment. Debug Tool determines the userid for a debugging session as described below.

Batch (non-CICS) Environment

In the batch environment, Debug Tool determines the userid as follows:

- For a VSE/ESA system IPLed with security checking active (SEC=YES specified in the IPL SYS command):
  - If a userid is associated with the job, Debug Tool uses that userid. The userid associated with a job can be provided by the following sources:
    - The userid specified in the // ID job control statement
    - The userid specified in the SEC parameter of the VSE/POWER * $$ JOB statement
    - The logon userid, if the job was submitted by a user logged on to the VSE/ESA Interactive Interface
    - The logon userid, if the job was submitted by a user logged on to VSE/ICCF
    - The logon userid, if the job was submitted by a user logged on to a workstation with the SEND/RECEIVE command interface

  – If a userid is not associated with the job, Debug Tool sets the userid to one of the following:
    - The name of the VTAM logical unit when debugging in full-screen mode session
    - The Debug Tool default value, DTVSE, when debugging in batch mode
- For a VSE/ESA system IPLed without security checking active (SEC=NO specified in the IPL SYS command), Debug Tool sets the userid to one of the following:
  – The name of the VTAM logical unit when debugging in full-screen mode session
  – The Debug Tool default value, DTVSE, when debugging in batch mode

CICS Environment

In the CICS environment, Debug Tool sets the userid to one of the following:

- The CICS logon userid, if available
- The Debug Tool default value, DTVSE, if the CICS logon userid is not available

Specifying Files to Debug Tool

A primary commands file and a preferences file can be passed to Debug Tool as suboptions of the TEST run-time option. In addition, the following commands can be used to specify source files, commands files, and log files to Debug Tool:

- PANEL LISTINGS
- PANEL SOURCES
- SET LOG
- SET SOURCE
- USE

A file specified to Debug Tool can be any of the following:

- A sublibrary member
- A SAM ESDS file
- A sequential disk file
- For a commands file, the system input device (SYSIPT)

Sublibrary Members

You indicate to Debug Tool that a file is a sublibrary member by enclosing the member name and type in parentheses and connecting the name and type with a period (.) character. For example, the following command tells Debug Tool to start writing the session log to the sublibrary member MYLOG.DTLOG:

```
SET LOG ON FILE (MYLOG.DTLOG)
```

If you want, you can qualify the sublibrary member name with the name of the sublibrary. You do this by specifying the library and sublibrary names, connected with a period (.) character, immediately before the left parenthesis preceding the member name. For example, the following command tells Debug Tool to start writing the session log to the sublibrary member MYLOG.DTLOG in sublibrary TEST.USER:

```
SET LOG ON FILE TEST.USER(MYLOG.DTLOG)
```

If a sublibrary member name is not qualified by a sublibrary name, Debug Tool needs to determine which sublibrary is to be used. Where Debug Tool searches for, or writes, a sublibrary member depends upon its member type. When Debug Tool is required to read from a sublibrary member, it searches for that member in the sublibraries specified by the LIBDEF JCL statement matching the member type. When Debug Tool must write to a sublibrary member, it either creates the member in the first sublibrary specified by the LIBDEF JCL statement matching the member type (if the member does not already exist), or rewrites the member in the same sublibrary (if the member already exists).

The following describes the sublibraries used according to member type:

Member Type	Sublibraries Used
PROC	Sublibraries in the PROCEDURE chain (LIBDEF PROC,SEARCH)
OBJ	Sublibraries in the OBJECT chain (LIBDEF OBJ,SEARCH)
Other	Sublibraries in the SOURCE chain (LIBDEF SOURCE,SEARCH)

Note: Do not use a member type of DUMP or PHASE when specifying a sublibrary member to Debug Tool. Using such a member type will cause an error when Debug Tool attempts to open the file.

SAM ESDS Files and Sequential Disk Files

You can specify a SAM ESDS file or sequential disk file to Debug Tool using either the filename (DLBL) or file-id. If a file-id could be interpreted as a filename it must be preceded with a slash (/).

The following examples use a SAM ESDS listing file with the filename MYLIST, and a sequential disk commands file with the file-id MY.COMMAND.FILE. The JCL for these files could be specified as follows:

```
// DLBL MYLIST,'COMPILER.LISTING.FILE',0,VSAM
// DLBL MYCMDS,'MY.COMMAND.FILE',0,SD
// EXTENT SYS045,DSKVL1
```

The following example tells Debug Tool to find the compiler source listing for compile unit *cuname* in the SAM ESDS file with the filename MYLIST:

```
SET SOURCE ON (cuname) MYLIST;
```

The following example tells Debug Tool to read commands from the sequential disk file with a file-id of MY.COMMAND.FILE:

```
USE my.command.file;
```

Under certain circumstances, Debug Tool is able to dynamically create a SAM ESDS file given just a file-id, without a corresponding DLBL job control statement. If you specify a new (non-existent) log file to Debug Tool using a file-id, Debug Tool will, if possible, dynamically create the log file as a SAM ESDS file. The new file will be cataloged in the VSAM job catalog (the VSAM catalog specified in the label information for IJSYSUC) or, if no job catalog is available, the VSAM master catalog. You do not need to provide a DLBL job control statement for Debug Tool to be able to dynamically create the file.

System Input Device (SYSIPT)

When specifying the name of a commands file, you can specify the system input device (SYSIPT) to Debug Tool as follows:

```
USE SYSIPT;
```

Using the Run-Time TEST Option

You can use the run-time TEST option to invoke Debug Tool and begin testing your program. The option is passed as an execution time parameter when you invoke your application program, as shown in the following examples.

For C, and PL/I:

```
// EXEC MYPROG,PARM='TEST(suboptions) / prog arg list'
```

For COBOL:

```
// EXEC MYPROG,PARM='prog arg list / TEST(suboptions)'
```

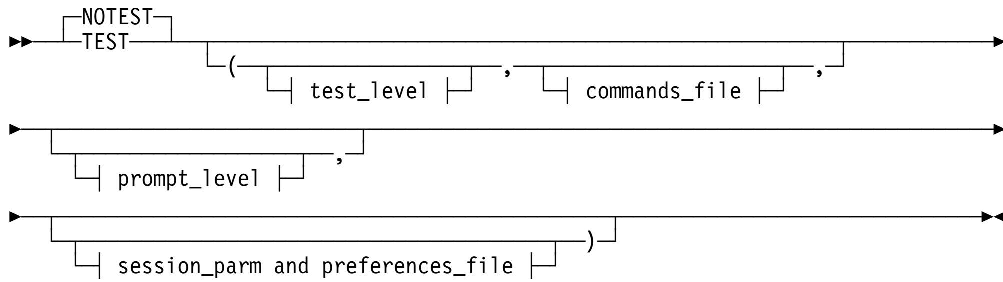
The simplest form of the TEST option is TEST with no suboption, however, suboptions provide you with more flexibility. There are four suboptions available:

- `test_level` (determines what HLL conditions raised by your program will cause Debug Tool to gain control)
- `commands_file` (determines which primary commands file is used as the initial source of commands in the absence of, or as an alternative to, a terminal)
- `prompt_level` (determines whether an initial commands list is unconditionally executed during program initialization)
- `session_parm and preferences_file` (provides the session parameter(s) 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)

Run-Time TEST Option Syntax

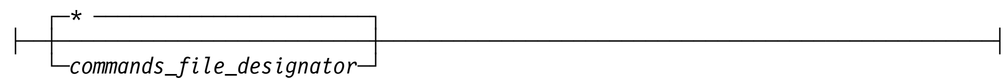
You can specify any combination of the run-time TEST suboptions, but they must be specified in the order presented, as shown in the following syntax diagram. Any option or suboption referred to as "default" is the IBM-supplied default, and might have been changed by your system administrator during installation. For examples of how to use TEST and each of its suboptions, see “Run-Time TEST Option Examples” on page 40.

The syntax for this option is:

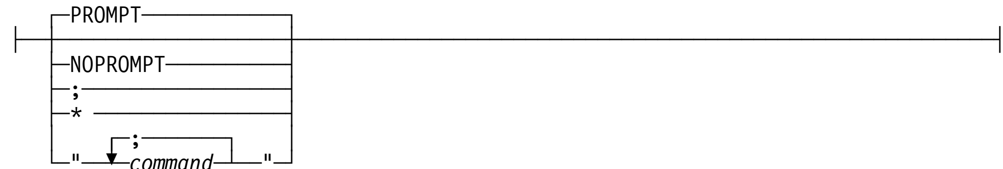


test_level:

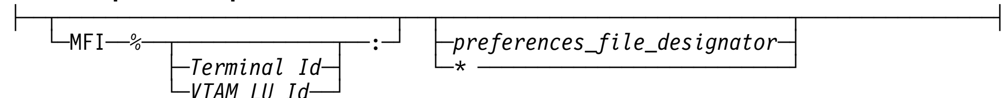
commands_file:



prompt_level:



session_parm and preferences_file:



NOTEST

Specifies that Debug Tool is not invoked at program initialization. However, invoking Debug Tool is still possible through the use of CEETEST, PLITEST, or the `__ctest()` function. In such a case, the suboptions specified with NOTEST are used when Debug Tool is invoked.

TEST

Specifies that Debug Tool is given control according to its suboptions. The TEST suboptions supplied will also be used if Debug Tool is invoked with CEETEST, PLITEST, or `__ctest()`.

test_level:

ALL (or blank)

Specifies that the occurrence of an attention interrupt, termination of your program (either normally or through an ABEND), or any program or LE/VSE condition of Severity 1 and above causes Debug Tool to gain control, regardless of whether a breakpoint is defined for that type of condition. If a condition occurs and a breakpoint exists for the condition, the commands specified in the breakpoint are executed. If a condition occurs and a breakpoint does not exist for that condition, or if an attention interrupt occurs, Debug Tool does the following:

- In interactive mode, it reads commands from a commands file (if it exists) or prompts you for commands
- In noninteractive mode, it reads commands from the commands file

For more information about attention interrupts, see "Requesting an Attention Interrupt During Interactive Sessions" on page 133.

ERROR
Specifies that only the following conditions cause Debug Tool to gain control without a user-defined breakpoint.

- An attention interrupt
- Program termination
- A predefined LE/VSE condition of Severity 2 or above
- Any C condition other than SIGUSR1, SIGUSR2, SIGINT, or SIGTERM.

LE/VSE conditions are described in *LE/VSE Debugging Guide and Run-Time Messages*.

If a breakpoint exists for one of the above conditions, commands specified in the breakpoint are executed. If no commands are specified, Debug Tool reads commands from a commands file or prompts you for them in interactive mode.

NONE
Specifies that Debug Tool gains control from a condition only if a breakpoint is defined for that condition. If a breakpoint does exist for the condition, the commands specified in the breakpoint are executed. An attention interrupt does not cause Debug Tool to gain control unless Debug Tool has previously been invoked. For information about how to change the TEST level after you start your session, see "SET TEST" on page 323.

commands_file:

*** (or blank)**
Indicates that no primary commands file is supplied; the terminal, if available, is used as the source of Debug Tool commands.

commands_file_designator
Is the valid designation for the primary commands file which is used as the initial source of commands after the preferences file. The primary commands file can be one of:

- A filename (DLBL). If the specified filename is longer than seven characters, it is automatically truncated, but no error message is issued.
- A file-id. If the file-id provided could be interpreted as a filename, it must be preceded with a slash (/) and enclosed in quotation marks, for example "/FILEID".
- A member of a sublibrary. The member name must be enclosed in brackets and the brackets further enclosed in double quotation marks ("), for example "(MEMBER.NAME)". If the sublibrary containing the member is in the LIBDEF search chain that corresponds to the member

type, the library name need not be specified. For information about which LIBDEF search chain is searched for a given member type, see "Sublibrary Members" on page 31.

- SYSIPT. The system input device.

If the designator might otherwise cause an ambiguity in the list of suboptions, you can enclose it in double quotation marks to differentiate it from the remainder of the list. If you are using a single file-id, no quotation marks are required.

The primary commands file designator has a maximum length of 80 characters.

Once the primary commands file is accessed as a source of commands, it continues to act in this capacity until all commands have been executed.

In interactive mode, when the end of the primary commands file is reached, Debug Tool prompts your terminal for commands until a QUIT command or the end of your application is reached.

In batch mode, Debug Tool reads and executes commands from the primary commands file until either the file runs out of commands or your program finishes running. If the end of the file is reached without encountering a QUIT command, Debug Tool forces a GO until the end of your program is reached.

It is possible to use a log file from one Debug Tool session as a source of commands for a subsequent session to regression test your application.

prompt_level:

PROMPT (or ; or blank)
Indicates that you want Debug Tool to be invoked immediately after LE/VSE initialization. Commands are read from the preferences file and then any designated primary commands file. In interactive mode Debug Tool then prompts your terminal for commands.

NOPROMPT (or *)
Indicates that you do not want Debug Tool invoked immediately after LE/VSE initialization. Instead, your application begins running.

command
Is one or more valid Debug Tool commands. Debug Tool is invoked immediately after program initialization, and then the command (or command string) is executed. The command string can have a maximum length of 250 characters, and must be enclosed in double quotation marks. Multiple commands must be separated by a semicolon.

Note: If you include a STEP or GO in your command string, none of the subsequent commands are processed.

session_parm and preferences_file:

MFI
Indicates you are using a 3270-type terminal for your debugging sessions.

Terminal Id
(For CICS only) specifies an up to four character Terminal Id which receives Debug Tool screen output during a dual terminal debugging session. The corresponding terminal should be in service and acquired ready to receive Debug Tool related I/O.

VTAM LU Id
Specifies an up to eight character VTAM logical unit identifier for a 3270-type terminal to be used for a full-screen interactive debugging session. The corresponding terminal must be known to the VSE/ESA system on which the batch debugging job is executing and not be in session with any application.

If Debug Tool encounters an error with the specified terminal it will issue the following messages and the application will complete execution with no further interaction from Debug Tool.

```
EQA1996E THE VTAM LOGICAL UNIT DEFINED FOR THE 3270 SESSION COULD
         NOT BE ACQUIRED.
EQA1987E DEBUGGER TERMINATED, EXECUTION CONTINUES
```

preferences_file_designator
Is the valid designation specifying the preferences file to be used by Debug Tool (see *commands_file_designator*, above, for the required format).

This file is read the first time Debug Tool is invoked, and must contain a sequence of Debug Tool commands to be executed.

If no preferences file is specified Debug Tool looks for a default preferences file in the sublibrary member *userid*.DTPREF. Debug Tool searches for this member in the sublibraries specified in the SOURCE search chain (specified using the LIBDEF JCL statement). “Determining the Default Userid” on page 30 describes how the value for *userid* is derived.

***** Specifies that no preferences file is supplied.

“Files Used By Debug Tool” on page 27 provides more details on the various files used by Debug Tool.

Run-Time TEST Option Considerations

When using the run-time TEST option, remember that:

- The LE/VSE 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 LE/VSE assembler user exit. Debug Tool uses the DTCN transaction in the CICS environment and customized LE/VSE user exit EQADCCXT that is link-edited with the application. For additional information see “Preparing and Using DTCN to Invoke Debug Tool under CICS” on page 109.

[1] If the object module for the source program is input to the linkage editor before the CEEUOPT object module, then the run-time options specified in the source program override the options specified in CEEUOPT. You can force the order in which object modules are included by the order in which linkage editor INCLUDE statements are specified.

3. Options specified at the invocation of the application, using the run-time TEST option, unless disabled by the LE/VSE option, NOEXECOPS.
4. Options specified within the source program (with `#pragma` or PLIXOPT) or application options specified with CEEUOPT and link-edited with the application.[1]
5. Option defaults specified at installation in CEEDOPT.
6. IBM-supplied defaults.

- Commands and suboptions are processed in the following order:
  1. Suboption defaults
  2. Commands in a preferences file (preferences_file suboption)
  3. A command string passed in the run-time options (prompt_level suboption)
  4. The initial set of commands, whether it consists of a command string
  5. Commands in a primary commands file (commands_file suboption)
  6. Commands entered at the command line (interactive mode)
- In 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.
- Suboptions can be specified with NOTEST. This means you can start your program using the NOTEST option and specify suboptions you might want to take effect later in your debugging session. The program begins running without Debug Tool taking control.

  To enable the suboptions you specified with NOTEST, invoke Debug Tool during your program's execution using a library service call such as CEETEST, PLITEST, or the `__ctest()` function.
- 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 actions 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.
- Once the primary commands file is accessed as a source of commands, it continues to act in this capacity until all commands have been executed or Debug Tool has ended. This differs from a commands file specified in a Debug Tool USE command 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 invoked from within a primary commands file take on the characteristics of the primary commands file and can be executed until complete (see “USE Command” on page 333 for a description of the USE command).
- In batch mode, when end-of-file is reached in your commands file, a GO command is forced at each request for a command until the program terminates. If another command is requested after program termination, a QUIT command is forced.
- If Debug Tool is invoked during program initialization, invocation occurs before the main prolog has completed. 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 GOTO cannot be used. However, references to static variables can be made.

If you enter STEP at this point, before entering any other commands, both program and LE/VSE initialization will complete and give you access to all variables. You can also enter all valid commands.

- If Debug Tool is invoked during program execution (for example, using a CEETEST call), it may not be able to find all compile units associated with your application. Compile units located in phases that are not currently active are not known to Debug Tool, even if they were executed prior to Debug Tool's initialization.

  Debug Tool does not know about compile units that were compiled without the TEST option, even if they are active. Nor does it know about compile units written in unsupported languages.

  For example,

  1. Phase PHASE1 contains compile units CU1 and CU2, both compiled with the TEST option.
  2. The compile unit CU1 calls CUX, contained in phase PHASE2, which returns after it completes processing.
  3. 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 it. Debug Tool does not recognize CUX.

- The results of execution of the initial set of commands whether it consists of a command string included in the run-time options or a primary commands file, are logged as comments in the session log. The session log can be used as a commands file without having to edit out the results from a previous run.
- The initial set of commands, 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 invoked from the primary commands file, a USE file takes on the characteristics of the primary commands file. See “USE Command” on page 333 for details.
- The initial command string is performed only once, when Debug Tool is first initialized.
- Commands in the preferences file are performed only once, when Debug Tool is first initialized.
- You can change the run-time TEST/NOTEST options at any time with the SET TEST command. See “SET TEST” on page 323.
- The primary commands file is shared across multiple enclaves. That is, if a new enclave starts commands will be executed from the current location in the primary commands file.

Run-Time TEST Option Examples

The following examples of using the run-time TEST option are provided to illustrate run-time options available for your programs. They do not illustrate complete commands. For more information on specifying run-time options, see “Invoking Your Program for a Debugging Session” on page 42, and *LE/VSE Programming Reference*.

- `NOTEST`

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

- `TEST`

  Specifying TEST with no suboptions causes a check for other possible definitions of the suboption. For example, C allows default suboptions to be selected at compile time using `#pragma runopts`. Similarly, PL/I offers the PLIXOPT string. LE/VSE provides the macro CEEXOPT. Using this macro, you can specify installation and program-specific defaults. For more information on using CEEXOPT, see *LE/VSE Programming Guide*.

  If no other definitions for the suboptions exist, the IBM-supplied default test level is (ALL,*,PROMPT).

- `TEST(ALL,*,*,*)`

  Debug Tool is not invoked initially; however, any condition raised in your program causes Debug Tool to be invoked, 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 invoked 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 invoked using CEETEST, PLITEST, or `__ctest()`.

- `TEST(ALL,"TEST.LIBRARY(COBOL.CMD)",PROMPT,PREFER)`

  - Debug Tool is invoked at the end of environment initialization, but before the main program prolog has completed.
  - The SAM ESDS file or the sequential disk file with the filename (DLBL) PREFER is processed as the preferences file.
  - Subsequent commands are found in the COBOL.CMD member of the sublibrary TEST.LIBRARY.

  If all commands in the commands file are processed and you issue a STEP command when prompted, or a STEP command is executed 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 COBOL.CMD repeating this process until end-of-file is reached. If the end of the file is reached without encountering a QUIT command, Debug Tool forces a GO until the end of your program is reached.

- `TEST(ALL,,,MFI%D0820001:)`

  For full-screen interactive debugging, Debug Tool is invoked on the VTAM logical unit D0820001 at end of the environment initialization.

- `TEST(ALL,SYSIPT,,MFI%D0840001:SYSIPT)`

  Debug Tool is invoked on the terminal D0840001 at the end of environment initialization. Both the preferences file and primary commands file are specified as SYSIPT. As the preferences file is read to end-of-file first, the job stream to execute a COBOL program in this case would look like:

```
// EXEC PROG,PARM='/TEST(ALL,SYSIPT,,MFI%D0840001:SYSIPT)'
SET COLOR ... ;  ◄── preferences file commands
/*
STEP 10;         ◄── commands file commands
/*
```

- `TEST(ALL,,,MFI%F000:)`

  For CICS dual terminal, Debug Tool is invoked on the terminal F000 at the end of the environment initialization.

Specifying Run-Time TEST Option with #pragma runopts in C

The run-time TEST option can be specified either when you invoke your program, or directly in your source by using this `#pragma`:

```
#pragma runopts (execops,test(suboption,suboption...))
```

This `#pragma` must appear before the first statement in your source file. When EXECOPS is specified, any options entered on the command line override those in the `#pragma`. For example, if you specified the following in the source:

```
#pragma runopts (execops,notest(all,*,prompt))
```

then invoked the program with the following parameter:

```
// EXEC program,PARM='TEST/'
```

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 specify NOEXECOPS, either by using a `#pragma` or with the compile-time EXECOPS option, no command line run-time options take effect.

For more information on `#pragma runopts`, see *IBM C for VSE/ESA User's Guide* and *LE/VSE Programming Guide*.

Specifying Run-Time TEST Option with PLIXOPT string in PL/I

The run-time TEST option can be specified either when you invoke your program, or directly in your source by using a PLIXOPT string:

```
DCL PLIXOPT CHAR(nn) VAR STATIC EXTERNAL INIT('TEST(suboption,suboption...)');
```

When EXECOPS is specified, any options entered on the command line override those in the PLIXOPT string. For example, if you specified the following in the source:

```
DCL PLIXOPT CHAR(nn) VAR STATIC EXTERNAL INIT('NOTEST(ALL,*,PROMPT)');
```

then invoked the program with the following parameter:

```
// EXEC program,PARM='TEST(,,,MFI%MYLUNAME:*)/'
```

the result would be:

```
TEST(ALL,*,PROMPT,MFI%MYLUNAME:*)
```

TEST overrides the NOTEST option specified in the PLIXOPT string and, because TEST does not contain values for the first three suboptions, suboptions ALL, *, and PROMPT remain in effect.

For more information on the PLIXOPT string, see *IBM PL/I for VSE/ESA Programming Guide* and *LE/VSE Programming Guide*.

Invoking Your Program When Starting a Debugging Session

After you have decided what level of testing you want to employ during your debugging session, you can invoke your program using the appropriate run-time TEST option. If you are using Debug Tool, this requires no special procedures (although certain considerations exist and are covered in "Invoking Your Program for a Debugging Session").

Invoking Your Program for a Debugging Session

To begin a debugging session, ensure your program has been compiled with the compile-time TEST option, and take the following steps:

1. 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 available. This might involve including them as part of a sublibrary search chain (specified using the LIBDEF JCL statement).
2. Include JCL statements to access all other files containing data your program needs.
3. Ensure the Debug Tool log file will be written to the correct location. By default Debug Tool writes the log file for a debugging session to the system output device, SYSLST. The session log file keeps a record of your debugging session, and can be used as a commands file during subsequent sessions. For more information on session log files, see "Using the Session Log File to Maintain a Record of Your Session" on page 87.
4. Start your program with the run-time TEST option, specifying the appropriate suboptions, or include a call to CEETEST, PLITEST, or `__ctest()` in the program's source. For more information about these calls, see "Using Alternative Debug Tool Invocation Methods" on page 43.

For more information about LE/VSE run-time options like TRAP(ON), see *LE/VSE Programming Reference*.

Invoking Debug Tool under CICS

To use Debug Tool under CICS, you need to ensure that you have completed all of the required installation and configuration steps for CICS, LE/VSE, and Debug Tool. See "Debugging CICS Programs" on page 107 and the appropriate product installation information.

You can invoke Debug Tool in three ways:

- **Single Terminal Mode**. Debug Tool displays its screens on the same terminal as the application. This can be set up using CEETEST, the run-time TEST option (specified using the CEEUOPT macro, the `pragma runopts` compiler directive, or the PLIXOPT string), or using DTCN.
- **Dual Terminal Mode**. Debug Tool displays its screens on a different terminal than the one used by the application. This can be set up using the run-time TEST option (specified using the CEEUOPT macro, the `pragma runopts` compiler directive, or the PLIXOPT string) or using DTCN.
- **Non-terminal 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 CEETEST, the run-time TEST option (specified using the CEEUOPT macro, the `pragma runopts` compiler directive, or the PLIXOPT string), or using DTCN.

See "Debugging CICS Programs" on page 107 for more details.

Using Alternative Debug Tool Invocation Methods

Debug Tool can also be invoked directly from within your program using one of the following methods:

- LE/VSE provides the callable service CEETEST which is invoked from LE/VSE-enabled languages.
- For C programs, you can use a `__ctest()` function call.

  Note: The `__ctest()` function is not supported in CICS.
- For PL/I programs, you can use a call to the PLITEST built-in subroutine.

To invoke Debug Tool using these alternatives, you still need to be aware of the TEST suboptions specified using NOTEST, CEEUOPT, or other "indirect" settings. See "Run-Time TEST Option Considerations" on page 37 for more information.

Invoking Debug Tool with CEETEST

Using CEETEST, you can invoke 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 invoke 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.

Usage Notes:

C
Include `leawi.h` header file.

PL/I
Include CEEIBMAW and CEEIBMCT. These files are installed with LE/VSE. The default installation sublibrary for this file is PRD2.SCEEBASE. See the example on page 48.

Batch and CICS Nonterminal Processes
It is strongly recommended that you use feedback codes (`fc`) when using CEETEST to initiate Debug Tool from a batch process or a CICS nonterminal task; otherwise results are unpredictable.

See *LE/VSE Programming Reference* for more details on the list of files used in C, COBOL, and PL/I programs when using the LE/VSE callable services (like CEETEST).

The syntax for CEETEST is:

For C

```
►►──void──CEETEST──(──┬────────────────────┬──,──┬────┬──)──────────────►◄
                      └─string_of_commands─┘     └─fc─┘
```

For COBOL

```
►►──CALL──"CEETEST"──USING──string_of_commands──,──fc───────────────────►◄
```

For PL/I

```
►►──CALL──CEETEST──(──┬─*──────────────────┬──,──┬─*──┬──)──────────────►◄
                      └─string_of_commands─┘     └─fc─┘
```

string_of_commands (input)
A halfword length prefixed string containing a Debug Tool command list, string_of_commands is optional.

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

If the string_of_commands is omitted, Debug Tool will read commands from the primary commands file, if available, or, in interactive mode, prompt you for commands.

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 debug tool was not available

Note: The CEE2F2 feedback code can also be obtained by batch applications or CICS nonterminal tasks getting allocation failures. For example, either the Debug Tool environment was corrupted or the debug event handler could not be loaded.

LE/VSE provides a callable service called CEEDCOD to help you decode the fields in the feedback code. Requesting the return of the feedback code is recommended. See *LE/VSE Programming Reference* for details.

For C and COBOL, if Debug Tool was invoked through CALL CEETEST the GOTO command is only allowed after Debug Tool has returned control to your program via STEP or GO.

The following examples show how to use CEETEST to invoke Debug Tool from each language:

Examples of CEETEST Function Calls for C

Example 1: In this example, a Null command string is passed to Debug Tool and a pointer to the LE/VSE feedback code is returned. If no other TEST run-time options have been compiled into the program, the call to CEETEST invokes Debug Tool with all defaults in effect. After it gains control, Debug Tool prompts you for commands.

```
#include <leawi.h>
#include <string.h>
#include <stdio.h>

int main(void) {
  _VSTRING  commands;
  _FEEDBACK fc;

  strcpy(commands.string, "");
  commands.length = strlen(commands.string);

  CEETEST(&commands, &fc);

}
```

Example 2: In this example, a string of valid Debug Tool commands is passed to Debug Tool and a pointer to the LE/VSE feedback code is returned. The call to CEETEST invokes 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, Debug Tool regains control at program termination and prompts you for commands. The command LIST(Z) is discarded when the command GO is executed.

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

```
#include <leawi.h>
#include <string.h>
#include <stdio.h>

int main(void) {
  _VSTRING  commands;
  _FEEDBACK fc;

  strcpy(commands.string, "AT LINE 23; {LIST(x); LIST(y);} "
                          "GO; LIST(z)");
  commands.length = strlen(commands.string);
⋮
  CEETEST(&commands, &fc);
⋮
}
```

Example 3: In this example, a string of valid 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 invoked 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, Debug Tool regains control at program termination and prompts you for commands.

```
#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);
      exit(2999);
   }
}
```

Examples of CEETEST Calls for COBOL

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.

For Debug Tool, remember to use the continuation character if your command exceeds 72 characters. See “Continuation (Full-screen mode)” on page 196.

```
77  FC                         Picture x(12) Value  ZEROES.
77  DebugTool                  Picture x(7)  Value 'CEETEST'.

01  Parms.
    AA                         Picture 99    Value 14.
    BB                         Picture x(11) Value 'LIST CALLS;'.

CALL DebugTool USING Parms FC.
```

Example 2: A string of commands is passed to Debug Tool when it is invoked. 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     Picture 99    Value 60.
   05 BB     Picture x(60) Value 'AT STATEMENT 23; LIST (x); LIST (y); GO;'.
```

In addition, the result of the call is returned in the feedback code, using a variable defined as:

```
77  fc                         Picture x(12).
```

in the data division of your program. You are not prompted for commands.

```
CALL "CEETEST" USING COMMAND-STRING fc.
```

Examples of CEETEST Calls for PL/I

Example 1: Assuming all required declarations have been made, no 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.

```
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 the TERMINATION breakpoint, where Debug Tool prompts for further commands.

```
DCL   ch  char(50)
      init('AT STATEMENT 10 DO; LIST(x); LIST(y); END; GO;');

DCL   1  fb,
         5  Severity  Fixed bin(15),
         5  MsgNo     Fixed bin(15),
         5  flags,
            8  Case   bit(2),
            8  Sev    bit(3),
            8  Ctrl   bit(3),
         5  FacID     Char(3),
         5  I_S_info  Fixed bin(31);


DCL   CEETEST  ENTRY  ( CHAR(*)  VAR  OPTIONAL,
         1   optional ,
           254 real fixed bin(15),     /*MsgSev*/
           254 real fixed bin(15),     /*MSGNUM*/
           254                         /*Flags*/,
             255 bit(2),               /*Flags_Case */
             255 bit(3),               /*Flags_Severity*/
             255 bit(3),               /*Flags_Control */
           254 char(3),                /*Facility_ID */
           254 fixed bin(31) )         /*I_S_Info  */
                 options(assembler) ;

CALL  CEETEST(ch, fb);
```

Example 3: This example assumes that you use predefined function prototypes and macros by including CEEIBMAW, and predefined feedback code constants and macros by including CEEIBMCT.

A command string is passed to Debug Tool which sets a breakpoint on every tenth executed statement. Once a breakpoint is set, 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 CEEIBMAW;
%INCLUDE CEEIBMCT;
DCL 01 FC FEEDBACK;

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

Declare CEEIBMCT  Character(8) Based;
Declare ADDR      Builtin;
%DCL FBCHECK ENTRY;
%FBCHECK:  PROC( fbtoken, condition ) RETURNS( CHAR );
   DECLARE
      fbtoken    CHAR,
      condition  CHAR;
RETURN('(ADDR('||fbtoken||')->CEEIBMCT = '||condition||')');
%END FBCHECK;
%ACT FBCHECK;
                      ---------- comment end --------------- */

Call CEETEST('AT Every 10 STATEMENT * Do; Q Loc; Go; End;'||
             'List AT;', FC);

If ¬FBCHECK(FC, CEE000)
   Then Put Skip List('----> ERROR! in CEETEST call', FC.MsgNo);
```

Invoking Debug Tool with the __ctest() Function

You can also use the C library routine `__ctest()` or `ctest()` to invoke Debug Tool. Add:

```
#include <ctest.h>
```

to your program to use the `ctest()` function.

Note: If you do not include `ctest.h` in your source or if you compile using the option LANGLVL(ANSI), you **must** use `__ctest()` function.

The `__ctest()` function is not supported in CICS.

When a list of commands is specified with `__ctest()`, Debug Tool runs the commands in that list. If you specify a null argument, Debug Tool gets commands by reading from the supplied commands file or by prompting you. If control returns to your application before all commands in the command list are run, the remainder of the command list is ignored. Debug Tool will continue reading from the specified commands file or prompt for more input.

If you do not want to compile your program with hooks, you can use `__ctest()` function calls to invoke 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:

```
►►──int──__ctest─(1)─(──char──*char_str_exp──)─────────────────────────►◄
```

Note:
[1] The syntax for ctest() and __ctest() is the same.

char_str_exp
Specifies a list of Debug Tool commands.

Examples of __ctest() Calls for C

Example 1: A null argument is passed to Debug Tool when it is invoked. 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 invoked. After it gains control, Debug Tool sets a breakpoint at statement 23 and 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 GO command.

```
__ctest("at line 23 {"
        "  list x;"
        "  list y;"
        "}"
        "go;"
        "list z;");
```

Example 3: Variable `ch` is declared as a pointer to character string and initialized as a string of commands. The string of commands is passed to Debug Tool when it is invoked. After it runs the string of commands, Debug Tool prompts you for more commands.

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

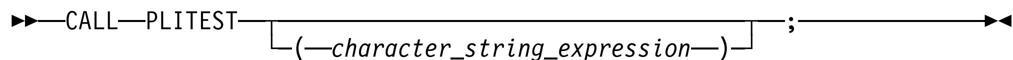
Example 4: A string of commands is passed to Debug Tool when it is invoked. After Debug Tool gains control, you are not prompted for commands. Debug Tool runs the commands in the command string and returns control to the program by way of the GO command.

```
#include <stdio.h>
#include <string.h>

char *ch = "at line 23 printf(\"x.y is %d\n\", x.y); go;";
char buffer[132];

strcpy(buffer, "at change x.y;");

__ctest(strcat(buffer, ch));
```

Invoking Debug Tool with PLITEST

For PL/I programs, the preferred method of invoking 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:



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.

Examples of PLITEST Calls for PL/I

Example 1: No argument is passed to Debug Tool when it is invoked. After gaining control, Debug Tool prompts you for commands.

```
CALL PLITEST;
```

Example 2: A string of commands is passed to Debug Tool when it is invoked. After gaining control, Debug Tool sets a breakpoint at statement 23, and returns control to the program. You are not prompted for commands. In addition, the LIST Y; command is discarded because of the execution of the GO command.

```
CALL PLITEST('At statement 23 Do; List X; End; Go; List Y;');
```

Example 3: Variable CH is declared as a character string and initialized as a string of commands. The string of commands is passed to Debug Tool when it is invoked. After it runs the commands, Debug Tool prompts you for more commands.

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

CALL PLITEST(ch);
```

Chapter 4. Debugging Your Programs in Full-Screen Mode

The most common features of Debug Tool are described in this chapter to help you get started using this tool to debug your programs. Language-specific examples and explanations of the most common tasks are provided to help you quickly gain a basic understanding of how to use Debug Tool.

The program function (PF) key definitions used in this chapter are based on the default settings for the keys.

Preparing for Debugging

Before using Debug Tool you must compile at least one part of your program with the compile-time TEST option. This inserts hooks, which are assembly instructions that you can see in an assembly listing. The execution of these hooks enables Debug Tool to gain control during program execution. A detailed description of the compile-time TEST option for each language is provided in Chapter 2, “Preparing to Debug Your Program” on page 12.

The simplest way to compile your program while you are learning to use Debug Tool is one of the following:

- For C, compile your program with TEST
- For PL/I and COBOL, compile your program with TEST(ALL,SYM)

Link your program as usual, except for programs to be run under CICS where member EQADCCXT must be included from the Debug Tool library.

For C Only

When running Debug Tool, links or calls to C programs linked with AMODE 24 are not supported if the C library phases CEEEV003 and EDCZ24 are loaded in the 31-bit SVA or in a partition that spans the 16MB line. To run any 24-bit C program under Debug Tool, CEEEV003 and EDCZ24 must be loaded below the line.

End of For C Only

Invoking Your Program with Debug Tool

To use Debug Tool in interactive mode from a batch program you need to include the Debug Tool library in your PHASE library search chain and invoke your program with the run-time TEST option as shown in the following example for C, and PL/I:

```
// EXEC MYPROG,PARM='TEST(,,,MFI%luname:) / prog arg list'
```

For COBOL, invoke your program as follows:

```
// EXEC MYPROG,PARM='prog arg list / TEST(,,,MFI%luname:)'
```

where `luname` is the VTAM logical unit name of the terminal you want the full-screen debugging session to become active on.

© Copyright IBM Corp. 1995, 1996

Contact your systems programmer if you do not know the name of the Debug Tool sublibrary on your system, or if you do not know the VTAM logical unit name of the terminal on which you want to run your debugging session.

For information about invoking your program with Debug Tool in batch (non-interactive mode) or CICS, see the appropriate sections in Chapter 7, "Using Debug Tool in Different Modes and Environments" on page 107.

Ending a Debug Session

When you have finished debugging your program, you can either press QUIT (PF3) or enter QUIT on the command line to end your Debug Tool session.

Basic Tasks of Debug Tool

This section describes how you interface to Debug Tool and describes how to navigate through the windows provided by Debug Tool. It also describes how to navigate through a debugging session and how to find help if you need it.

Debug Tool Interface

Debug Tool has a command line for issuing commands, and three windows:

- The Source window displays your source code
- The Log window records your commands and Debug Tool's responses
- The Monitor window continuously displays the values of monitored variables and other items depending on the command used.

Help

You can get help by either pressing ? (PF1) or entering a question mark (?) on the command line. This action lists all Debug Tool commands in the Log window. Putting a question mark after a partial command displays a list of possible subcommands. For example, enter on the command line:

```
?
WINDOW ?
WINDOW CLOSE ?
WINDOW CLOSE SOURCE
```

You can reopen the Source window with:

```
WINDOW OPEN SOURCE
```

Window Control

The relative layout of the Source, Monitor, and Log windows can be changed with the PANEL LAYOUT command. When you are displaying the windows you can resize the windows by typing WINDOW SIZE on the command line, moving the cursor to the new intersection point and then pressing Enter.

Finding Text

To find a string within a window, place the string to be searched for in double quotes (single quotes for a PL/I string) on the command line **without** pressing Enter, move the cursor into the window to be searched, then press FIND (PF5). Pressing FIND (PF5) will do repeat finds of the same string in the window where the cursor resides.

Scrolling

If the cursor is on the command line, you can page the Source window up by pressing UP (PF7) and down by pressing DOWN (PF8). To page through other windows, place the cursor in the desired window and press UP (PF7) or DOWN (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 ZOOM (PF10). Another ZOOM (PF10) will toggle back. ZOOM LOG (PF11) will toggle the Log window the same way without the cursor needing to be in the Log window.

You can scroll to an absolute line of the source file displayed in the Source window by using the SCROLL command. For example, your source file is in the Source window and you want to see line 188. To get there, enter the following command:

```
SCROLL TO 188
```

Changing Source Files

To change the code being viewed in the Source window, you can overtype the name after SOURCE: on the top line of the Source window with the desired name. This only works if the compilation unit (CU) is already known to Debug Tool

Alternately you can enter the command:

```
LIST NAMES CUS
```

to determine which CUs are known. A list of Compilation Units will be displayed in the Log window, as shown in the following examples.

Example of a List of C CUs:

```
THE FOLLOWING CUS ARE KNOWN IN *:
DD:LIBRARY.NAME(CALC.C)
DD:LIBRARY.NAME(PUSHPOP.C)
DD:LIBRARY.NAME(READTOKN.C)
```

Example of a List of COBOL or PL/I CUs:

```
THE FOLLOWING CUS ARE KNOWN IN *:
PLICALC
POP
PUSH
READTOK
```

You can overtype/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:

```
SET QUALIFY CU "DD:LIBRARY.NAME(READTOKN.C)";
```

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

Pressing LIST (PF4) with the cursor on the command line brings up the Source Identification Panel, where associations are made between source listings or source files shown in the source Window and their compile units. Overtype the Listings/Source File field with the new name.

Displaying the Halted Location

After displaying different source files and scrolling, you can go back to the halted execution point by entering the following command:

```
SET QUALIFY RESET
```

Setting a Line Breakpoint

Pressing AT/CLEAR (PF6) when the cursor is over a particular executable line in the Source window sets or clears a line breakpoint for that line. You can temporarily 'turn them off' with DISABLE and back on with ENABLE.

Stepping through or Running Your Program.

When Debug Tool comes up, none of your program has run yet.

Pressing STEP (PF2) runs your program, halting on the next hook encountered. If you compiled with TEST for C, or TEST(ALL,SYM) for COBOL or PL/I, STEP performs one statement.

Pressing GO (PF9) runs your program until a breakpoint is reached, the program ends, or a condition is raised.

Note: A condition being raised is determined by the setting of the run-time TEST suboption `test_level`.

If your program calls a function (or procedure or subroutine), you can use the STEP OVER command to run the called function without stepping into it; that is, the function is executed without intervention from Debug Tool. If you accidentally step into a function when you meant to step over it, issue the STEP RETURN command which steps to the return point (just after the call point). See “STEP Command” on page 327 for more details on this command.

Displaying a Variable's Value

To list the contents of a single variable, move the cursor to an occurrence of the variable name in the Source window and press LIST (PF4). The value of the variable is displayed in the Log window.

Continuously Displaying a Variable's Value

To continuously display or monitor a variables value, you can issue most LIST commands preceded by the word MONITOR. For example, enter:

```
MONITOR LIST num ;
```

the output for this command, in this case the contents of variable NUM, is continuously displayed in the Monitor window. The MONITOR command makes it easy to watch values while stepping through your program.

Setting a PF Key

Suppose you want to set PF9 to be the STEP OVER command with the message STEPOVER appearing under the PF9. key. You do it by entering:

```
 SET PF9 "STEPOVER" = STEP OVER;
```

Error Numbers for Messages in the log Window

When an error message shows up in the Log window, you can also get the message ID number to show up as

```
EQA1807E The command element d is ambiguous.
```

instead of

```
The command element d is ambiguous.
```

by modifying your profile. Use the PANEL PROFILE command and set SHOW MESSAGE ID NUMBERS to YES by overtyping.

For error message descriptions see Appendix F, “Debug Tool Messages” on page 355.

Finding a Renamed Source File Using Debug Tool

The name of the current source (or listing) file may have been changed since the program was compiled.

Pressing LIST (PF4) with the cursor on the command line brings up the Source Identification Panel, where associations are made between source listings or source files shown in the Source window and their compile units. Overtype the Listing/Source file field with the new name. If you need to do this repeatedly, 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.

Using a C Program to Demonstrate a Debug Tool Session

This section uses the information given thus far on Debug Tool's basic tasks and shows you how to apply them to your C applications by using an example C program (CALC) to demonstrate how they're used.

The CALC program is referred to in the following C Tasks section. It is a simple calculator which 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.

You can find the source for the following sample programs in the sublibrary member EQAWUGH1.Z in the Debug Tool installation sublibrary.

```
/*----- FILE CALC.H --------------------------------------------------*/
/*                                                                    */
/* Header file for CALC.C PUSHPOP.C READTOKN.C                        */
/* a simple calculator                                                */
/*--------------------------------------------------------------------*/
typedef enum toks {
  T_INTEGER,
  T_PLUS,
  T_TIMES,
  T_MINUS,
  T_DIVIDE,
  T_EQUALS,
  T_STOP
} Token;
Token read_token(char buf[]);
typedef struct int_link  {
  struct int_link * next;
  int i;
} IntLink;
typedef struct int_stack {
  IntLink * top;
} IntStack;
extern void  push(IntStack *, int);
extern int pop(IntStack *);
```

Figure 5. Sample C Program - Header File CALC.H

```
/*----- FILE CALC.C --------------------------------------------------*/
/*                                                                    */
/* A simple calculator which does operations on integers which       */
/* 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;
  for(;;)
  {
    tok=read_token(word);
    switch(tok)
    {
```

Figure 6 (Part 1 of 2). Sample C Program - main() Function

```
      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, pop(&stack)-num);
        break;
      case T_TIMES:
        push(&stack, pop(&stack)*pop(&stack) );
        break;
      case T_DIVIDE:
        num = pop(&stack);
        push(&stack, pop(&stack)/num);   /* 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;
}
```

Figure 6 (Part 2 of 2). Sample C Program - main() Function

```
/*----- 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;
}
```

Figure 7. Sample C Program - push() and pop() Functions

```
/*----- 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                                            */
```

Figure 8 (Part 1 of 2). Sample C Program - read_token() Function

```
/*------------------------------------------------------------------*/
static char nextchar(void)
{
/*------------------------------------------------------------------*/
/* input  action:                                                   */
/*    2      push 2 on stack                                        */
/*    18     push 18                                                */
/*    +      pop 2, pop 18, add, push result (20)                   */
/*    =      output value on the top of the stack (20)              */
/*    5      push 5                                                 */
/*    /      pop 5, pop 20, divide, push result (4)                 */
/*    =      output value on the top of the stack (4)               */
/*------------------------------------------------------------------*/
  char * buf_in  = "2 18 + = 5 / = ";
  static int index;     /* starts at 0 */
  char ret;
  ret = buf_in[index];
  ++index;
  return ret;
}
/*------------------------------------------------------------------*/
/* output: buf - null terminated token                              */
/* return: token type                                               */
/* action: reads chars through nextchar() and tokenizes them        */
/*------------------------------------------------------------------*/
Token read_token(char buf[])
{
  int i;
  char c;
  /* skip leading white space */
  for( c=nextchar();
       isspace(c);
       c=nextchar())
    ;
  buf[0] = c;  /* get ready to return single char e.g. "+" */
  buf[1] = 0;
  switch(c)
  {
    case '+' : return T_PLUS;
    case '-' : return T_MINUS;
    case '*' : return T_TIMES;
    case '/' : return T_DIVIDE;
    case '=' : return T_EQUALS;
    default:
      i = 0;
      while (isdigit(c)) {
        buf[i++] = c;
        c = nextchar();
      }
      buf[i] = 0;
      if (i==0)
        return T_STOP;
      else
        return T_INTEGER;
  }
}
```

Figure 8 (Part 2 of 2). Sample C Program - read_token() Function

C Tasks

The following sections identify typical tasks you might want to perform while using Debug Tool with your C program and explain how to accomplish these tasks. The CALC program is used to demonstrate some of these actions.

Setting a Breakpoint to Halt when Certain Functions Are Called

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 compile-time TEST option.

Note: 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.

Modifying the Value of a 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 LIST (PF4). The value is displayed in the Log window. This is equivalent to entering LIST TITLED `variable` on the command line. For instance, run the CALC program and, using STEP (PF2), step through the program to the statement labeled **CALC1** . Move the cursor over `num` and press LIST (PF4). The following appears in the Log window:

```
LIST ( num ) ;
num = 2
```

To modify the value of `num` to 22, overtype the `num = 2` line to `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 STEP (PF2) 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:

```
ATTRIBUTES for * ptr
  struct int_link {
    struct int_link *next;
    signed int i;
  }
```

You can use this action as a simple browser for 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 something like this:

```
LIST * ptr ;
(* ptr).next =       0x0
(* 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 ;
```

Stopping on a 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 set a simple line breakpoint because you will have to keep entering GO. For example, in `main` you want to stop at T_DIVIDE only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 39 { if(num != 0) GO; }
```

Line 39 is the statement labeled **CALC2** . The command will cause Debug Tool to stop at line 39. If the value of `num` is not 0, the program will continue. The command causes Debug Tool to stop on line 39 only if the value of `num` is 0.

Debugging When Only a Few Parts Are Compiled with TEST

Suppose you want to set a breakpoint at entry to function *push()* in 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 compilation 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.

When specifying compile units to Debug Tool you need to specify them in exactly the same format as Debug Tool displays them, however, if the name contains special characters, such as colons (:), you must enclose the name in double quotes ("). For example, if in response to the LIST NAMES CUS command Debug Tool displays the name for the compile unit PUSHPOP as DD:LIBRARY.NAME(PUSHPOP.C) you must enter the name as "DD:LIBRARY.NAME(PUSHPOP.C)".

Note: If the name appears in the Log window you can modify the line in the log and press Enter to put it on the command line.

If 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 PUSHPOP
AT ENTRY push;
GO ;
```

or

```
AT ENTRY PUSHPOP:>push
GO;
```

If it is not displayed, set an appearance breakpoint as follows:

```
AT APPEARANCE PUSHPOP ;
GO ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE PUSHPOP AT ENTRY push; GO;
```

The only purpose for this APPEARANCE breakpoint is to gain control the first time a function in the PUSHPOP compilation unit is run. When that happens, you can set a breakpoint at entry to *push()* like this:

```
AT ENTRY push;
```

Capturing Output to `stdout`

To redirect `stdout` to the Log window, issue the following command:

```
SET INTERCEPT ON FILE stdout ;
```

With this set, 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`.

Invoking Interactive Function Calls

You can invoke a library function (such as `strlen`) or one of the program functions interactively by calling it on the command line. In the next example, 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 will produce different results than before because of the additional value pushed on the stack.

Displaying Raw Storage

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) ;
```

Getting a 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 ;
```

For example, if you run the CALC example with the commands:

```
AT ENTRY read_token ;
GO ;
LIST CALLS ;
```

the log will contain something like:

```
At ENTRY in C function READTOKN :> read_token.
From LINE 18 in C function CALC :> main :> %BLOCK2.
```

which shows the traceback of callers.

Tracing the Run-Time Path for Code Compiled with TEST

To trace a program showing the entry and exit 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 member of a sublibrary, (STDOUT.CTRACE), that contains the following Debug Tool commands:

```
int indent;
indent = 0;
SET INTERCEPT ON FILE stdout;
AT ENTRY * { \
  ++indent; \
  if (indent < 0) indent = 0; \
  printf("%*.s>%s\n", indent, " ", %block); \
  GO; \
}
AT EXIT * {\
  if (indent < 0) indent = 0; \
  printf("%*.s<%s\n", indent, " ", %block); \
  --indent; \
  GO; \
}
```

You can use this file as the source of commands to Debug Tool by entering the following command:

```
USE (STDOUT.CTRACE)
```

If, after executing the USE file, you run the program listed below:

```
int foo(int i, int j) {
  return i+j;
}
int main(void) {
  return foo(1,2);
}
```

the following trace is displayed in the Log window:

```
stdout:  >main
stdout:   >foo
stdout:   <foo
stdout:  >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

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 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 which 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

To help find your uninitialized storage errors, run your program with the Language Environment run-time TEST and STORAGE options. In the following example:

```
// EXEC CALC,PARM='TEST(,,,MFI%luname:) 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 here 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 run-time STORAGE 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
```

Setting a Breakpoint to Halt before Calling a NULL Function

Calling an undefined function or calling a function through a function pointer which points to NULL is a severe error. To halt just before such a call is run, set this breakpoint:

```
AT CALL 0
```

When Debug Tool stops at this breakpoint, you can bypass the CALL by entering the GO BYPASS command. This allows you to continue your debugging session without raising a condition.

Using a COBOL Program to Demonstrate a Debug Tool Session

This section uses the information given thus far on Debug Tool's basic tasks and shows you how to apply them to your COBOL applications by using an example COBOL program (COBCALC) to demonstrate how they're used.

The COBCALC program is referred to in “COBOL Tasks” on page 71. It is a simple program which reads its input from the system console. The program calls a number of sub-programs to calculate a loan payment amount or the future value of a series of cash flows by utilizing several COBOL built-in functions.

You can find the source for the following sample programs in the sublibrary member EQAWUGC1.Z in the Debug Tool installation sublibrary.

```
      **********************************************************
      * FILE COBCALC.COBOL                                     *
      *                                                        *
      * A simple program which allows financial functions to   *
      * be performed using intrinsic functions.                *
      *                                                        *
      **********************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBCALC.
       ENVIRONMENT DIVISION.
```

Figure 9 (Part 1 of 2). Sample COBOL Program - Main Program COBCALC

```
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01  PARM-1.
          02 CALL-FEEDBACK     PIC XX.
        01  FIELDS.
          02 INPUT-1           PIC X(40).
        PROCEDURE DIVISION.
            DISPLAY "CALC Begins. Enter END to terminate."
              UPON CONSOLE.
            MOVE " " TO INPUT-1.
       * Keep accepting data until END requested
            PERFORM ACCEPT-INPUT UNTIL INPUT-1 EQUAL TO "END".
       * END requested
            DISPLAY "END requested." UPON CONSOLE
            DISPLAY "CALC Ends." UPON CONSOLE.
            GOBACK.
       * End of program.
       *
       * Accept input
       *
        ACCEPT-INPUT.
            DISPLAY "Functions are: END, LOAN, or PVALUE."
              UPON CONSOLE.
            ACCEPT INPUT-1 FROM CONSOLE.
       * Allow user to enter UPPER or lower case data
            EVALUATE FUNCTION UPPER-CASE(INPUT-1)  CALC1
              WHEN "END"
                MOVE "END" TO INPUT-1
              WHEN "LOAN"
                PERFORM CALCULATE-LOAN
              WHEN "PVALUE"
                PERFORM CALCULATE-VALUE
              WHEN OTHER
                DISPLAY "Invalid input: " INPUT-1 UPON CONSOLE
            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." UPON CONSOLE.
       *
       * 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." UPON CONSOLE.
```

Figure 9 (Part 2 of 2). Sample COBOL Program - Main Program COBCALC

```
*********************************************************
* FILE COBLOAN.COBOL                                    *
*                                                       *
* A simple subprogram which calculates payment amount   *
* for a loan.                                           *
*                                                       *
*********************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBLOAN.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  FIELDS.
   02 INPUT-1           PIC X(26).
   02 PAYMENT           PIC S9(9)V99 USAGE COMP.
   02 PAYMENT-OUT       PIC $$$$,$$$,$$9.99 USAGE DISPLAY.
   02 LOAN-AMOUNT       PIC S9(7)V99 USAGE COMP.
   02 LOAN-AMOUNT-IN    PIC X(16).
   02 INTEREST-IN       PIC X(5).
   02 INTEREST          PIC S9(3)V99 USAGE COMP.
   02 NO-OF-PERIODS-IN  PIC X(3).
   02 NO-OF-PERIODS     PIC 99 USAGE COMP.
 LINKAGE SECTION.
 01  PARM-1.
   02 CALL-FEEDBACK     PIC XX.
 PROCEDURE DIVISION USING PARM-1.
     MOVE "NO" TO CALL-FEEDBACK.
     DISPLAY "Enter Loan Amount, Interest and Number of Months sep
-    "arated by spaces" UPON CONSOLE.
     ACCEPT INPUT-1 FROM CONSOLE.
     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 PAYMENT TO PAYMENT-OUT.
     DISPLAY "Repayment Amount is: " PAYMENT-OUT UPON CONSOLE.
     MOVE "OK" TO CALL-FEEDBACK.
     GOBACK.
```

Figure 10. Sample COBOL Program - Subroutine COBLOAN

```
**********************************************************
* FILE COBVALU.COBOL                                     *
*                                                        *
* A simple subprogram which calculates present value     *
* for a series of cash flows.                            *
*                                                        *
**********************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBVALU.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CHAR-DATA.
   02 INPUT-1          PIC X(10).
   02 PAYMENT-OUT      PIC $$$$,$$$,$$9.99 USAGE DISPLAY.
   02 INTEREST-IN      PIC X(5).
   02 NO-OF-PERIODS-IN  PIC X(3).
 01 NUM-DATA.
   02 PAYMENT          PIC S9(9)V99 USAGE COMP.
   02 INTEREST         PIC S9(3)V99 USAGE COMP.
   02 COUNTER          PIC 99 USAGE COMP.
   02 NO-OF-PERIODS    PIC 99 USAGE COMP.
   02 VALUE-AMOUNT     OCCURS 99 PIC S9(7)V99 COMP.
 LINKAGE SECTION.
 01  PARM-1.
   02 CALL-FEEDBACK    PIC XX.
 PROCEDURE DIVISION USING PARM-1.
     MOVE "NO" TO CALL-FEEDBACK.
     DISPLAY "Enter Interest (Discount) Rate and Number of Periods
-      " separated by spaces." UPON CONSOLE.
     ACCEPT INPUT-1 FROM CONSOLE.
     UNSTRING INPUT-1 DELIMITED BY "," OR ALL " "
      INTO INTEREST-IN NO-OF-PERIODS-IN.
* Convert to numeric values
     COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).  VALU1
     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  VALU2
      VALUE-AMOUNT(ALL) ).
* Make it presentable
     MOVE PAYMENT TO PAYMENT-OUT.
     DISPLAY "Present Value is: " PAYMENT-OUT UPON CONSOLE.
     MOVE "OK" TO CALL-FEEDBACK.
     GOBACK.
*
* Get cash flows for each period
*
 GET-AMOUNTS.
     DISPLAY "Enter Value for period " COUNTER UPON CONSOLE.
     ACCEPT INPUT-1 FROM CONSOLE.
     COMPUTE VALUE-AMOUNT (COUNTER) = FUNCTION NUMVAL(INPUT-1).
```

Figure 11. Sample COBOL Program - Subroutine COBVALU

COBOL Tasks

The following sections identify typical tasks you might want to perform while using Debug Tool with your COBOL program and explain how to accomplish these tasks. The COBCALC program is used to demonstrate some of these actions.

Capturing I/O to the System Console

To redirect output that would normally appear on the System Console to your Debug Tool terminal, enter the following command:

```
SET INTERCEPT ON CONSOLE ;
```

This command will not only capture output directed to the System Console, but will also allow you to input data from your Debug Tool terminal instead of the System Console by using the Debug Tool INPUT command. For example, if you run COBCALC and issue the Debug Tool SET INTERCEPT ON CONSOLE command, followed by the GO command, you will see the following output displayed in the Debug Tool Log:

```
CONSOLE : CALC Begins. Enter END to terminate.
CONSOLE : Functions are: END, LOAN, or PVALUE.
CONSOLE : IGZ0000I AWAITING REPLY
THE PROGRAM IS WAITING FOR INPUT FROM CONSOLE
USE THE INPUT COMMAND TO ENTER 114 CHARACTERS FOR THE INTERCEPTED FIXED-FORMAT FILE.
```

You can then 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.

Setting a Breakpoint to Halt when Certain Functions Are Called

To halt just before `COBLOAN` is called, issue the command:

```
AT CALL COBLOAN ;
```

If the CU `COBVALU` is known to Debug Tool (it has previously been called), to halt just after `COBVALU` is called, issue the command:

```
AT ENTRY COBVALU ;
```

If the CU `COBVALU` is not known to Debug Tool (it has not previously been called), to halt just before `COBVALU` is entered the first time, issue the command:

```
AT APPEARANCE COBVALU ;
```

To take advantage of either of the above actions, you must compile your program with the compile-time TEST option.

Note: 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.

Modifying the Value of a 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 LIST (PF4). The value is displayed in the Log window. This is equivalent to entering LIST TITLED `variable` on the command line. For instance, run the COBCALC program to the statement labeled **CALC1**. Remember to reply to the program's input request by using the INPUT command. Move the cursor over INPUT-1 and press LIST (PF4). The following appears in the Log window:

```
 LIST ( INPUT-1 ) ;
INPUT-1 = 'some data                                '
```

To modify the value of INPUT-1 enter the command:

```
MOVE 'pvalue' to INPUT-1 ;
```

on the command line.

You can enter most COBOL expressions on the command line.

Now step into the call to COBVALU by pressing STEP (PF2) and step until the statement labeled **VALU1** is reached. (Respond to the message COBVALU issues by using the INPUT command to enter an interest rate followed by a number of periods). To view the attributes of variable INTEREST, issue the Debug Tool command:

```
DESCRIBE ATTRIBUTES INTEREST ;
```

The result in the Log window is:

```
 ATTRIBUTES FOR INTEREST
   ITS LENGTH IS 4
   ITS ADDRESS IS 00527008
   02 COBVALU:>INTEREST   S999V99 COMP
```

You can use this action as a simple browser for group items and data hierarchies.

For example, you can list all the values of the elementary items for the CHAR-DATA group with the command:

```
LIST CHAR-DATA ;
```

with results in the Log window appearing something like this:

```
 LIST CHAR-DATA ;
02 COBVALU:>INPUT-1  of 01 COBVALU:>CHAR-DATA  = '.12 5      '
INVALID DATA FOR 02 COBVALU:>PAYMENT-OUT  of 01 COBVALU:>CHAR-DATA   IS FOUND.
02 COBVALU:>INTEREST-IN  of 01 COBVALU:>CHAR-DATA  = '.12  '
02 COBVALU:>NO-OF-PERIODS-IN  of 01 COBVALU:>CHAR-DATA  = '5  '
```

Note: If you use the LIST command to list the contents of an uninitialized variable, or a variable that contains invalid data, Debug Tool will display INVALID DATA.

Stopping on a 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 set a simple line breakpoint because you will have to keep entering GO. For example, in COBVALU you want to stop at the calculation of present value only if the discount rate is less than -1 (before the exception occurs). Set the breakpoint like this:

```
AT 41 IF INTEREST > -1 THEN GO ; END-IF ;
```

Line 41 is the statement labeled **VALU2**. The command will cause Debug Tool to stop at line 41. If the value of INTEREST is greater than -1, the program will continue. The command causes Debug Tool to remain on line 41 only if the value of INTEREST is less than or equal to -1.

Debugging When Only a Few Parts Are Compiled with TEST

Suppose you want to set a breakpoint at entry to function COBVALU. COBVALU has been compiled with TEST but the other files have not. Debug Tool comes up with an empty Source window. To display the compilation 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 COBVALU 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 COBVALU
AT ENTRY COBVALU;
GO ;
```

If it is not displayed, set an appearance breakpoint as follows:

```
AT APPEARANCE COBVALU ;
GO ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE COBVALU AT ENTRY COBVALU; GO;
```

The only purpose for the APPEARANCE breakpoint is to gain control the first time a function in the COBVALU compilation unit is run.

Displaying Raw Storage

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 CHAR-DATA enter:

```
LIST STORAGE(CHAR-DATA,30)
```

Getting a 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 ;
```

For example, if you run the COBCALC example with the commands:

```
AT APPEARANCE COBVALU AT ENTRY COBVALU;
GO;
INPUT pvalue;
GO;
LIST CALLS;
```

the log will contain something like:

```
 AT APPEARANCE COBVALU
   AT ENTRY COBVALU ;
 GO ;
CONSOLE : CALC Begins. Enter END to terminate.
CONSOLE : Functions are: END, LOAN, or PVALUE.
CONSOLE : IGZ0000I AWAITING REPLY
THE PROGRAM IS WAITING FOR INPUT FROM CONSOLE
USE THE INPUT COMMAND TO ENTER 114 CHARACTERS FOR THE INTERCEPTED FIXED-FORMAT FILE.
 INPUT pvalue ;
 GO ;
 LIST CALLS ;
At ENTRY IN COBOL program COBVALU.
From LINE 57.1 IN COBOL program COBCALC.
```

which shows the traceback of callers.

Tracing the Run-Time Path for Code Compiled with TEST

To trace a program showing the entry and exit 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 member of a sublibrary, CALLS.COBTRC, that contains the following Debug Tool commands:

```
      * Commands in a COBOL USE file must be coded in columns 8-72.
      * If necessary, commands can be continued by coding a '-' in
      * column 7 of the continuation line.
       01 LEVEL PIC 99 USAGE COMP;
       MOVE 1 TO LEVEL;
       AT ENTRY * PERFORM;
         COMPUTE LEVEL = LEVEL + 1;
         LIST ( "Entry:", LEVEL, %CU);
         GO;
         END-PERFORM;
       AT EXIT * PERFORM;
         LIST ( "Exit:", LEVEL);
         COMPUTE LEVEL = LEVEL - 1;
         GO;
         END-PERFORM;
```

You can use this file as the source of commands to Debug Tool by entering the following command:

```
USE (CALLS.COBTRC)
```

If, after executing the USE file, you run the following program sequence:

```
COBMAIN:
  ...
  CALL "COBSUB"
  ...

COBSUB:
  ...
  CALL "COBSUB2".
  GOBACK.
  ...

COBSUB2:
  ...
  GOBACK.
  ...
```

the following trace, or something similar, is displayed in in the Log window:

```
Entry:
LEVEL = 00002
%CU = COBSUB
Entry:
LEVEL = 00003
%CU = COBSUB2
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.

Finding Unexpected Storage Overwrite Errors

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.

```
   02 FIELD-1          PIC X(8) OCCURS 2.
   02 FIELD-2          PIC X(8)
 PROCEDURE DIVISION.
*                    ( An invalid index value is set )
     MOVE 3 TO CTR.
     MOVE "TOO MUCH" TO FIELD-1( CTR ).
```

Find the address of FIELD-2 with the command

```
DESCRIBE ATTRIBUTES FIELD-2
```

Suppose the result is X'00521D42'. To set a breakpoint which watches for a change in storage values starting at that address for the next 8 bytes, issue the command

```
AT CHANGE %STORAGE(H'00521D42',8)
```

When the program is run, Debug Tool will halt if the value in this storage changes.

Setting a Breakpoint to Halt before Calling an invalid Program

Calling an undefined program is a severe error. If you have developed a main program which calls a subprogram which 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 debugging session without raising a condition.

Using a PL/I Program to Demonstrate a Debug Tool Session

This section uses the information given thus far on Debug Tool's basic tasks and shows you how to apply them to your PL/I applications by using an example PL/I program (PLICALC) to demonstrate how they're used.

The PLICALC program is referred to in “PL/I Tasks” on page 80. It is a simple calculator which 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.

You can find the source for the following sample programs in the sublibrary member EQAWUGP1.Z in the Debug Tool installation sublibrary.

```
plicalc: proc options(main);
/*----- file plicalc.pli--------------------------------------------*/
/*                                                                  */
/* A simple calculator which does operations on integers which      */
/* are pushed and popped on a stack                                 */
/*                                                                  */
/*------------------------------------------------------------------*/
dcl index  builtin;
dcl length builtin;
dcl substr builtin;
dcl sysprint print;
dcl 1 stack,
      2 stkptr fixed bin(15,0) init(0),
      2 stknum(50) fixed bin(31,0);
dcl 1 bufin,
      2 bufptr fixed bin(15,0) init(0),
      2 bufchr char (100) varying;
dcl 1 tok char (100) varying;
dcl 1 tstop char(1) init ('s');
dcl 1 ndx fixed bin(15,0);
```

Figure 12 (Part 1 of 2). Sample PL/I Program - Main Program PLICALC

```
dcl num      fixed bin(31,0);
dcl i        fixed bin(31,0);
dcl push entry external;
dcl pop  entry returns (fixed bin(31,0)) external;
dcl readtok entry returns (char (100) varying) external;
/*------------------------------------------------------------------*/
/* input  action:                                                   */
/*    2      push 2 on stack                                        */
/*    18     push 18                                                */
/*    +      pop 2, pop 18, add, push result (20)                   */
/*    =      output value on the top of the stack (20)              */
/*    5      push 5                                                 */
/*    /      pop 5, pop 20, divide, push result (4)                 */
/*    =      output value on the top of the stack (4)               */
/*------------------------------------------------------------------*/
bufchr = '2 18 + = 5 / =';
do while (tok ¬= tstop);
  tok = readtok(bufin);          /* get next 'token' */
  select (tok);
    when (tstop)
      leave;
    when ('+') do;
      num = pop(stack);
      call push(stack,num);      /*  CALC1  statement */
    end;
    when ('-') do;
      num = pop(stack);
      call push(stack,pop(stack)-num);
    end;
    when ('*')
      call push(stack,pop(stack)*pop(stack));
    when ('/') do;
      num = pop(stack);
      call push(stack,pop(stack)/num); /*  CALC2  statement */
    end;
    when ('=') do;
      num = pop(stack);
      put skip (num);
      call push(stack,num);
    end;
    otherwise do;/* must be an integer */
      num = atoi(tok);
      call push(stack,num);
    end;
  end;
end;
return;
```

Figure 12 (Part 2 of 2). Sample PL/I Program - Main Program PLICALC

```
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;
```

Figure 13. Sample PL/I Program - TOK Function

```
push: procedure(stack,num);
/*----- file push.pli ----------------------------------------------*/
/*                                                                  */
/* a simple push function for a stack of integers                   */
/*                                                                  */
/*------------------------------------------------------------------*/
dcl 1 stack connected,
      2 stkptr fixed bin(15,0),
      2 stknum(50) fixed bin(31,0);
dcl num      fixed bin(31,0);
stkptr = stkptr + 1;
stknum(stkptr) = num; /*  PUSH1  statement */
return;
end push;
```

Figure 14. Sample PL/I Program - PUSH Function

```
pop: procedure(stack) returns (fixed bin(31,0));
/*----- file pop.pli -----------------------------------------------*/
/*                                                                  */
/* 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;
```

Figure 15. Sample PL/I Program - POP Function

```
readtok: procedure(bufin) returns (char (100) varying);
/*----- file readtok.pli -------------------------------------------*/
/*                                                                 */
/* a function to read input and tokenize it for a simple calculator */
/*                                                                 */
/* action: get next input char, update index for next call         */
/* return: next input char(s)                                      */
/*-----------------------------------------------------------------*/
dcl length builtin;
dcl substr builtin;
dcl verify builtin;
dcl 1 bufin connected,
      2 bufptr fixed bin(15,0),
      2 bufchr char (100) varying;
dcl 1 tok char (100) varying;
dcl 1 tstop char(1) init ('s');
dcl 1 j fixed bin(15,0);
                                 /* start of processing */
if bufptr > length(bufchr) then do;
  tok = tstop;
  return ( tok );
end;
bufptr = bufptr + 1;
do while (substr(bufchr,bufptr,1) = ' ');
  bufptr = bufptr + 1;
  if bufptr > length(bufchr) then do;
    tok = tstop;
    return ( tok );
  end;
end;
tok = substr(bufchr,bufptr,1); /* get ready to return single char */
select (tok);
  when ('+','-','/','*','=')
    bufptr = bufptr;
  otherwise do;               /* possibly an integer */
    tok = '';
    do j = bufptr to length(bufchr);
      if verify(substr(bufchr,j,1),'0123456789') ¬= 0 then
        leave;
    end;
    if j > bufptr then do;
        j = j - 1;
      tok = substr(bufchr,bufptr,(j-bufptr+1));
      bufptr = j;
    end;
    else
      tok = tstop;
  end;
end;
return (tok);
end readtok;
```

Figure 16. Sample PL/I Program - READTOK Function

PL/I Tasks

The following sections identify typical tasks you might want to perform while using Debug Tool with your PL/I program and explain how to accomplish these tasks. The PLICALC program is used to demonstrate some of these actions.

Setting a Breakpoint to Halt when Certain Functions Are Called

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 either of the above actions, you must compile your program with the compile-time TEST option.

Note: 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.

Modifying the Value of a 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 LIST (PF4). The value is displayed in the Log window. This is equivalent to entering LIST TITLED `variable` on the command line. For instance, run the PLICALC program to the statement labeled **CALC1** . Move the cursor over NUM and press LIST (PF4). The following appears in the Log window:

```
 LIST NUM ;
NUM =             2
```

To modify the value of NUM to 22, overtype the NUM = 2 line to 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 STEP (PF2) and step until the statement labeled **PUSH1** is reached. To view the attributes of variable STKNUM, issue the Debug Tool command:

```
DESCRIBE ATTRIBUTES STKNUM;
```

The result in the Log window is:

```
 ATTRIBUTES FOR STKNUM
   ITS ADDRESS IS  005C1344 AND ITS LENGTH IS 200
     PUSH : STACK.STKNUM(50)  FIXED BINARY(31,0) REAL PARAMETER
         ITS ADDRESS IS  005C1344 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 =        1
 STACK.STKNUM(1)  =           0
 STACK.STKNUM(2)  =     6034884
 ...
 STACK.STKNUM(50) =     6034884
```

You can change the value of a structure member by issuing the assignment as a command as in the following example:

```
STKNUM(STKPTR) = 33;
```

Stopping on a 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 set a simple line breakpoint because you will have to keep entering GO. 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 32 DO; IF NUM ¬= 0 THEN GO; END;
```

Line 32 is the statement labeled **CALC2**. The command will cause Debug Tool to stop at line 32. If the value of NUM is not 0, the program will continue. The command causes Debug Tool to stop on line 32 only if the value of NUM is 0.

Debugging When Only a Few Parts Are Compiled with TEST

Suppose you want to set a breakpoint at entry to subroutine PUSH in file PUSH.LIST. PUSH has been compiled with TEST but the other files have not. Debug Tool comes up with an empty Source window. To display the compilation 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 compilation unit is run. When that happens, you can set a breakpoint at entry to PUSH like this:

```
AT ENTRY PUSH;
```

Displaying Raw Storage

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 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 ;
```

For example, if you run the CALC example with the commands:

```
AT ENTRY READTOK ;
GO ;
LIST CALLS ;
```

the log will contain something like:

```
At ENTRY IN PL/I subroutine READTOK.
From LINE 18.1 IN PL/I subroutine PLICALC.
```

which shows the traceback of callers.

Tracing the Run-Time Path for Code Compiled with TEST

To trace a program showing the entry and exit 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 member of a sublibrary, CALL.PLITRC, that contains the following Debug Tool commands:

```
DCL LVLSTR CHARACTER ( 50 ) ;
DCL LVL    FIXED BINARY ( 15 ) ;
LVL = 0 ;
AT ENTRY *
  DO ;
    LVLSTR = ' ' ;
    LVL = LVL + 1 ;
    SUBSTR ( LVLSTR, LVL, 1 ) = '>' ;
    SUBSTR ( LVLSTR, LVL + 1, 8 ) = %CU ;
    LIST UNTITLED ( LVLSTR ) ;
    GO ;
  END ;
AT EXIT *
  DO ;
    SUBSTR ( LVLSTR, LVL, 1 ) = '<' ;
    SUBSTR ( LVLSTR, LVL + 1, 8 ) = %CU ;
    LIST UNTITLED ( LVLSTR ) ;
    LVL = LVL - 1 ;
    GO ;
  END ;
```

You can use this file as the source of commands to Debug Tool by entering the following command:

```
USE (CALL.PLITRC)
```

If, after executing the USE file, you run the following program sequence:

```
PLIMAIN:
  ...
  CALL 'PLISUB';
  ...
PLISUB:
  ...
  CALL 'PLISUB2';
  ...
PLISUB2:
  ...
  CALL 'PLISUB2';
  END;
```

the following trace, or something similar, is displayed in in the Log window:

```
'>PLIMAIN                                        '
' >PLISUB                                        '
'  >PLISUB2                                      '
'  <PLISUB2                                      '
' <PLISUB                                        '
'<PLIMAIN                                        '
```

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

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.

```
    2 FIELD1(2) CHAR(8);
    2 FIELD2 CHAR(8);
     CTR = 3;         /* an invalid index value is set */
     FIELD-1(CTR) = 'TOO MUCH';
```

Find the address of FIELD2 with the command

```
DESCRIBE ATTRIBUTES FIELD2
```

Suppose the result is X'00521D42'. To set a breakpoint which 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 will halt if the value in this storage changes.

Setting a Breakpoint to Halt before Calling an Undefined Program

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 debugging session without raising a condition.

Chapter 5. Using the Debug Tool Interfaces

This chapter describes how you interface to Debug Tool and helps you understand and navigate through the windows provided. It covers customizing your display, choosing Debug Tool settings to adjust your debugging environment, entering commands on the command line, and getting help.

Customizing Debug Tool for Your Environment

Debug Tool provides its own full-screen support to supply you with a full-screen, interactive session for debugging your application. You can configure the screen into as many as three windows. Using all three windows, you can simultaneously view:

- Source window - displays the source file (for C) or the source listing (for COBOL and PL/I)
- Monitor window - displays the changing values of variables
- Log window - displays a log of your interactions with Debug Tool

Using the Debug Tool Session Panel

After you invoke your program, execution of Debug Tool begins, depending on the specified suboptions of the run-time TEST option. If Debug Tool gains control (for example, because of `__ctest()` or CALL CEETEST statements, or because TEST(ALL) is specified) and prompts you for input, the Debug Tool session panel appears. This panel is similar to the one shown in Figure 19 on page 86, and you use it to accomplish most of your tasks and communications with Debug Tool.

The Debug Tool session panel contains a header field with information about the program you are debugging, and can also contain up to three windows: a Monitor window, a Log window, and a Source window, in any combination. The following sections explain what these windows are for, how to use them, how to move from one to the other (navigate), and how to arrange their appearance and content.

Session Panel Header Fields

Figure 17 and Figure 18 on page 85 show two examples. The first is a header for a COBOL program, and the second is a header for C program. Descriptions of the specific areas follow the figures.

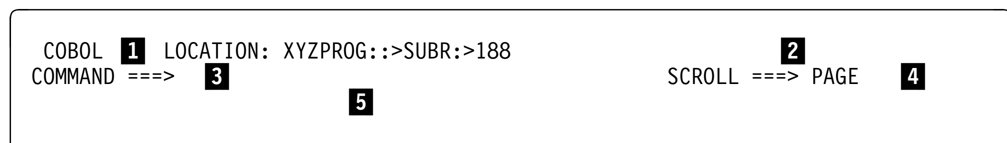


Figure 17. Session Panel Header Fields for a COBOL Program

© Copyright IBM Corp. 1995, 1996

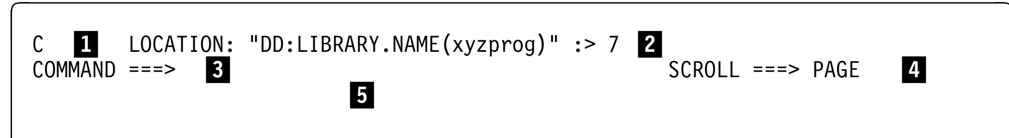


Figure 18. Session Panel Header Fields for a C Program

1 **C**, **COBOL**, or **PL/I**:

The name of the current programming language. This is not necessarily the programming language of what appears in the source window.

2 **LOCATION**:

The program unit name and statement where execution is suspended. (It is usually in the form of *compilation unit:>nnnnnn*.) In the first example, execution in XYZPROG is suspended at line 188 of subroutine SUBR. In the second example, execution in member XYZPROG of sublibrary LIBRARY.NAME is suspended at line 7.

3 **COMMAND**:

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

4 **SCROLL**:

The number of lines or columns you want to scroll when you enter a scroll command without an amount specified. You can set the display on or off using the SET SCROLL DISPLAY command. Modify the scroll amount with 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. The scrolling commands can be used to scroll by increments of n lines, half a page, a full page, to the top or bottom of the data, to the limit of the data, to the left or right by specified amounts, or to the position of the cursor.

5 **Message areas**:

Display information and error messages in the space immediately below the command line.

Session Panel Windows

Figure 19 on page 86 shows the entire Debug Tool session panel, including the session panel header and the default configuration for the Source window, the Log window, and the Monitor window.

```
COBOL    LOCATION: IBTUFS4 :> 100.1
Command ===>                                                   Scroll ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 1 OF 3
******************************* TOP OF MONITOR ********************************
0001  1 77 IBTUFS4:>VARBL2    21
0002  2 77 IBTUFS4:>VARBL1    11      3
0003  3 77 IBTUFS4:>X    1
****************************** BOTTOM OF MONITOR ******************************
SOURCE: IBTUFS4 --1----+----2----+----3----+----4----+----5---- LINE: 98 OF 118
      98              ADD 1 TO VARBL1                                         .
      99  1           ADD 1 TO VARBL2                                         .
     100              CALL "SUBPRO1" USING BY CONTENT PARAM1                  .
     101              ADD 1 TO X                                              .
     102            END-PERFORM.                                              .
LOG 0----+----1----+----2----+----3----+----4----+----5----+---- LINE: 13 OF 19
0013 The command element MONITOR is invalid.
0014  MONITOR
0015    LIST VARBL2 ;
0016  MONITOR               2
0017    LIST VARBL1 ;
0018  MONITOR
0019    LIST X ;
```

Figure 19. Session Panel with Opened Monitor, Source, and Log Windows

Source Window 1

The Source window displays the source file or source listing. The Source window has four parts: the header area, the prefix area, the source display area, and the suffix area.

Header Area: The header area identifies the window and shows the compilation unit name. It also shows the current position of the source or listing.

Prefix Area: The prefix area appears in the leftmost eight columns of the Source window, and 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. For more on prefix commands, see “Using Prefix Commands” on page 90.

Source Display Area: The source display area shows the source code (for a C program), or the source listing (for a COBOL or PL/I program) for the currently qualified program unit. The source display is usually shown with the current statement highlighted (if the statement can be found).

Suffix Area: The suffix area is a narrow, variable-width column at the right of the screen. Debug Tool uses the suffix area for displaying frequency counts. It is only as wide as the largest count it must display.

The suffix area is optional, and you can turn it on with SET SUFFIX ON, while SET SUFFIX OFF removes it from the screen. You can also set it on or off with the SOURCE LISTING SUFFIX field in the Profile Settings Panel. More information on the Profile Settings Panel is included in “Customizing Settings” on page 99.

Monitor Window 3

Use the Monitor window to continuously display output from the MONITOR LIST, MONITOR QUERY, and MONITOR DESCRIBE commands. This window is first opened when you enter a monitor 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, and added to the monitor list. You can specify the monitor number; however, you must either replace an existing monitor number, thus redefining the referenced command, or use the next sequential number.

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 so it indicates 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.

Log Window 2

This window records and displays your interactions with Debug Tool. All commands that are valid in full-screen mode, and their responses, are automatically appended to the Log window except the full-screen commands PANEL, FIND, CURSOR, RETRIEVE, SCROLL, WINDOW, and IMMEDIATE, and the QUERY and SHOW prefix commands. 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. The default for the number of log lines retained for display is 1000 lines, but you can specify a different value with SET LOG KEEP `n`, where `n` is the number of lines you want to keep.

The maximum length is determined by the amount of storage available.

Using the Session Log File to Maintain a Record of Your Session

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 during a later session by specifying it as your primary commands file. This is a convenient method of reproducing debugging 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 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, and so on)

The default for the Debug Tool session log file is the system output device, SYSLST. (Refer to “The Log File” on page 30 for details on the session log file.)

For COBOL Only

If you want to subsequently use the session log file as a commands file, make the LRECL less than or equal to 72. Debug Tool ignores everything after column 72 for file input during a COBOL debugging session.

End of For COBOL Only

For CICS Only

Under CICS, SET LOG OFF is the default. To start the log, you must issue:

```
SET LOG ON FILE log.file;
```

End of For CICS Only

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 created.

Entering the command, SET LOG ON FILE xxx.log, will cause the log for the Debug Tool session to be appended to an existing file `xxx.log` or, if it does not exist, file `xxx.log` will be created.

If a log file was not created for your session, you can create one with the SET LOG command by entering:

```
SET LOG ON FILE (log.file);
```

This creates the log file `log.file` in the first sublibrary specified in the SOURCE search chain.

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 both full-screen mode and batch mode.

Entering Commands in a Debug Tool Session

You can enter a command or modify what is on the session panel in seven areas. These areas are indicated in Figure 20.

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

Figure 20. Session Panel with Command Areas Indicated

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.

For information about retrieving and modifying commands, see "Retrieving Lines from the Session Log and Source Windows" on page 92.

Command Sequencing

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. Compile unit name area
3. Scroll area
4. Window id area
5. Source/Log window
6. Command line

Using the Command Line

You can type any Debug Tool command in this field. If you need to enter a command that is longer than the field, you can indicate to Debug Tool that you want to continue the command by typing the Debug Tool command continuation character, the SBCS hyphen (-), at then end of your input. When the current programming language is 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 current programming language is C and your command was begun with a left brace ({) that has not been matched by a right brace (}).

If you 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.

Using Prefix Commands

Certain commands, known as *prefix commands*, can be typed over the prefix area in the Source window, and then processed by pressing Enter. These commands—AT, CLEAR, DISABLE, ENABLE, QUERY, and SHOW—pertain only to the line or lines of code at 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 before 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 Cursor Commands

Certain commands are sensitive to the position of the cursor. These commands, called *cursor-sensitive* commands, include all those that contain the keyword CURSOR (such as AT CURSOR, DESCRIBE CURSOR, LIST CURSOR, SCROLL...CURSOR, and 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.

Using Program Function (PF) Keys to Enter Commands

The cursor commands, as well as other full-screen tasks, can be issued more quickly by assigning PF keys to them than by typing them on the command line. 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.

Defining PF Keys

To define your PF keys, use the SET PFKEY command. For example, to define PF key 8 as SCROLL DOWN PAGE, issue:

```
SET PF8 'Down' = SCROLL DOWN PAGE ;
```

The string set apart by single quotations ('Down' in this instance) 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.

Abbreviating Commands

When you issue Debug Tool commands, you can abbreviate most keywords. Usually, you need enter only enough characters in a command keyword to uniquely specify it. You can even use an abbreviation that is the same as a variable in your program. Debug Tool gives precedence to abbreviations of commands over variable names.

However, you cannot truncate keywords reserved for other programming languages, or special case keywords such as CALL, COMMENT, END, FILE (in the SET INTERCEPT and SET LOG commands), GOTO, INPUT, LISTINGS (in the SET DEFAULT LISTINGS command), or USE.

PROCEDURE can be abbreviated only as PROC.

Retrieving Commands

You can retrieve the last command you entered by entering RETRIEVE; on the command line. The retrieved command is displayed on the command line, and can be issued by pressing Enter again. You can modify retrieved commands before you reissue them.

Repeated executions of the RETRIEVE command scrolls through previous commands in reverse order; that is, the last command entered is displayed first, then the command prior to that, then the command prior to that, for as long as you continue to press Enter.

To make the use of this command more convenient, assign RETRIEVE to a PF key using the SET PFKEY command. Press the RETRIEVE PF key to display the retrieved command on the command line. If a retrieved command is too long to fit in the command line, only its last line is displayed.

Retrieving Lines from the Session Log and Source Windows

You can retrieve lines from your session 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.

Creating EQUATES and Using String Substitution

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)"`.

- `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. For a complete list of these operands, see “SET EQUATE” on page 309.

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 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. For more information on these settings, see "Customizing Settings" on page 99.

Moving the Cursor

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. (For details on how to assign commands to PF keys, see "Using Program Function (PF) Keys to Enter Commands" on page 91.) 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.

Scrolling the Windows

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.

Positioning Lines at the Top of Windows

If you want to display a selected line at the top of a window, issue the SCROLL TO command. Use the statement numbers shown in the window prefix areas. Type the line number on the command line, move the cursor to the selected window, and press the SCROLL TO PF key. Or, type 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.

Searching for a Character or Character String

To search the Log, Source, or Monitor window for a given character or graphic string while you are engaged in a full-screen Debug Tool session, issue the FIND command. The following list provides you with examples of using the FIND command:

- If you want to search your listing for the variable `var1`, enter the following command, place the cursor in the Source window, and press Enter:

  ```
  FIND "var1";
  ```

  - Alternatively, you can enter:

    ```
    FIND "var1" SOURCE;
    ```

  - If `var1` is in the Log or Monitor window, enter:

    ```
    FIND "var1" LOG
    ```

    or

    ```
    FIND "var1" MONITOR
    ```

  If `var1` is found but not visible in the specified window, the window scrolls forward vertically and horizontally in order to display it. When Debug Tool locates and displays it, `var1` is highlighted and the cursor is placed at the variable. The search wraps around so if the window is positioned past the last occurrence, the first occurrence in the window is found.

- If you want to search the specified window for the next occurrence of `var1`, just enter the following command, place the cursor in the window you are searching, and press Enter:

  ```
  FIND
  ```

  You do not need to provide the variable name, because Debug Tool remembers the string you last searched for. Again, the specified window is scrolled forward, `var1` is highlighted, and the cursor points to the variable.

  You can think of the FIND command as a cursor-sensitive command, and you can conveniently issue it if you first assign it to a PF key.

- Assume you have assigned FIND to a PF key and want to search for the variable `var1` in the Source window. All you need to do is type `"var1"` or `'var1'` on the command line, move the cursor to the Source window, and press the FIND PF key. The window scrolls forward and displays the occurrence of `var1`.

If you do not place the cursor in a selected window or specify a window on the command line, the FIND command searches the window specified with the SET DEFAULT WINDOW command or the **Default window** entry in your Profile Settings Panel.

If you are searching for strings with trigraphs in them, the trigraphs or their equivalents can be used as input, and Debug Tool matches them to trigraphs or their equivalents.

Customizing Your 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. For information on the settings included in that panel, see "Customizing Settings" on page 99.

Changing Session Panel Window Layout

You can change window placements on the session panel during your session by using the PANEL LAYOUT command. The PANEL keyword is optional. When you issue this command, you are presented with a configuration panel as shown in Figure 21. The configuration panel displays six possible ways you can change your Debug Tool session panel window placements.

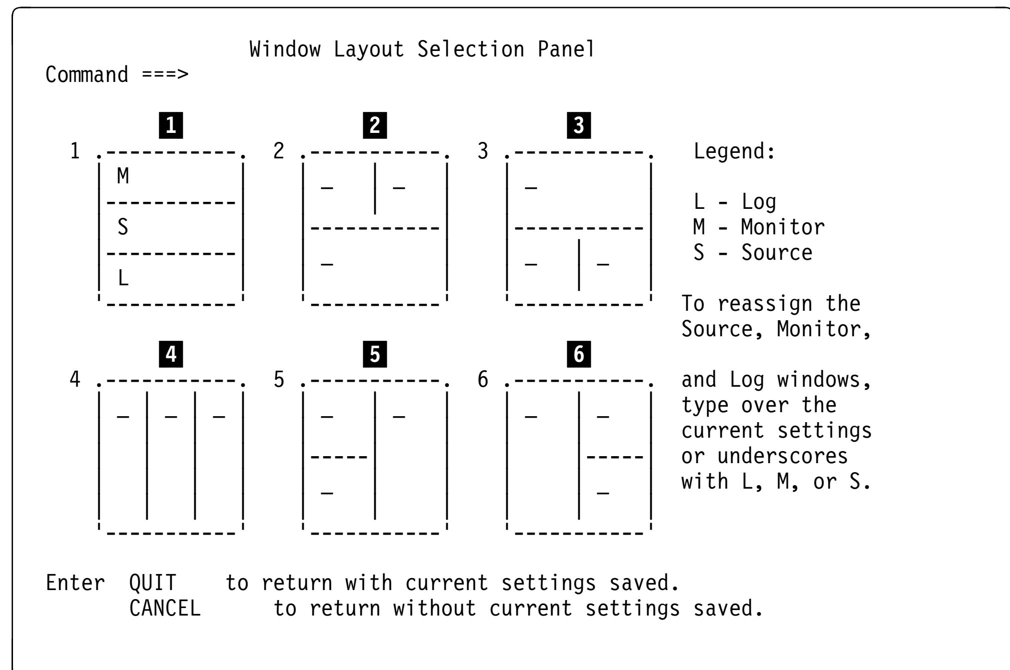


Figure 21. Window Layout Selection Panel. The default configuration is shown as option 1.

Initially, the session panel looks like the default window configuration shown as **1** in Figure 21.

To change the window placements for your Debug Tool session, select a configuration example and move the cursor to your selected example. Type the desired window letters—L for Log, M for Monitor, and S for Source—over the underscores; then press Enter. In Figure 21, configuration **1** is the chosen configuration.

Note: If you choose a different configuration than that displayed you should not blank out the window letters in that configuration, just enter the letters in your new configuration.

You can select only one configuration at a time. Also, only one of each type of window can be visible at a time on your session panel. For example, you cannot assign the session log to be visible in more than one window.

After you reassign the window placements, issue the END command or press the END PF key to save the changes and return to the session display.

Opening and Closing Session Panel Windows

To open and close any of the windows on the Debug Tool session panel, issue the WINDOW OPEN and WINDOW CLOSE commands. For example, if you want to open the Monitor window, enter:

```
WINDOW OPEN MONITOR;
```

You can also issue the WINDOW CLOSE command by typing it on the command line, placing the cursor in the desired window (or by specifying the name of the window as an operand of the WINDOW CLOSE command), and pressing Enter. When you close one or two specified windows, the remaining windows occupy the full area of the screen. For example, to close the Source window from the command line, enter:

```
WINDOW CLOSE SOURCE;
```

The WINDOW CLOSE command can be assigned to a PF key. For details, see “Using Program Function (PF) Keys to Enter Commands” on page 91.

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 fills in the available space indicated by your selected configuration.

If at anytime during your session you open a window and the contents assigned to it are not available, the window is empty.

Sizing Session Panel Windows

In addition to configuring, opening, and closing the Debug Tool session panel windows, you can control the relative sizes of these windows by using the WINDOW SIZE command. You can either explicitly specify the number of rows or columns you want the window to contain (as appropriate for the window configuration) or use the WINDOW SIZE command with the cursor. The WINDOW keyword is optional. For example, to explicitly change the Source window from 10 rows deep to 12 deep, enter:

```
WINDOW SIZE 12 SOURCE
```

By positioning the cursor at the point on the screen where you want the window boundary and issuing the WINDOW SIZE command, you can adjust the relative sizes of windows with great flexibility. For instance, assume only the Source and Log windows are open and you want to enlarge the size of the Source window before you step through your program. Enter:

```
WINDOW SIZE SOURCE;
```

on the command line, move the cursor to the desired row, and press Enter. The boundary of the Source window moves to the cursor position.

WINDOW SIZE can be assigned to a PF key. For details, see “Using Program Function (PF) Keys to Enter Commands” on page 91.

During your session, if you modify the relative sizes of your windows using the cursor you can restore them to the default sizes by entering:

```
PANEL LAYOUT RESET;
```

Intersecting Windows

To change the size of any intersecting windows (in configurations **2**, **3**, **5**, and **6**, shown in Figure 21) type:

```
WINDOW SIZE;
```

on the command line, move the cursor to where you want the windows to intersect, and press Enter. The windows are resized according to the new point of intersection.

Horizontal Windows

To change the size of the upper two horizontal windows (in configuration **1**, shown in Figure 21), use the WINDOW SIZE command as above, either moving the cursor below the window intersection to increase the top window and decrease the middle one, or moving it above the intersection to increase the middle window and decrease the top one.

Similarly, you can change the size of the middle and bottom windows.

Vertical Windows

To change the size of the left and middle windows (in configuration **4**, shown in Figure 21), use the WINDOW SIZE command, either moving the cursor to the left of the window intersection to increase the middle window and decrease the left one, or moving it to the right of the intersection to increase the left window and decrease the middle one.

Zooming a Window

The WINDOW ZOOM command specifies that the indicated window be expanded to fill the screen. This function allows you to view more data, reducing the amount of scrolling needed.

If the specified window is already zoomed and you specify ZOOM again, the currently defined window configuration is restored.

Customizing 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 in Figure 22 on page 98 appears.

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.

```
                      Color Selection Panel
Command ===>
                        Color    Highlight   Intensity
Title  : field headers  TURQ     NONE        HIGH
         output fields  GREEN    NONE        LOW      Valid Color:
Monitor: contents       TURQ     REVERSE     LOW        White Yellow Blue
         line numbers   TURQ     REVERSE     LOW        Turq Green Pink Red
Source : listing area   WHITE    REVERSE     LOW
         prefix area    TURQ     REVERSE     LOW      Valid Intensity:
         suffix area    YELLOW   REVERSE     LOW        High Low
         current line   RED      REVERSE     HIGH
         breakpoints    GREEN    NONE        LOW      Valid Highlight:
Log    : program output TURQ     NONE        HIGH       None Reverse
         test input     YELLOW   NONE        LOW        Underline Blink
         test output    GREEN    NONE        HIGH
         line numbers   BLUE     REVERSE     HIGH     Color and Highlight
 Command line           WHITE    NONE        HIGH     are valid only with
 Window headers         GREEN    REVERSE     HIGH     color terminals.
 Tofeof delimiter       BLUE     REVERSE     HIGH
 Search target          RED      NONE        HIGH
Enter  END/QUIT    to return with current settings saved.
       CANCEL      to return without current settings saved.
```

Figure 22. Color Selection Panel with Default Settings

Initially, the session panel areas and fields have the default color and attribute values shown in Figure 22.

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.

Color and attribute settings remain in effect for the entire debug session.

To preserve any changes you make to the default field colors, Debug Tool saves color and attribute settings for use during subsequent sessions in a profile settings file in the sublibrary member *userid*.DTSAFE. If Debug Tool finds member *userid*.DTSAFE when it initializes the debugging session, it saves revised color and attribute settings in the same member in the same sublibrary (that is, it overwrites the existing member). If it has to create the member, it writes it to the first sublibrary in the SOURCE search chain (see “Profile Settings File” on page 29 for further information). If this member is not available for your next session, Debug Tool begins the next debugging session with the values shown in Figure 22.

Customizing 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 in Figure 23 with the IBM-supplied initial settings. You can change the settings by either typing over them with the desired values, or by issuing the appropriate SET command from the command line or from within a commands file.

```
                         Profile Settings Panel
Command ===>

                                Current Setting
                                ---------------
 Change Test Granularity        STATEMENT      (All,Blk,Line,Path,Stmt)
 DBCS characters                NO             (Yes or No)
 Default Listing sublibrary
 Default scroll amount          PAGE           (Page,Half,Max,Csr,Data,int)
 Default window                 SOURCE         (Log,Monitor,Source)
 Execute commands               YES            (Yes or No)
 History                        YES            (Yes or No)
 History size                   100            (nonnegative integer)
 Logging                        YES            (Yes or No)
 Pace of visual trace           2              (steps per second)
 Refresh screen                 NO             (Yes or No)
 Rewrite interval               50             (number of output lines)
 Session log size               1000           (number of retained lines)
 Show log line numbers          YES            (Yes or No)
 Show message ID numbers        NO             (Yes or No)
 Show monitor line numbers      YES            (Yes or No)
 Show scroll field              YES            (Yes or No)
 Show source/listing suffix     YES            (Yes or No)
 Show warning messages          YES            (Yes or No)
 Test level                     ALL            (All,Error,None)
Enter  QUIT    to return with current settings saved.
       CANCEL      to return without current settings saved.
```

Figure 23. Profile Settings Panel with Default Settings

A list of the profile parameters, their descriptions, and the equivalent SET commands follows.

Change Test Granularity
: Specifies the granularity of testing for AT CHANGE. Equivalent to SET CHANGE.

DBCS characters
: Controls whether the *shift-in* and *shift-out* characters are recognized. Equivalent to SET DBCS.

Default Listing sublibrary
: Specifies the sublibrary where Debug Tool looks for the source/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 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 Figure 23.

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.

To preserve any changes you make to the session panel settings, Debug Tool saves these settings for use during subsequent sessions in a profile settings file in the sublibrary member *userid*.DTSAFE. If Debug Tool finds member *userid*.DTSAFE when it initializes the debugging session, it saves revised session panel settings in the same member in the same sublibrary (that is, it overwrites the existing member). If it has to create the member, it writes it to the first sublibrary in the SOURCE search chain (see “Profile Settings File” on page 29 for further information).

All PANEL settings are saved, except the setting for the Listing Panel and the following settings:

COUNTRY
FREQUENCY
INTERCEPT
LOG
NATIONAL LANGUAGE
PROGRAMMING LANGUAGE
QUALIFY
SOURCE
TEST

If this sublibrary member is not available for your session, Debug Tool begins the next debugging session with the values shown in Figure 23.

Settings remain in effect for the entire debug session.

Getting Help During Your Session

Command syntax help is available with Debug Tool. If you are uncertain as to the proper syntax or exact keywords required by a command, enter the command, followed by a question mark, on the command line:

```
STEP ?
```

The following information is displayed in your Log window:

```
The partially parsed command is:
STEP
The next word can be one of:
*           OVER
;           RETURN
unsigned positive integer
INTO
```

Chapter 6. Multiple Enclaves

This chapter discusses debugging multiple enclaves, and using Debug Tool features with multiple enclaves.

The following topics are covered in this chapter:

Invoking Debug Tool within an enclave
Using the Source window
Retaining a log file of your Debug Tool session
Processing commands from a commands file
Using breakpoints within multiple enclaves
Ending a Debug Tool session
Using Debug Tool commands within multiple enclaves

In LE/VSE terminology, an enclave is collection of routines, one of which is designated as the main routine. An enclave is equivalent to a COBOL run unit, a C program consisting of a main C function and its subroutines, or a PL/I main procedure and all its subprocedures. An enclave exists within an LE/VSE process. In LE/VSE, one enclave (the parent enclave) can create a second enclave (the child enclave) in the same process by using the following methods:

- Under CICS, the EXEC CICS LINK and EXEC CICS XCTL commands
- In the batch environment, the C `system()` function

Note, however, that a child enclave is only created if the target routine of these commands is written in an LE/VSE-conforming HLL or LE/VSE-conforming assembler.

For more information about enclaves, processes, and multiple (or nested) enclaves, see *LE/VSE Programming Guide*.

Invoking Debug Tool within an Enclave

There is a single Debug Tool session across all enclaves in a process. Once Debug Tool is activated by any enclave in the process, 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 (see “SET TEST” on page 323).

If Debug Tool is first activated in a child 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 invoked for a new enclave.

© Copyright IBM Corp. 1995, 1996

Using the Source Window

A particular enclave's Source and Listing windows are hidden when that enclave invokes another enclave. You cannot open a Source or Listing window for a compile unit unless that compile unit is in the current enclave.

Retaining a Log File of your Debug Tool Session

Ensure that your log file is correctly allocated. See “Using the Session Log File to Maintain a Record of Your Session” on page 87.

Processing Commands from a Commands File

A commands file continues to process its series of commands regardless of what level of enclave is entered.

Using Breakpoints within Multiple Enclaves

When any process is initialized, a termination breakpoint is automatically defined for the process, and each enclave in the process. Unless you clear or disable this breakpoint, it will be triggered when each enclave finishes execution. During run time of a termination breakpoint, GO and STEP are valid commands that cause your program to continue running the next enclave in the series.

Ending a Debug Tool Session

In a single enclave, QUIT closes Debug Tool. In a nested enclave, however, QUIT causes Debug Tool to signal a severity 3 condition corresponding to LE/VSE message CEE2529S, and the enclave is terminated. LE/VSE then terminates the entire process with abend 4094 reason code 40 (X'28').

There is one case where Debug Tool raises LE/VSE severity 3 condition and all enclaves in the process do not terminate: Under CICS, when the assembler user exit for the application (or the default assembler user exit) does not perform an EXEC CICS ABEND for unhandled severity 3 conditions. In these cases, the application continues to run, but Debug Tool becomes inactive.

For CICS Only

Under CICS, an abend appears on the application terminal. For LE/VSE it is 4038. An abend at termination of a nested enclave is normal and should be expected.

End of For CICS Only

Using Debug Tool Commands within Multiple Enclaves

Some Debug Tool commands and variables have a specific scope for enclaves and processes. Table 3 on page 105 summarizes the behavior of specific Debug Tool commands and variables when you are debugging an application that consists of multiple enclaves. For syntax and a full description of each of the Debug Tool commands, see Chapter 13, “Debug Tool Commands” on page 206.

Table 3 (Page 1 of 2). Scope of Debug Tool Commands and Variables across Multiple Enclaves

Debug Tool Command	Affects Current Enclave Only	Affects Entire Debug Tool Session	Comments
%CAAADDRESS	X		
AT GLOBAL		X	
AT TERMINATION		X	
CLEAR AT	X	X	In addition to clearing breakpoints set in the current enclave, CLEAR AT can clear global breakpoints.
CLEAR DECLARE		X	
CLEAR VARIABLES		X	
Declarations		X	Session variables are cleared at the termination of the process in which they were declared.
DISABLE	X	X	In addition to disabling breakpoints set in the current enclave, DISABLE can disable global breakpoints.
ENABLE	X	X	In addition to enabling breakpoints set in the current enclave, ENABLE can enable global breakpoints.
LIST AT	X	X	In addition to listing breakpoints set in the current enclave, LIST AT can list global breakpoints.
LIST CALLS	X		Lists the call chain for the current active thread in the current active enclave.
LIST EXPRESSION	X		You can only list variables in the currently active thread.
LIST LAST		X	
LIST NAMES CUS		X	Applies to compile unit names.
LIST NAMES TEST		X	Applies to Debug Tool session variable names.
MONITOR GLOBAL		X	Applies to Global monitors.
PROCEDURE		X	
SET COUNTRY[1]	X		This setting affects both your application and Debug Tool. At the beginning of an enclave, the settings are those provided by LE/VSE or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.
SET EQUATE[1]		X	
SET INTERCEPT[1]		X	For C, intercepted streams or files cannot be part of any C I/O redirection during the execution of a nested enclave. For example, if `stdout` is intercepted in program A, program A cannot then redirect `stdout` to `stderr` when it does a `system()` call to program B. Also, not supported for PL/I.
SET NATIONAL LANGUAGE[1]	X		This setting affects both your application and Debug Tool. At the beginning of an enclave, the settings are those provided by LE/VSE or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.

Table 3 (Page 2 of 2). Scope of Debug Tool Commands and Variables across Multiple Enclaves

Debug Tool Command	Affects Current Enclave Only	Affects Entire Debug Tool Session	Comments
SET PROGRAMMING LANGUAGE[1]	X		Applies only to programming languages in which compile units known in the current enclave are written (a language is "known" the first time it is entered in the application flow).
SET QUALIFY[1]	X		Can only be issued for phases, compile units, and blocks that are known in the current enclave.
SET TEST[1]		X	
TRIGGER condition[2]	X		Applies to triggered conditions.[2] Conditions can be either an LE/VSE symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.
TRIGGER AT	X	X	In addition to triggering breakpoints set in the current enclave, TRIGGER AT can trigger global breakpoints.

Note:

1. SET commands other than those listed in this table affect the entire Debug Tool session.
2. If no active condition handler exists for the specified condition, the default condition handler can cause the program to end prematurely.

Chapter 7. Using Debug Tool in Different Modes and Environments

This chapter describes:

- Using Debug Tool in interactive (full-screen) and batch mode
- Programming considerations for CICS, DL/I, and SQL/DS in interactive or batch mode
- Examples of build steps for CICS, DL/I, and SQL/DS applications, including preprocessing, compile, link, run, and debug
- Suggestions on how to invoke Debug Tool in CICS, DL/I, and SQL/DS environments
- Using Debug Tool CICS Interactive Run-Time Facility (DTCN)

Using Debug Tool in Batch Mode

Debug Tool can run in batch mode, creating a non-interactive session.

In batch mode Debug Tool receives its input from either the preferences file, primary commands file, a USE file, or the command string specified in the run-time TEST option, and writes its normal output to a log file.

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 a CICS non-terminal application.

When Debug Tool is reading commands from a specified file and no more commands are available in that file, it forces a GO command until the end of the program is reached.

When debugging in batch mode, use a QUIT command to end your session.

Debugging CICS Programs

Before you can debug your programs under CICS, make sure your Systems Programmer has made the appropriate changes to your CICS partition to support Debug Tool (see *Debug Tool for VSE/ESA Installation and Customization Guide*). You also need to ensure that your program is translated by the CICS translator **prior** to compilation. The program listing (for COBOL and PL/I) or the program source file (for C) must be retained in a permanent file for Debug Tool to read when you debug your program.

© Copyright IBM Corp. 1995, 1996

Note: For C, it is the input to the compiler (that is, the output from the CICS translator) that needs to be retained. To enhance performance when using Debug Tool, use a large blocksize when saving these to sequential access files.

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:**

  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.

  Note: In this mode it is recommended to set screen refreshing on by issuing the Debug Tool command SET REFRESH ON.

- **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.

  When the application issues an EXEC CICS RECEIVE, the Debug Tool terminal will wait until you respond to the application terminal.

  Note: 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.

- **Interactive Non-terminal Mode:**

  Use this mode if you are debugging a transaction which does not normally have a terminal associated with it. Debug Tool screens are displayed on a 3270 session that you name.

- **Non-interactive Non-terminal Mode:**

  In this mode, Debug Tool does not have a terminal associated with it. It receives its commands from a command file and writes its results to a log file. This mode is useful if you want Debug Tool to debug a program automatically.

Mechanisms for Invoking Debug Tool under CICS

There are several different mechanisms available to invoke Debug Tool under CICS:

1. Debug Tool CICS Interactive Utility (DTCN)

   DTCN is a full-screen CICS transaction that allows you to specify your debugging requirements dynamically. It also gives you the opportunity to provide any LE/VSE run-time options you might want to override during your debugging session.

   DTCN is the recommended mechanism for invoking Debug Tool for most debug sessions and can support all the modes outlined above.

2. LE/VSE CEEUOPT module link-edited into your application, containing an appropriate TEST option.

   This CEEUOPT mechanism tells LE/VSE to invoke Debug Tool every time the application is run. It can be useful during initial testing of new code when you will want to run Debug Tool frequently.

   You need to relink the application, removing the CEEUOPT module when you have finished debugging the application.

3. A run-time directive within the application, such as `#pragma runopts(test)` for C, the PLIXOPT string or `CALL PLITEST` for PL/I, or `CALL CEETEST`.

   These directives can be useful when you need to run multiple debug sessions for a piece of code which is deep inside a multiple enclave or multiple CU application. The application will run without Debug Tool until it encounters the directive, at which time Debug Tool is invoked 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.

Preparing and Using DTCN to Invoke Debug Tool under CICS

In order to use the DTCN utility to invoke Debug Tool, link-edit the DTCN customized LE/VSE user exit into the CICS program you want to debug. The JCL which link-edits your application should include the library which contains EQADCCXT in the OBJ search chain. Your link-edit options should include the following statement:

```
INCLUDE EQADCCXT
```

In order to use the DTCN utility, run the DTCN transaction at a 3270 session. If DTCN is not available, the CICS partition most likely has not been set up to run Debug Tool, and you should consult your Systems Programmer.

After DTCN is started, the panel below is provided. The fields are designed to capture the information needed for Debug Tool to start a debugging session with your application. The data provided by the user is then stored in a Debug Tool run-time start-up profile repository.

DTCN Screen

Start DTCN by entering the transaction identifier DTCN. The following screen appears:

```
DTCN               DEBUG TOOL CICS Interactive Runtime Facility       DBDCCICS

Item                Choice              Possible choices

Terminal Id     ==> L084                Application Terminal Id
Transaction Id  ==>                     Any valid Trans Id

Session Parm
 DT/VSE Term Id ==>                     MFI - DT Term Id(dual terminal mode)

Test Option     ==> Test                Test/Notest
Test Level      ==> All                 All/Error/None
Command File    ==>
Prompt Level    ==> Prompt
Preference File ==> *

Any other valid Language Environment Options
==>

EQA2007E SHOW FAILED - PROFILE DOES NOT EXIST



PF1=HELP 2=GHELP 3=EXIT  4=ADD 5=REPLACE 6=DELETE 7=SHOW 8=NEXT  10=CLOSE DTCN
```

Figure 24. Initial DTCN screen with no user profile.

The sections that follow provide a detailed description for each area of the initial DTCN screen shown above.

Header Area

```
DTCN               DEBUG TOOL CICS Interactive Runtime Facility       DBDCCICS

Item                Choice              Possible choices
```

Figure 25. Header Area on the DTCN Screen

The Header area contains:

- Identifier of the transaction - DTCN.
- Application Id of the CICS partition in which the transaction is running - DBDCCICS in the case above.
- Column's description.

Input Area

The following sections show the Input Area on the DTCN screen.

Area 1

```
Terminal Id     ==> L084              Application Terminal Id
Transaction Id  ==>                   Any valid Trans Id
```

Area 2

```
Session Parm
 DT/VSE Term Id ==>                   MFI - DT Term Id(dual terminal mode)
```

Area 3

```
Test Option     ==> Test              Test/Notest
Test Level      ==> All               All/Error/None
Command File    ==>
Prompt Level    ==> Prompt
Preference File ==> *
```

Area 4

```
Any other valid Language Environment Options
```

Figure 26. Sections of the Input Area on the DTCN Screen

The input area is used to display and enter the data for the debugging profile. The input area of the DTCN panel is divided into four sections:

Area 1: This section contains the Terminal Id and Transaction Id which when concatenated together are the key used by DTCN to process debugging profiles.

Terminal Id CICS terminal identifier where you want to run your application in debugging mode.

Note: The default value of this field is the terminal identifier from where DTCN is being run.

Transaction Id CICS transaction identifier you want to debug.

Area 2: This section contains the Terminal Id of the terminal used to display Debug Tool panels when Debug Tool is run in dual terminal mode.

DT/VSE Term Id
The CICS terminal identifier where you want Debug Tool to initialize during a dual-terminal mode debugging session.

Area 3: For more detailed information on the following fields, see “Run-Time TEST Option Syntax” on page 33.

TEST Option *TEST/NOTEST* specifies the conditions under which Debug Tool assumes control during the initialization of the application.

Test Level *ALL/ERROR/NONE* specifies what conditions need to be met for Debug Tool to gain control.

Command File A valid file-id specifying the primary command file for this run.

Note: Enclosing the name of the file in single or double quotes is not allowed.

Prompt Level *PROMPT/NOPROMPT/*/;* specifies whether Debug Tool is invoked at LE/VSE initialization.

Preference File
A valid file-id specifying the preference file to be used.

Note: Enclosing the name of the file in single or double quotes is not allowed.

Area 4

Other LE/VSE options
This area is provided to allow the user to specify additional run-time options needed to debug their application.

Message Line

```
==>

EQA2007E SHOW FAILED - PROFILE DOES NOT EXIST
```

Figure 27. Message Line on the DTCN Screen

DTCN displays the messages at the bottom of the screen. When DTCN is started, it attempts to show the profile for the terminal on which it has been invoked. If a profile has not previously been established for the given terminal, message EQA2007E is displayed. For more information about profiles, see "Profile Repository" on page 113.

For a successful Show or Next command, DTCN displays the full LE/VSE options stored in the Profile Repository. Options longer than 79 characters are truncated but their contents are properly displayed in the input field section. Check Appendix F, "Debug Tool Messages" on page 355 for an explanation and programmers response.

DTCN PF Key Definitions

```
PF1=HELP 2=GHELP 3=EXIT  4=ADD 5=REPLACE 6=DELETE 7=SHOW 8=NEXT  10=CLOSE DTCN
```

Figure 28. PF Key Area on the DTCN Screen

The PF keys are described as follows:

PF1 Help Context sensitive help. Provides detailed help for the entry fields when positioning the cursor on the field and pressing ? (PF1).

PF2 GHelp General help for DTCN

PF3 Exit Exits DTCN

PF4 Add Adds a new profile to the profile repository (no replace)

PF5 Replace Replaces an existing profile in the repository (no add)

PF6 Delete Removes the current profile from the repository

PF7 Show Retrieves the specified profile from the repository

PF8 Next Retrieves the next profile from the repository

PF10 Close Deletes the profile repository with all stored debugging profiles

Profile Repository

DTCN allows you to build Debug Tool run-time start-up profiles that are used when you run your application. These profiles are stored in a Debug Tool Profile Repository that contains all the entries for the CICS partition you are working in. The fields on the DTCN screen defined earlier make up the data for one entry. The DTCN screen allows you to add, replace, or delete an entry from the Profile Repository.

DTCN uses a key that uniquely identifies each entry in the Profile Repository. The key is made up by concatenating the application Terminal Id and Transaction Id fields from the DTCN screen. Entries are sorted and stored in the repository using this key.

Note: The Debug Tool Profile Repository is stored in working storage, and is only held by Debug Tool for the duration of the current CICS job. The Debug Tool Profile Repository is not maintained from one invocation of CICS to the next.

The scope of the debugging profile depends on the contents of the Terminal Id and Transaction Id. There are three levels of the profiles:

Terminal Id (blank)
Debugging profile applies to any invocation of a given Transaction Id partition wide (generic transaction debugging profile). This is used to interactively debug CICS non-terminal transactions or for troubleshooting programs run with:

```
(TEST(ERROR,,NOPROMPT,*))
```

Terminal Id, Transaction Id (blank)
Any enabled transaction run on a specified terminal uses this stored debugging profile (generic terminal debugging profile).

Terminal Id, Transaction Id
Debugging profile applies only to enabled corresponding transaction running on corresponding terminal (specific for terminal and transaction).

When the application is run, during the initialization of the first enclave, the profile presented back to LE/VSE (and hence Debug Tool) is that with the narrowest scope, that is, a profile for a specific terminal and transaction takes precedence over one for a specific terminal (generic transaction) which takes precedence over the generic transaction profile.

Modifying Other Options

You can dynamically change any other LE/VSE options defined in your CICS installation as overrideable except the STACK option. For additional information about LE/VSE options, see the various LE/VSE publications or contact your CICS system programmer.

DTCN Data Entry Errors

DTCN performs data verification on the data 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.

Once you have entered your debug requirements and saved them, you can start the application. Debug Tool will run according to the options you have specified.

After you have finished debugging your program, use DTCN again to turn off your debugging profile. While your program is being tested, you do not need to remove EQADCCXT from the phase; in fact, it's a good idea to leave it there for the next time you want to invoke Debug Tool. You should, however, remove EQADCCXT from the phase before you migrate your program to production.

Preparing and Using CEEUOPT to Invoke Debug Tool under CICS

You can include a user run-time options module, CEEUOPT, to define TEST run-time options. For instructions on how to create the CEEUOPT run-time options module using the CEEXOPT macro, follow steps 1 to 4 on page 116.

Debug Tool runs in the mode defined in the run-time TEST 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 invoke Debug Tool, run the application. Don't forget to remove the CEEUOPT containing your run-time TEST option when you have finished debugging your program.

For information on the run-time TEST option, see “Using the Run-Time TEST Option” on page 33.

Preparing and Using Compile-Time Directives To Invoke Debug Tool under CICS

In addition to using the user run-time options module, CEEUOPT, to specify the TEST run-time option, you can specify the option in the source of your C or PL/I program. For more information, see “Specifying Run-Time TEST Option with #pragma runopts in C” on page 41 or “Specifying Run-Time TEST Option with PLIXOPT string in PL/I” on page 41.

Note: For COBOL, you will need to include CEEUOPT to specify run-time TEST options as there is no other way to pass run-time options in COBOL.

To request the program itself to invoke Debug Tool, you can code calls to the LE/VSE callable service CEETEST or, for PL/I programs, the PL/I built-in subroutine PLITEST. For more information, see “Invoking Debug Tool with CEETEST” on page 43 or “Invoking Debug Tool with PLITEST” on page 51. Whenever Debug Tool is invoked by a call to CEETEST or PLITEST, it is invoked in the mode specified by the suboptions in the TEST or NOTEST run-time option.

Restrictions When Debugging Under CICS

The following restrictions apply when debugging programs with the Debug Tool in a CICS environment:

- The `__ctest()` function with CICS does nothing.
- The CIND transaction is a special Debug Tool service transaction, and is not intended for activation by direct terminal input. If CIND is invoked via terminal entry, it will return to the caller (no function is performed).
- Applications which issue EXEC CICS POST cannot be debugged in Dual Terminal mode.

- The JCL for your CICS startup job might not include all the JCL for the Debug Tool files you refer to during your debug session. Therefore, you should refer to all Debug Tool files, including source files, the log file, USE files, and preferences file, by their full names. Sublibrary member names should include the name of the library and sublibrary, as well as the member name and type. SAM ESDS files and sequential disk files should be referred to by their full file-ids.

  Important: Debug Tool uses VSE operating system services, not CICS services, to perform I/O on the files it uses. Therefore, you should only use Debug Tool in a CICS development environment. Using Debug Tool in a CICS production environment might impact the performance of your CICS system.

- CICS does not support an attention interrupt from the keyboard.
- The log file 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.)

Debugging DL/I Programs

When you are planning to use Debug Tool to debug your DL/I programs, certain steps need to be taken. These steps are described in detail below.

Programming Considerations

When using Debug Tool to debug your DL/I application programs:

- Do not call CEETEST or `__ctest`. Instead, use the run-time options module CEEUOPT (shown in Figure 29 on page 117), to specify the TEST parameter.
- For COBOL, the following rules apply:
  - Do not use the ENTRY 'anyname' USING .... statement. Instead, code the USING clause on the PROCEDURE DIVISION statement.
  - If your COBOL program calls other COBOL programs that you also want to debug, do not use ENTRY statements in the called programs. The program name must be the same as its entry point name. Debug Tool cannot locate the program listing when entry points are used.

Program Preparation

Program preparation steps for DL/I include compile and link activities.

Compile Requirements

Your program must be compiled with the compile-time TEST option. Use the default options to gain maximum debugging facilities.

Important: Ensure that your source (if you are working with C language) or compiler listing (if you are working with COBOL or PL/I) is stored in a permanent file that is available to Debug Tool.

Link Requirements

With a DL/I program, the run-time TEST option cannot be specified at program start, and must be coded and assembled in a user-defined run-time options module. When you link your program include a run-time options module, CEEUOPT, in your link-edit by doing the following:

1. Find the user run-time options source program CEEUOPT.A in the LE/VSE installation sublibrary; the default installation sublibrary for this file is PRD2.SCEEBASE.
2. Change the NOTEST parameter into a default TEST parameter:

   ```
   old:  NOTEST=(ALL,*,PROMPT,''),
   new:  TEST=(,*,;,*),
   ```

   Note: To invoke a full-screen debugging session the MFI suboption must be specified, for example:

   ```
   new:  TEST=(,*,;,MFI%vtam_lu:),
   ```

   where `vtam_lu` is the VTAM terminal you wish Debug Tool to connect to.

3. Assemble the CEEUOPT program and keep the object code.
4. Link-edit the CEEUOPT object code with any program to invoke Debug Tool.

Figure 29 on page 117 shows the modified assembler program, CEEUOPT.

```
*/******************************************************************/
*/*                                                                */
*/*    LICENSED MATERIALS - PROPERTY OF IBM                        */
*/*                                                                */
*/*    5686-094  (C) COPYRIGHT IBM CORP. 1991,  1996               */
*/*    ALL RIGHTS RESERVED.                                        */
*/*                                                                */
*/*                                                                */
*/*     US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR */
*/*     DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM */
*/*     CORP.                                                      */
*/*                                                                */
*/*                                                                */
*/******************************************************************/
CEEUOPT  CSECT
CEEUOPT  AMODE ANY
CEEUOPT  RMODE ANY
         CEEXOPT ABPERC=(NONE),                                        X
               ABTERMENC=(RETCODE),                                    X
               AIXBLD=(OFF),                                           X
               ALL31=(OFF),                                            X
               ANYHEAP=(16K,8K,ANYWHERE,FREE),                         X
               BELOWHEAP=(8K,4K,FREE),                                 X
               CBLOPTS=(ON),                                           X
               CBLPSHPOP=(ON),                                         X
               CHECK=(ON),                                             X
               COUNTRY=(US),                                           X
               DEBUG=(ON),                                             X
               DEPTHCONDLMT=(10),                                      X
               ENVAR=(''),                                             X
               ERRCOUNT=(20),                                          X
               HEAP=(32K,32K,ANYWHERE,KEEP,8K,4K),                     X
               LIBSTACK=(8K,4K,FREE),                                  X
               MSGFILE=(SYSLST),                                       X
               MSGQ=(15),                                              X
               NATLANG=(UEN),                                          X
               TEST=(,*,;,*),                                          X
               RPTOPTS=(OFF),                                          X
               RPTSTG=(OFF),                                           X
               RTEREUS=(OFF),                                          X
               STACK=(128K,128K,BELOW,KEEP),                           X
               STORAGE=(NONE,NONE,NONE,8K),                            X
               TERMTHDACT=(TRACE),                                     X
               TRACE=(OFF,4K,DUMP,LE=0),                               X
               TRAP=(ON),                                              X
               UPSI=(00000000),                                        X
               XUFLOW=(AUTO)
         DC    C'5686-094 (C) COPYRIGHT IBM CORP. 1991, 1996. '
         DC    C'LICENSED MATERIALS - PROPERTY OF IBM'
         END
```

Figure 29. Run-Time Options Module CEEUOPT

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 invoked.

For C and PL/I Only

It is possible to use a run-time directive within the program, such `#pragma runopts` for C and PLIXOPT for PL/I, to invoke Debug Tool. See "Specifying Run-Time TEST Option with #pragma runopts in C" on page 41 and "Specifying Run-Time TEST Option with PLIXOPT string in PL/I" on page 41 for more information.

End of For C and PL/I Only

Using Debug Tool with DL/I Programs

DL/I programs can be debugged in either batch or interactive mode. When using batch mode you must know the exact Debug Tool commands you want to have executed during the test, and include them in the primary commands file. In interactive mode, the debugging commands can be entered interactively.

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. This might involve including them as part of a library search chain (specified using the LIBDEF JCL statement).
2. Specify your debug commands in the primary commands file.
3. Invoke the program with standard job control statements.

Interactive Mode

In this mode, you can decide at debug time on the debugging commands to be issued during the test. To debug your program in this mode, follow these steps:

1. Make sure the Debug Tool modules are available. This might involve including them as part of a library search chain (specified using the LIBDEF JCL statement).
2. Invoke the program with standard job control statements.

After your program has been initiated, debug your program by issuing the required Debug Tool commands.

Note: If your source is not displayed in the Source window when you launch Debug Tool, check that the source or listing file name corresponds to the member name or filename of your source or listing file. For more information see "PANEL Command (Full-Screen Mode)" on page 289, "SET DEFAULT LISTINGS" on page 307, and "SET SOURCE" on page 322.

Debugging SQL/DS Programs

When you are planning to use Debug Tool to debug your SQL/DS programs, certain steps need to be taken. These steps are described in detail below.

Programming Considerations

There are no special coding techniques for any SQL/DS programs you might want to debug using Debug Tool. For details on how to code your program to access a SQL/DS database, see the relevant HLL Programming Guide and *SQL/Data System Application Programming for VSE*.

To communicate with SQL/DS, you should:

- Delimit SQL statements with EXEC SQL and END-EXEC statements
- Declare SQLCA in working storage
- Declare program variables
- Code the appropriate SQL statements
- Test the SQL/DS return codes

Program Preparation

Program preparation includes the SQL/DS preprocessor, the compiler, the LE/VSE prelinker, and the linkage editor.

Preprocessor Requirements

Before your program can be compiled, the SQL statements must be prepared using the SQL/DS preprocessor. For details about the preprocessor, see *SQL/Data System Application Programming for VSE*. No special preparations are needed in the preprocessor step to use Debug Tool.

When debugging a program containing SQL, keep the following in mind:

- The SQL preprocessor replaces all the SQL statements in the program with language statements. 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 language code inserted by the SQL preprocessor invokes 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.

Compile Requirements

The output from the preprocessor must be used as input to the compiler. To debug your program with Debug Tool, use the compile-time TEST option. A description of TEST is found in one of the following sections:

"Compiling a C Program with the Compile-Time TEST Option" on page 12
"Compiling a COBOL Program with the Compile-Time TEST Option" on page 16
"Compiling a PL/I Program with the Compile-Time TEST Option" on page 19

The suboptions of the compile-time TEST option control the production of such debugging aids as dictionary tables and program hooks that Debug Tool needs in order to debug your program. The choices you make when compiling your program can affect the amount of Debug Tool function available during your debugging

session. When a program is under development, you should compile it with TEST(ALL) to get the full capability of Debug Tool.

Important: Ensure that the source file produced by the SQL/DS preprocessor (if you are working with C language) or compiler listing file (if you are working with COBOL or PL/I) is stored in a permanent file that is available to Debug Tool.

Link Requirements

The output from the compiler must then be linked into your program phase sublibrary. To define TEST run-time options you need to perform one of the following actions:

1. If you are running your SQL/DS program in multiple user mode you can pass parameters directly to the program. This means you can pass TEST run-time options to the program to invoke Debug Tool without needing to specify them in a CEEUOPT object module.
2. If you are running in single user mode you must link-edit a CEEUOPT object with your program to specify the appropriate TEST options. You can include a user run-time options module, CEEUOPT, to define TEST run-time options. For instructions on how to create the CEEUOPT run-time options module using the CEEXOPT macro, follow steps 1 to 4 on page 116.
3. It is possible to use a run-time directive within the program, such as `#pragma runopts(test)` for C, or PLIXOPT for PL/I, to invoke Debug Tool. Refer to "Using Alternative Debug Tool Invocation Methods" on page 43 for more details.

For C Only

When running Debug Tool, links or calls to C programs linked with AMODE 24 are not supported if the C library phases CEEEV003 and EDCZ24 are loaded in the 31-bit SVA or above the line in a partition that spans the 16MB line. To run any 24-bit C program under Debug Tool, CEEEV003 and EDCZ24 must be loaded below the line.

End of For C Only

Using Debug Tool with SQL/DS Programs

SQL/DS programs can be debugged in either batch or full-screen mode. When using batch mode you must know the exact Debug Tool commands you want to have executed during the test, and include them in the primary commands file. In full-screen, the debugging commands can be entered interactively.

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. This might involve including them as part of a sublibrary search chain (specified using the LIBDEF JCL statement).
2. Specify your debug commands in the primary commands file.

3. Invoke the program with standard job control statements.

Interactive Mode

In this mode, you can decide at debug time which debugging commands to be issued during the test. To debug your program in this mode, follow these steps:

1. Make sure the Debug Tool modules are available. This might involve including them as part of a sublibrary search chain (specified using the LIBDEF JCL statement).
2. Invoke the program with standard job control statements.

After your program has been initiated, debug your program by issuing the required Debug Tool commands.

Note: If your source is not displayed in the Source window when you launch Debug Tool, check that the source or listing file name corresponds to the member name or filename of your source or listing file. For more information see “PANEL Command (Full-Screen Mode)” on page 289, “SET DEFAULT LISTINGS” on page 307, and “SET SOURCE” on page 322.

See also the note (marked **Important**) in “Compile Requirements” on page 119.

The program listing that Debug Tool displays and uses for the debugging session is the output from the compile step, and thus includes all the SQL/DS expansion code produced by the SQL/DS preprocessor.

Part 2. Language-Specific Information

© Copyright IBM Corp. 1995, 1996

Chapter 8. Debug Tool Support of Programming Languages

This chapter discusses the ways Debug Tool makes it possible for you to debug programs of different languages, structures, conventions, variables, and methods of evaluating expressions.

As part of the effort to support multiple high-level programming languages, Debug Tool has adapted its commands to the different HLLs, enabled you to use *interpretive subsets* of commands from the various HLLs, and mapped common attributes of data types across the languages. It does the following:

- Maps compatible attributes between HLL data types
- Evaluates HLL expressions
- Interprets HLL variables and constants

This chapter also describes the concept of interpretive command subsets, exceptions and conditions in Debug Tool, and Debug Tool's built-in functions.

A general rule to remember is that Debug Tool tries to let the language itself guide how Debug Tool works with it. Further information is available in the various HLL language reference manuals, listed in the bibliography.

Multiple Enclaves and Interlanguage Communication (ILC)

Debugging a multi-enclave ILC application with Debug Tool is supported. However, keep the following points in mind:

- The SET PROGRAMMING LANGUAGE command can be used to change the current programming language setting. However, the programming language setting is limited to the languages currently known to Debug Tool (that is, languages contained in the current phase).

- Command lists on monitors and breakpoints have an implied programming language setting, which is the language that was in effect at the time the monitor or breakpoint was established. This means that if you change the language setting, errors may result when the monitor is refreshed or the breakpoint is triggered.

Compatible Attributes Mapped Between HLL Data Types

Debug Tool allows you, while working in one language, to declare session variables you can continue to use after calling in a phase of a different language. See the Attribute Mapping table in "Language Compatible Attributes" on page 246 for more information on how session data attributes are mapped across programming languages. Attributes not shown in the table cannot be mapped to other programming languages.

Also remember that variables with incompatible attributes cannot be accessed from another programming language.

© Copyright IBM Corp. 1995, 1996

Debug Tool Evaluation of HLL Expressions

Whenever an expression is entered, Debug Tool will remember the programming language in effect at that time. When the command is run, the expression will be passed to the language run time that was in effect when the expression was entered, which may be different from the one in effect when the expression is run.

When you are entering an expression that will not be run immediately, it is recommended that all program variables be fully qualified. This will ensure that proper context information (such as phase, block, etc.) will be passed with the expression to the language run time when the statement is run. If this is not done, the context may not be the one you intended when you set the breakpoint, and the language run time may fail to evaluate the expression.

Debug Tool Interpretation of HLL Variables and Constants

Debug Tool also supports the use of HLL variables and constants, both as a part of evaluating portions of your test program and in declaring and using temporary variables.

Three general types of variables are supported by Debug Tool. These are:

- Program variables defined by the HLL compiler's symbol table
- Debug Tool variables denoted by the percent (**%**) sign
- Temporary, or session, variables declared for a given Debug Tool session and existing only for the session

Variables

Some variable references require language-specific evaluation, such as pointer referencing or subscript evaluation. Once again, 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 PL/I: dot (.) qualification, high-level to low-level
  COBOL: IN or OF keyword, low-level to high-level

- Subscripting

  C: `name [subscript1][subscript2]...`
  COBOL and PL/I: `name(subscript1,subscript2,...)`

Constants

You can use both string constants and numeric constants. Debug Tool accepts both types of constants in C, COBOL, and PL/I.

Debug Tool Variables (or Intrinsic Functions)

Debug Tool has reserved several variables to contain its own information. These variables are denoted by the percent sign (**%**) as a first character, to distinguish them from program variables, and can be accessed while testing programs in any supported HLL.

Table 4 on page 126 shows a list of Debug Tool variables and the languages with which they can be used. Following the table is a list of their definitions.

Table 4. Descriptions of Debug Tool Variables and Their Corresponding Languages

Debug Tool Variable	C	PL/I	COBOL	Description
%GPRn	X	X	X	Represents general-purpose registers.
%FPRn	X	X	X	Represents single-precision floating-point registers.
%LPRn	X	X	X	Represents double-precision floating-point registers.
%EPRn	X	X		Represents extended-precision floating-point registers.
%ADDRESS	X	X	X	Contains the address of the location where your program was interrupted.
%AMODE	X	X	X	Contains the current AMODE of the suspended program (either 24 or 31).
%BLOCK	X	X	X	Contains the name of the current block. **Note:** The block name provided may not be unique within a compile unit.
%CAAADDRESS	X	X	X	Contains the address of the CAA control block associated with the suspended program.
%CONDITION	X	X	X	Contains the name (or number) of the condition identification when Debug Tool is entered because of an AT OCCURRENCE.
%COUNTRY	X	X	X	Contains the current country code.
%CU	X	X	X	Contains the name of the primary entry point of the current program. Equivalent to %PROGRAM.
%EPA	X	X	X	Contains the address of the primary entry point in the currently interrupted program.
%HARDWARE	X	X	X	Identifies the type of hardware where the application is running.
%LINE	X	X	X	Contains the current line number. Equivalent to %STATEMENT.
%LOAD	X	X	X	Contains the name of the phase of the current program, or an asterisk (*).
%NLANGUAGE	X	X	X	Contains the national language currently in use.
%PATHCODE	X	X	X	Contains an integer value identifying the type of change occurring when the program flow changes.
%PLANGUAGE	X	X	X	Contains the current programming language.
%PROGRAM	X	X	X	Contains the name of the primary entry point of the current compile unit. Equivalent to %CU.
%RC	X	X	X	Contains a return code whenever a Debug Tool command ends.
%RUNMODE	X	X	X	Contains a string identifying the presentation mode of Debug Tool.
%STATEMENT	X	X	X	Contains the current statement number. Equivalent to %LINE.
%SUBSYSTEM	X	X	X	Contains the name of the underlying subsystem, if any, where the program is executing.
%SYSTEM	X	X	X	Contains the name of the operating system supporting the program.

You can use all Debug Tool variables in expressions. Additionally, the first four variables, representing the various types of registers, can be used as the targets of assignments.

Note: Use caution when assigning new values to registers. Important program information can be lost.

Detailed descriptions of the Debug Tool variables follow.

Modifiable Debug Tool Variables

%GPR0, %GPR1,...,%GPR15
Represent general-purpose registers at the point of interruption in a program.

%FPR0, %FPR2, %FPR4, %FPR6
Represent single-precision floating-point registers.

%LPR0, %LPR2, %LPR4, %LPR6
Represent the double-precision floating-point registers. They are similar to the single-precision floating-point registers (%FPRs).

%EPR0, %EPR4
Represent the extended-precision floating-point registers.

Nonmodifiable Debug Tool Variables

%ADDRESS
Contains the address of the location where the program has been interrupted.

%AMODE
Contains the current AMODE of the suspended program. Possible values are 24 or 31.

%BLOCK
Contains the name of the current block.

%CAAADDRESS
Contains the address of the CAA control block associated with the suspended program.

%CONDITION
Contains the name (or number) of the condition identification when Debug Tool is entered due to an AT OCCURRENCE.

%COUNTRY
Contains the current country code.

%CU
Contains the name of the primary entry point of the current compile unit.

%CU is equivalent to %PROGRAM.

%EPA
Contains the address of the primary entry point of the currently interrupted program.

%HARDWARE
Identifies the type of hardware where the application program is running. A possible value is: 370/ESA.

%LINE
Contains the current line number. This value can include a period, since the current line can be a statement other than the first statement on a source line.

If the program is at the entry or exit of a block, %LINE contains ENTRY or EXIT, respectively.

If the line number cannot be determined (for example, a run-time line number does not exist or the address where the program is interrupted is not in the program), %LINE contains an asterisk (*).

%LINE is equivalent to %STATEMENT.

%LOAD
Contains an asterisk (*) unless the current program is part of a fetched or called phase. If the current program is part of a fetched or called phase, %LOAD contains the name of that phase.

%NLANGUAGE
Indicates the national language currently in use. Possible values are:

ENGLISH
UENGLISH
JAPANESE

%PATHCODE
Contains an integer value that identifies the kind of change occurring when the path of program execution has reached a point of discontinuity and the path condition is raised.

The possible values vary according to the language of your program. See:

"Using Debug Tool Variables in C" on page 140 for your C program

"Using Debug Tool Variables in COBOL" on page 167 for your COBOL program, or

"Using Debug Tool Variables in PL/I" on page 181 for your PL/I program.

%PLANGUAGE
Indicates the programming language currently in use.

%PROGRAM
Contains the name of the primary entry point of the current program.

%PROGRAM is equivalent to %CU.

%RC
Contains a return code whenever a Debug Tool command ends.

%RC initially has a value of zero unless the log file cannot be opened, in which case it has a value of -1.

The %RC return code is a Debug Tool variable. It is not related to the return code that can be found in Register 15.

%RUNMODE
Contains a string identifying the presentation mode of Debug Tool. Possible values are:

SCREEN
BATCH

%STATEMENT
Contains the current statement number. This value can include a period, since the current statement can be one other than the first statement in a source line.

If the program is at the entry or exit of a block, %STATEMENT contains ENTRY or EXIT, respectively.

If the statement number cannot be determined (for example, a run-time statement number does not exist or the address where the program is interrupted is not in the program), %STATEMENT contains an asterisk (*).

%STATEMENT is equivalent to %LINE.

%SUBSYSTEM
Contains the name of the underlying subsystem, if any, where the program is executing. Possible values are:

CICS
NONE

%SYSTEM
Contains the name of the operating system supporting the program. The only possible value is: VSE.

Interpretive Subsets

To allow you to use familiar commands while in a debugging 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 phase in a different language, the current programming language is automatically updated.

The following types of Debug Tool commands have the same syntax (or a subset of it) as the corresponding statements (if defined) in each supported programming language:

Assignment These commands allow you to assign a value to a variable or reference.

Conditional These commands evaluate an expression and control the flow of execution of Debug Tool commands according to the resulting value.

Declarations These commands allow you to declare temporary variables.

Looping These commands allow you to program an iterative or logical loop as a Debug Tool command.

Multiway These commands allow you to program multiway logic in the Debug Tool command language.

In addition, Debug Tool supports special kinds of commands for some languages.

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 be able 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 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).

Qualification

Qualification is a method you can use to specify to what procedure or phase a particular variable belongs. You do this by prefacing the variable with the block, compile unit, and phase (or as many of these labels as are necessary), separating each label with a colon (or double colon following the phase specification) and a greater-than sign (:>), as follows:

```
PHASE_NAME::>CU_NAME:>BLOCK_NAME:>object
```

This procedure, known as *explicit qualification*, lets Debug Tool know precisely where the variable is.

PHASE_NAME is the phase name. It is required only when the application consists of multiple phases and when you want to change the qualification to other than the current phase. PHASE_NAME can be the Debug Tool variable %LOAD.

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.

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.

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:

```
PHASE1::>PROC1:>variable
```

is valid.

```
PHASE1::>PROC1:>PROC1:>variable
```

is not valid.

End of PL/I only

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: %BLOCKnnn, 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.

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 operand

```
PHASE_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 PHASEX::>CUX:>BLOCKX. Also, the phase PHASEZ, containing the compile unit CUZ and the block BLOCKZ, is known to Debug Tool. The settings currently in effect are:

%LOAD = PHASEX
%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 phasez::>cuz:>blockz;
```

the following settings are in effect:

%LOAD = PHASEZ
%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 phase.

Debug Tool Handling of Conditions and Exceptions

To suspend program execution just before your application would terminate abnormally, start your application with the following 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 LE/VSE's condition handler, that you have already handled the condition by issuing a GO BYPASS command. Beware that some of the code that follows the instruction that raised the condition may be relying on data that was not properly stored or handled.

When debugging with Debug Tool, you have a choice of either instructing Debug Tool to handle program exceptions and conditions, or passing them on to your own exception handler. Programs also have access to LE/VSE services to deal with program exceptions and conditions.

Condition Handling in Debug Tool

You can use either or both of the following 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 debugging session, Debug Tool gains control at the occurrence of most conditions.

  Note: Debug Tool recognizes all LE/VSE conditions that are detected by the LE/VSE 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. For more information on OCCURRENCE breakpoints, see “AT OCCURRENCE” on page 225.

For a description of HLL conditions, see the corresponding language references and the *LE/VSE Programming Guide*.

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 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 PL/I).

What Happens 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.
- Control of your program's execution can be returned to the HLL exception handler, 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, bypassing any further processing of this exception either by the user program or the environment.

- PL/I allows *GO TO out of block;*, so execution control can be passed to some other point in the program.
- If no circumstances exist explicitly directing the assignment of control, your primary commands file or terminal is queried for another command.

If, after the execution of any defined breakpoint, control returns to your program with a GO command, the condition is raised again in the program (if possible and still applicable). If you use a GOTO to bypass the failing statement, you also bypass your program's error handling facilities.

Exception Handling within Expressions (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 as such 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 displayed.

However, you might want to pass an exception on to your program, perhaps to test an error recovery procedure. In this case, use SET WARNING OFF.

Requesting an Attention Interrupt During Interactive Sessions

During an interactive Debug Tool session you can request an attention interrupt, if necessary, by pressing the attention key on your keyboard. 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.

LE/VSE run-time option TRAP should be set to ON in order for attention interrupts that are recognized by the operating system to be also recognized by LE/VSE. The test level suboption of the run-time TEST option should *not* be set to NONE. See *LE/VSE Programming Guide*.

For CICS Only

An “attention interrupt” key is not supported in CICS.

End of For CICS Only

The attention key might not be marked ATTN on your keyboard. Often the PA1 key is used.

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 specify 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

Debug Tool's Built-in Functions

Debug Tool provides you with several built-in functions, available while debugging programs in all supported languages, which allow you to perform variable manipulations. These functions are distinguished by a percent sign (**%**) as the first character. Below is a brief description of each function, including its proper syntax.

For Use with C, COBOL, and PL/I

The following Debug Tool built-in functions are for use with C, COBOL, and PL/I.

%HEX

You can use %HEX with the LIST command to display the hexadecimal value of an operand.

```
►►──%HEX──(──reference──)──────────────────────────────────────────►◄
```

reference
A valid COBOL or PL/I reference, or C `lvalue`.

%STORAGE

You can use %STORAGE to reference storage by address and length. You can use this function only in conjunction with commands employing AT CHANGE.

```
►►──%STORAGE──(──address──┬──────────────┬──)──────────────────────►◄
                          └─,──length────┘
```

address
The starting address of storage to be monitored for changes. This must be an **0x** constant in C, an **H** constant in COBOL, or a **PX** constant in PL/I.

length
The number of bytes of storage to be monitored for changes. This must be a positive integer constant. The default value is 1.

For Use with C and PL/I

The following Debug Tool built-in functions are for use only with C and PL/I programs.

%INSTANCES

You can use %INSTANCES to provide the maximum value of %RECURSION (the most recent recursion number) for a given block.

```
►►──%INSTANCES──(──reference──)────────────────────────────────────►◄
```

reference
An automatic variable or a subroutine parameter. If necessary, you can use qualification to specify the variable.

%RECURSION

You can use %RECURSION to access an automatic variable or a parameter in a specific instance of a recursive procedure.

```
►►──%RECURSION──(──reference──,──expression──)──────────────────────►◄
```

reference
An automatic variable or a subroutine parameter. If necessary, you can use qualification to specify the variable.

expression
The recursion number of the variable or parameter. The oldest recursion is referenced by %RECURSION(var, 1) and the most recent by %RECURSION(var, %INSTANCES(var)).

For Use with PL/I

The following Debug Tool built-in function is for use only with PL/I programs.

%GENERATION

You can use %GENERATION to access a specific generation of a controlled variable in your program.

```
►►──%GENERATION──(──reference──,──expression──)─────────────────────►◄
```

reference
A controlled variable.

expression
The generation number (N) of a controlled variable (X), where:

`1 ≤ N ≤ ALLOCATION(X)`

The oldest instance of X is referenced by %GENERATION(X,1), and the most recent by %GENERATION(X,ALLOCATION(X)).

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 a list of conditions being monitored by the run-time environment. For example, if you enter DESCRIBE ENVIRONMENT while debugging a C program, you might get the following output:

```
Currently open files
    DD:SYSLST
    stdout
The following conditions are enabled:
    SIGFPE
    SIGILL
    SIGSEGV
    SIGTERM
    SIGINT
    SIGABRT
    SIGUSR1
    SIGUSR2
    SIGABND
```

Low-Level Debugging

Debug Tool is not an assembly-level debug tool, but 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 includes 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.

For example, here is a C program that you can run:

```
int dbl(int j)             /* line 1 */
{                          /* line 2 */
  return 2*j;              /* line 3 */
}                          /* line 4 */
int main(void)
{
  int i;
  i = 10;
  return dbl(i);
}
```

With the compile-time options `TEST(ALL),LIST`, your pseudo assembly listing will contain something like:

```
*  int dbl(int j)           /* line 1 */
           ST    r1,152(,r13)
*  {                        /* line 2 */
           EX    r0,HOOK..PGM-ENTRY
*    return 2*j;            /* line 3 */
           EX    r0,HOOK..STMT
           L     r15,152(,r13)
           L     r15,0(,r15)
           SLL   r15,1
           B     @5L2
           DC    A@5L2-ep)
           NOPR
 @5L1      DS    0D
*  }                        /* line 4 */
 @5L2      DS    0D
           EX    r0,HOOK..PGM-EXIT
```

Issue the command:

```
MONITOR LIST REGISTERS
```

to continuously monitor the registers. After a few steps, Debug Tool halts on line 1 and you have halted on the program entry hook seen above. Another STEP takes you to line 3 and you have halted on the statement hook. The next STEP takes you to line 4 and you have halted on the program exit hook. In accord with 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 ;
```

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.

Chapter 9. Using Debug Tool with C Programs

This chapter provides information on using C variables and expressions with Debug Tool. It covers the Debug Tool subset of C commands and reserved words, accessing program variables, declaring temporary variables (also known as session variables), displaying values of C variables, assigning values to C variables, and using Debug Tool variables.

It also covers expressions, including discussions of function calls, operators, and C unique statements; and qualification and multiple phases.

Debug Tool Commands

Debug Tool's command language is a subset of C statements 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 interpretive subset of C statements recognized by Debug Tool is shown in Table 16 on page 346. This subset of statements is valid only when the current programming language is C.

For specific usage notes concerning each command, see the appropriate section of Part 3, “Debug Tool Reference” on page 193.

In addition to the subset of C statements that you can use is a list of reserved keywords used and recognized by C that you cannot abbreviate, use as variable names, or use as any other type of identifier. This list is shown in Table 17 on page 346. These keywords are reserved only when the current programming language is C.

For explanations of command usage and keyword meaning, see *IBM C for VSE/ESA Language Reference*.

Using C Variables with Debug Tool

Debug Tool can process all program variables that are valid in C. It allows you to assign and display the values of variables during your session. It also allows you to declare temporary variables with the recognized C declarations to suit your testing needs.

Accessing 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.

See “Compiling a C Program with the Compile-Time TEST Option” on page 12 for more details.

© Copyright IBM Corp. 1995, 1996

Displaying Values of C Variables or Expressions

To display the values of variables or expressions, issue 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. See "LIST Command" on page 272 for more information.

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.

Declaring Temporary Variables

You might want to declare temporary variables, also known as session variables, for use during the course of your session. You cannot initialize temporary variables in declarations. However, you can use an assignment statement or function call to initialize a temporary 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. For more information see Table 11 on page 247.

To declare a floating-point variable called maximum, enter the following C declaration:

```
double maximum;
```

In Debug Tool you can only declare scalars, arrays of scalars, structures, unions, and pointers for all of these.

If you declare a temporary variable with the same name as a programming variable, the temporary 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.

For more on qualification, see “Using Qualification for C” on page 156. For more on declarations, see “Declarations (C)” on page 247.

Assigning Values to C Variables

To assign a value to a C variable, you use an assignment expression. See “Expression Command (C)” on page 261 for syntax information. 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.

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 operators except the tenary operator, as well as any other full C language assignments and function calls to user or C library functions. For more on function calls, see “Function Calls” on page 146.

Using Debug Tool Variables in C

Debug Tool variables, as shown in Table 5, provide information about your program that you can use during your session. These variables are distinguished by a percent character (**%**) as the first character in their names. To display the values of any of them during your session, use the LIST command.

Table 5 (Page 1 of 2). C Attributes for Debug Tool Variables

Debug Tool Variable	C Attributes	Description
%GPRn	signed int	Represents general-purpose registers.
%FPRn	float	Represents single-precision floating-point registers.
%LPRn	double	Represents double-precision floating-point registers.
%EPRn	long double	Represents extended-precision floating-point registers.
%ADDRESS	void *	Contains the address of the location where your program was interrupted.
%AMODE	signed short int	Contains the current AMODE of the suspended program (either 24 or 31).
%BLOCK	unsigned char[]	Contains the name of the current block.

Table 5 (Page 2 of 2). C Attributes for Debug Tool Variables

Debug Tool Variable	C Attributes	Description
%CAAADDRESS	void *	Contains the address of the CAA control block associated with the suspended program.
%CONDITION	unsigned char[]	Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or LE/VSE condition.
%COUNTRY	unsigned char[]	Contains the current country code.
%CU	unsigned char[]	Contains the name of the current compilation unit. Equivalent to %PROGRAM.
%EPA	void *	Contains the address of the primary entry point in the currently interrupted program.
%HARDWARE	unsigned char[]	Identifies the type of hardware where the application is running.
%LINE	unsigned char[]	Contains the current line number. Equivalent to %STATEMENT.
%LOAD	unsigned char[]	Contains the name of the phase of the current program.
%NLANGUAGE	unsigned char[]	Contains the national language currently being used.
%PATHCODE	signed short int	Contains an integer value identifying the type of change occurring when Debug Tool is entered because of a path breakpoint.
%PLANGUAGE	unsigned char[]	Contains the current programming language.
%PROGRAM	unsigned char[]	Contains the name of the primary entry point of the current program. Equivalent to %CU.
%RC	signed short int	Contains a return code whenever a Debug Tool command ends.
%RUNMODE	unsigned char[]	Contains a string identifying the presentation mode of Debug Tool.
%STATEMENT	unsigned char[]	Contains the current statement number. Equivalent to %LINE.
%SUBSYSTEM	unsigned char[]	Contains the name of the underlying subsystem, if any, where the program is executing.
%SYSTEM	unsigned char[]	Contains the name of the operating system supporting the program.

You can use all Debug Tool variables in expressions. Additionally, the variables representing general and floating-point registers are modifiable and can be used as the targets of assignment commands.

Note: When modifying register values, do not modify the base register.

Detailed descriptions of the Debug Tool variables follow.

%GPR0, %GPR1,...,%GPR15
Represent general-purpose registers at the point of interruption in a C program. You can use them in expressions:

```
list (%GPR5 + 10);
```

and as targets of assignments:

```
%GPR5 = name_table;
```

Notes:

- If you change a %GPRn register, the change is reflected when you resume program execution.
- Only %GPR12 can be used at external entry.
- Assigning new values to variables %GPR12 and %GPR13 does not result in an error, however, subsequent processing by Debug Tool will reset them to their previous values.
- If you change %GPR3 in an expression, the base register in the program can be lost.

%FPR0, %FPR2, %FPR4, %FPR6
Represent single-precision floating-point registers and are equivalent to `float` variables. You can use them in expressions:

```
x = %FPR4 / 6.3
```

and as targets of assignments:

```
%FPR0 = 3.14152
```

%LPR0, %LPR2, %LPR4, %LPR6
Represent the double-precision floating-point registers and are equivalent to `double` variables. Similar to the single-precision floating-point registers (%FPRs), you can use these registers in expressions and as targets of assignments.

%EPR0, %EPR4
Represent the extended-precision floating-point registers, and are equivalent to `long double` variables. Similar to the single-precision floating-point registers (%FPRs), you can use these registers in expressions and as targets of assignments.

%ADDRESS
Contains the address of the location where the program was interrupted.

%AMODE
Contains the current AMODE of the suspended program. Possible values are 24 or 31.

%BLOCK
Contains the name of the current block. To display the name of the current block, you can use the LIST command or issue:

```
DESCRIBE PROGRAM;
```

You can change or override the value of %BLOCK by using the SET QUALIFY command.

%CAAADDRESS
Contains the address of the CAA control block associated with the suspended program.

%CONDITION
Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or LE/VSE condition.

%COUNTRY
Contains the current country code.

%CU
Contains the name of the primary entry point of the current program.

You can change or override the value of %CU by using the QUALIFY command.

%CU is equivalent to %PROGRAM.

%EPA
Contains the address of the primary entry point of the currently interrupted program.

%HARDWARE
Identifies the type of hardware where the application program is running. A possible value is 370/ESA.

%LINE
Contains the current line (statement) number. This value can include a period since the current line can be a statement other than the first statement on a source line. For example, if `%LINE = 5.5`, the current statement is the fifth statement on the fifth source line.

If the program is at the entry or exit of a block, %LINE contains ENTRY or EXIT respectively.

If the line number cannot be determined (for example, if a run-time line number does not exist or the address where the program is interrupted is not in the program), %LINE contains an asterisk (*).

%LINE is equivalent to %STATEMENT.

%LOAD
Contains the name of the currently qualified phase and is used when an unqualified reference to a program or variable is made. If the currently qualified phase is the one initially loaded, %LOAD contains a single asterisk (*).

Whenever control is transferred to Debug Tool, %LOAD is set to the name of the currently executing phase (or to an asterisk in the case of the initial phase). You can change or override the value of %LOAD by using the SET QUALIFY command.

For phases to be recognized by Debug Tool, they must have been loaded by a language call and not through a direct operating system load command.

%NLANGUAGE
Indicates the national language currently in use. Its possible values include:

ENGLISH
UENGLISH
JAPANESE

%PATHCODE
Contains an integer value identifying the kind of path change taking place when Debug Tool is entered because of a path breakpoint. Possible values are:

-1 Debug Tool is not in control as the result of a path or attention situation.

0 An attention interrupt occurred.

1 A block has been entered.

2 A block is about to be exited.

3 Control has reached a user label.

4 Control is being transferred as a result of a function reference. The invoked routine's parameters, if any, have been prepared.

5 Control is returning from a function reference. Any return code contained in register 15 has not yet been stored.

6 Some logic contained by a conditional `do/while`, `for`, or `while` statement is about to be executed. This can be a single or Null statement and not a block statement.

7 The logic following an `if(...)` is about to be executed.

8 The logic following an `else` is about to be executed.

9 The logic following a `case` within a `switch` is about to be executed.

10 The logic following a default within a `switch` is about to be executed.

13 The logic following the end of a `switch`, `do`, `while`, `if(...)`, or `for` is about to be executed.

17 A `goto`, `break`, `continue`, or `return` is about to be executed.

Values in the range 3–17 can only be assigned to %PATHCODE if your program was compiled with an option supporting path hooks.

%PLANGUAGE
Indicates the programming language currently in use.

%PROGRAM
The name of the primary entry point of the current program.

You can change or override the value of %PROGRAM by using the QUALIFY command.

%PROGRAM is equivalent to %CU.

%RC
Contains a return code whenever a Debug Tool command ends.

%RC initially has a value of zero unless the log file cannot be opened, in which case it has a value of -1.

The %RC return code is a Debug Tool variable. It is not related to the return code that can be found in Register 15.

%RUNMODE
Contains a string identifying the presentation mode of Debug Tool. Possible values are:

SCREEN

BATCH

%STATEMENT
Contains the current statement number. This value can include a period since the current statement can be one other than the first statement in a source line.

If the program is at the entry or exit of a block, %STATEMENT contains ENTRY or EXIT, respectively.

If the statement number cannot be determined (for example, if a run-time statement number does not exist or the address where the program is interrupted is not in the program), %STATEMENT contains an asterisk (*).

%STATEMENT is equivalent to %LINE.

%SUBSYSTEM
Contains the name of the underlying subsystem, if any, where the program is executing. Possible values are:

CICS
NONE

%SYSTEM
Contains the name of the operating system supporting the program. The only possible value is: VSE.

C Expressions

Debug Tool allows evaluation of expressions in your test program. All expressions available in 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 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 *IBM C for VSE/ESA User's Guide* or *IBM C for VSE/ESA Language Reference*.

The semantics for C operators are the same as in a compiled C program. Operands can be a mixture of constants (`integer`, `floating-point`, `character`, `string`, and `enumeration`), C variables, Debug Tool variables, or session variables declared during a Debug Tool session. Language constants are specified as described in the *IBM C for VSE/ESA Language Reference*.

The Debug Tool command DESCRIBE ATTRIBUTES can be used to display the resultant type of an expression, without actually evaluating the expression.

The C language does not specify the order of evaluation for function call arguments. Consequently, it is possible for an expression to have a different execution sequence in compiled code than within Debug Tool. For example, if you enter the following in an interactive session:

```
int x;
int y;

x = y = 1;

printf ("%d %d %d%" x, y, x=y=0);
```

the results can differ from results produced by the same statements located in a C 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.

For more information about expressions and operators, refer to *IBM C for VSE/ESA Language Reference*.

The following examples show you various ways Debug Tool supports the use of expressions in your programs:

- Debug Tool assigns 12 to `a` (the result of the `printf()`) function call, as in:

  ```
  a = (1,2/3,a++,b++,printf("hello world\n"));
  ```

- Debug Tool supports structure and array referencing and pointer dereferencing, as in:

  ```
  league[num].team[1].player[1]++;
  league[num].team[1].total += 1;
  ++(*pleague);
  ```

- Simple and compound assignment is supported, as in:

  ```
  v.x = 3;
  a = b = c = d = 0;
  *(pointer++) -= 1;
  ```

- C language constants in expressions can be used, as in:

  ```
  pointer_to_c = "abcdef" + 0x2;
  *pointer_to_long = 3521L + 0x69a1L;
  float_val = 3e-11 + 6.6E-10;
  char_val = '7';
  ```

- The comma expression can be used, as in:

  ```
  intensity <<= 1, shade * increment, rotate(direction);
  alpha = (y>>3, omega % 4);
  ```

- Debug Tool performs all implicit and explicit C conversions when necessary. Conversion to long double is performed in:

  ```
  long_double_val = unsigned_short_val;
  long_double_val = (long double) 3;
  ```

Function Calls

You can perform calls to user and C library functions within Debug Tool.

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 compilation 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 details, see “Compiling a C Program with the Compile-Time TEST Option” on page 12.

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 compilation unit
- Debug Tool is able to locate a prototype for the function in another compilation unit, or the function itself was compiled with TEST(SYM).

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 compiler.

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 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.

For more information about `#pragma` linkage and the `extern` keyword, refer to *IBM C for VSE/ESA Language Reference*.

Using Debug Tool Functions with C

Debug Tool provides built-in functions for use during a debugging session. These functions allow greater access to your programming environment and greater control over your debugging session. Using these functions, you can reference storage, translate the values of operands to hexadecimal characters, or access a variable or parameter during a specific instance of a recursive procedure.

Using %HEX

When used with the LIST command, %HEX allows you to display the value of an operand as a hexadecimal character string. For example, if you want to examine the internal representation of the packed decimal variable `zvar1` whose external representation is 235, you can enter:

```
LIST %HEX(zvar1);
```

The hexadecimal value of `235C` is displayed in the Log window.

Using %STORAGE

%STORAGE allows you to reference storage by address and length. By using %STORAGE 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 (0x00022222, 8)
   LIST "Storage has changed at Hex address 22222"
```

Using %RECURSION

%RECURSION allows you to access an automatic variable or a parameter in a specific instance of a recursive function. When you use %RECURSION, remember that:

- If the expression has a value of 1, the oldest generation is referenced. The higher the value of the expression, the more recent the generation of the variable Debug Tool references.
- %RECURSION can be used like a Debug Tool variable.

Using %INSTANCES

%INSTANCES returns the maximum value of %RECURSION (that is, the most recent recursion number) for a given block. %INSTANCES can be used like a Debug Tool variable.

%INSTANCES and %RECURSION can be used together to determine the number of times a function is recursively called. They can also give you access to an automatic variable or parameter in a specific instance of a recursive procedure. Assume, for example, your program contains these statements:

```
int RecFn(unsigned int i) {
  if (i == 0) {
    __ctest("");
```

At this point, the `__ctest()` call gives control to Debug Tool, and you are prompted for commands. If you enter:

```
LIST %INSTANCES(i);
```

Your Log window displays the number of times RecFn() was interactively called.

If you enter:

```
%RECURSION(i, 1);
```

you receive the value of `'i'` at the first call of RecFn().

If necessary, you can use qualification to specify the parameter. For example, if the current point of execution is in `%block2`, and `%block3` is a recursive function containing the variable `x`, you can write an expression using `x` by qualifying the variable, as follows:

```
%RECURSION(main:>%block3:>x, %INSTANCES(main:>%block3:>x, y+3)) = 10;
```

For the proper syntax of the functions described above, see “Debug Tool's Built-in Functions” on page 134.

The following are examples of command sequences issued to Debug Tool using C semantics and library functions:

- The following example gets a line of input from `stdin` using the C library routine `gets`.

```
char line[100];
char *result;
result = gets(line);
```

- The following example removes a file and checks for an error, issuing a message if an error occurs.

```
int result;
result = remove("mayfile.dat");
if (result != 0)
  perror("could not delete file");
```

- Debug Tool performs the necessary conversions when a call to a library function is made. The cast operator can be used. In the following example, the integer 2 is converted to a double, which is the required argument type for `sqrt`.

```
double sqrtval;
sqrtval = sqrt(2);
```

- Nested function calls can be performed, as in:

```
printf("absolute value is %d\n", abs(-55));
```

- Qualification can be used in expressions. In the following example the function `check` (from the current compilation unit) is called with the variable `table` (from the function `main`) as a parameter. The return value is assigned to the variable `rc`.

```
rc = %CU:>check(main:>table);
```

  See the section on “Using Qualification for C” on page 156 for details.

- C library variables such as `errno` and `stdout` can be used, as in:

```
fprintf(stdout, "value of errno is %d\n", errno);
```

Debug Tool Evaluation of 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 expressions following the rules presented in the *IBM C for VSE/ESA Language Reference* publication. The result of an expression is equal to the result that would have been produced if the same expression had been part of your compiled program.

Implicit string concatenation is supported. For example, "abc" "def" is accepted for "abcdef" and treated identically. Concatenation of wide string literals to string literals is not accepted. For example, L"abc"L"def" is valid and equivalent to L"abcdef", but "abc" L"def" is not valid.

Expressions you use during your session are evaluated with the same sensitivity to enablement as are compiled expressions. Conditions that are enabled are the same ones that exist for program statements.

During a Debug Tool session, if the current setting for WARNING is ON, the occurrence in your C program of any one of the conditions listed below causes the

display of a diagnostic message. The messages themselves are displayed on your terminal, and are explained in Appendix F, "Debug Tool Messages" on page 355. The list below is for reference only.

- 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<—>2**31
- A real constant not having an exponent of 3 or fewer digits
- A float constant not larger than 5.397605346934027890866469914250249 6E-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.

Using SET INTERCEPT with C Programs

Several considerations must be kept in mind when using the INTERCEPT command to intercept files while you are debugging a C application.

For CICS Only

SET INTERCEPT is not supported for CICS.

End of For CICS Only

You can use the following names with the SET INTERCEPT command during a debugging 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 in the following manner:

- Intercepted input behaves as though the file is 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.

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 filename (DLBL) and file 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 LE/VSE 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 filename (DLBL), interception occurs only if the filename is specified on the INTERCEPT command. Intercepting the underlying file name does not cause interception of the stream.
- 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 follows:

1.
   a. **1>&2**:

      If `stderr` is the target of the interception command, `stdout` is also intercepted. If `stdout` is the target of the interception command, `stderr` is not intercepted. When INTERCEPT is set OFF for `stdout`, the stream is redirected to `stderr`.

   b. **2>&1**:

      If `stdout` is the target of the interception command, `stderr` is also intercepted. If `stderr` is the target of the interception command, `stdout` is not intercepted. When INTERCEPT is set OFF for `stderr`, the stream is redirected to `stdout` again.

2.
   a. **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**

   b. **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**

3.
   a. **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 interception command, the behavior follows rule 1b above.

   b. **1>&2 2>file.name**:

      `stdout` is redirected to `stderr`, and both are redirected to **file.name**. If you specify **file.name** on the interception command, both `stderr` and `stdout` are intercepted. If you specify `stdout` or `stderr` on the interception command, the behavior follows rule 1a above.

4. The same standard stream cannot be redirected twice on the command line. Interception is undefined if this is violated.

   a. **2>&1 2>file.name**:

      Behavior of `stderr` is undefined.

   b. **1>&2 1>file.name**:

      Behavior of `stdout` is undefined.

Objects and Scopes

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.

In ANSI C, the four kinds of scope are:

Block
File
Function
Function prototype

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.

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 variables and functions, the source file was compiled with TEST(SYM) and the object was referenced somewhere within the source.
- For 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 temporary 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 phases. Multiple phases are managed through the C library functions `fetch()`, and `release()`. For example, let's assume the program shown in Figure 30 on page 154 is compiled with TEST(SYM). When Debug Tool gains control, the file scope variables `length` and `table` are available for change, as in:

```
length = 60;
```

The block scope variables `i`, `j`, and `temp` are not visible in this scope and cannot be directly referenced from within Debug Tool at this time. You can list the line numbers in the current scope by entering:

```
LIST LINE NUMBERS;
```

Now let's assume the program shown in Figure 30 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.

```
#pragma runopts(EXECOPS)
#include <stdlib.h>

main()
{
      >>> Debug Tool is given <<<
      >>> control here.    <<<
  init();
  sort();
}

short length = 40;
static long *table;

init()
{
  table = malloc(sizeof(long)*length);
  ...
}

sort ()
{                                    /* Block sort */
  int i;
  for (i = 0; i < length-1; i++) {    /* 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;
    }
  }
}
```

Figure 30. Program Showing Support for Referencing Variables in Multiple Phases

Storage Classes

Debug Tool supports the change and reference of all objects declared with the following storage classes:

```
auto
register
static
extern
```

Temporary 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:

- Function_name:>`%BLOCKzzz` (short form)
- Function_name:>`%BLOCKxxx`:>`%BLOCKyyy` : ... :> `%BLOCKzzz` (long form).

`%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;
```

In the program shown in Figure 30 on page 154, the function `sort` has three blocks:

```
sort
%BLOCK2
%BLOCK3.
```

The following example sets a breakpoint on entry to the second block of `sort`:

```
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.

```
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.

```
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 a list of conditions being monitored by the run-time environment. For example, if you enter DESCRIBE ENVIRONMENT while debugging a C program, you might get the following output:

```
Currently open files
    DD:SYSLST
    stdout
The following conditions are enabled:
    SIGFPE
    SIGILL
    SIGSEGV
    SIGTERM
    SIGINT
    SIGABRT
    SIGUSR1
    SIGUSR2
    SIGABND
```

Using Qualification for 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 phases, 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 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*. Figure 31 on page 157, shows a block of code from a C program. 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`.

```
PHASE NAME:  MAINMOD
SOURCE FILE  NAME:  SORTMAIN C A

short length = 40;
main ()
{
  long *table;
  void (*pf)();

  table = malloc(sizeof(long)*length);
   ...
  pf = fetch("SORTMOD");
  (*pf)(table);
   ...
  release(pf);
   ...
}

PHASE NAME:  SORTMOD
SOURCE FILE  NAME:  SORTSUB C A

short length = 40;
short sn = 3;
void sort(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;
    }
  }
}
```

Figure 31. Qualification for C

Using Qualifiers

You can precisely specify an object, provided you know the following:

- Phase name
- Source file (compilation unit) name
- Block name.

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:

```
PHASE_NAME::>CU_NAME:>BLOCK_NAME:>object
```

PHASE_NAME is the name of the phase. It is required only when the application consists of multiple phases and when you want to change the qualification to other than the current phase. PHASE_NAME is enclosed in double quotation marks. If it is not, it must be a valid identifier in the C programming language. PHASE_NAME can also be the Debug Tool variable %LOAD.

CU_NAME is the name of the compilation unit or source file. The CU_NAME must be the fully qualified source file name. 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 (").

BLOCK_NAME is the name of the block. This has the same syntax as described in the section on “Blocks and Block Identifiers for C” on page 155. BLOCK_NAME can be the Debug Tool variable %BLOCK.

The following examples are based on Figure 31 on page 157.

- Change the file scope variable `length` defined in the compilation unit SORTSUB:

```
"SORTMOD"::>"SORTSUB":>length = 20;
```

- Assume Debug Tool gained control from `main()`. The following changes the variable `length`:

```
%LOAD::>"SORTMAIN":>length = 20;
```

  Because `length` is in the current phase and compilation unit, it can also be changed by:

```
length = 20;
```

- Assume Debug Tool gained control as shown in Figure 31 on page 157. You can break whenever the variable `temp` in phase SORTMOD changes in any of the following ways:

```
AT CHANGE temp;
AT CHANGE %BLOCK3:>temp;
AT CHANGE sort:>%BLOCK3:>temp;
AT CHANGE %BLOCK:>temp;
AT CHANGE %CU:>sort:>%BLOCK3:>temp;
AT CHANGE "SORTSUB":>sort:>%BLOCK3:>temp;
AT CHANGE "SORTMOD"::>"SORTSUB":>sort:>%BLOCK3:>temp;
```

Changing the Point of View

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 phase, to another compilation unit, to a nested block, or to a block that is not nested. The SET keyword is optional.

The following examples of changing the point of view are based on Figure 31 on page 157:

- Qualify to the second nested block in the function `sort()` while in `sort`.

```
SET QUALIFY BLOCK %BLOCK2;
```

You can do this in a number of other ways, including:

```
QUALIFY BLOCK sort:>%BLOCK2;
```

Once the point of view changes, Debug Tool has access to objects accessible from this point of view. You can specify these objects in commands without qualifiers, as in:

```
j = 3;
temp = 4;
```

- Qualify to the function `main` in the phase MAINPHS in the compilation unit SORTMAIN and list the entries of `table`.

```
QUALIFY BLOCK "MAINPHS"::>"SORTMAIN":>main();
LIST table[i];
```

Chapter 10. Using Debug Tool with COBOL Programs

This chapter provides information on the way Debug Tool interacts with COBOL.

It covers such areas as the debugging environment provided by Debug Tool, the Debug Tool subset of COBOL commands and reserved words, Debug Tool evaluation of COBOL expressions, methods of program qualification, and changing the point of view among several phases.

This chapter also discusses variables: accessing program variables, declaring temporary variables, displaying values of COBOL variables, assigning values to COBOL variables, using Debug Tool variables in COBOL, and using DBCS characters in COBOL when testing with the Debug Tool.

The Debugging Environment Provided for COBOL Programs

While Debug Tool allows you to use many commands that are either very similar or equivalent to COBOL statements, Debug Tool does not necessarily interpret these commands as required by the compiler options you chose when compiling your program. This is because in the Debug Tool environment, the following settings are in effect:

DYNAM
NOCMPR2
NODBCS
NOWORD
NUMPROC(NOPFD)
QUOTE
TRUNC(BIN)
ZWB

For more information on these compile-time options, see *IBM COBOL for VSE/ESA Language Reference*.

Debug Tool Commands

To make testing COBOL programs easier, Debug Tool allows you to write debugging commands in a manner resembling COBOL statements. It does this by providing an *interpretive subset* of COBOL language statements. This interpretive subset is a list of commands recognized by Debug Tool that either closely resemble or duplicate the syntax and action of the appropriate COBOL statements. This not only allows you to work with familiar commands, but also simplifies the insertion into your source code of program patches developed while in your Debug Tool session.

The interpretive subset of COBOL statements recognized by Debug Tool is shown in Table 20 on page 349. This subset of statements is valid only when the current programming language is COBOL.

For specific usage notes concerning each command, see the appropriate section of Part 3, “Debug Tool Reference” on page 193.

© Copyright IBM Corp. 1995, 1996

For explanations of COBOL statement usage and keyword meaning, see *IBM COBOL for VSE/ESA Language Reference*.

Restrictions on COBOL-like Commands

Some restrictions apply to the use of the COBOL commands COMPUTE, MOVE, and SET; the conditional execution command, IF; the multiway switch, EVALUATE; the iterative looping command, PERFORM; and the subroutine call, CALL. The restrictions listed below for each command are in addition to restrictions found in *IBM COBOL for VSE/ESA Language Reference*.

COMPUTE

When using COMPUTE to assign the value of an arithmetic expression to a variable, keep the following restrictions in mind:

- COMPUTE can assign a value to only one identifier.
- If Debug Tool was invoked due to a computational condition or an attention interrupt, using an assignment to set a variable might not give the results you expect. This is due to the uncertainty of variable values within statements as opposed to their values at statement boundaries.
- The following phrases are not supported: ROUNDED, SIZE ERROR, and END-COMPUTE.
- The keyword EQUAL is not supported (= must be used).
- If the arithmetic expression in the COMPUTE operation consists of only one numeric operand, the command is treated as a MOVE command. Therefore, the same restrictions that apply to the MOVE command also apply to the COMPUTE command.

For more information, see “COMPUTE Command (COBOL)” on page 245 and *IBM COBOL for VSE/ESA Language Reference*.

MOVE

When using MOVE to assign the value of one program, session, or Debug Tool variable, or literal to another program, session, or Debug Tool variable, keep in mind the following restrictions:

- MOVE can assign a value to only one identifier.
- If Debug Tool was invoked due to a computational condition or an attention interrupt, using an assignment to set a variable might not give the results you expect. This is due to the uncertainty of variable values within statements, as opposed to their values at statement boundaries.
- The CORRESPONDING phrase is not supported.

Table 22 on page 351 shows the permissible moves for the MOVE command.

For more information, see “MOVE Command (COBOL)” on page 286 and *IBM COBOL for VSE/ESA Language Reference*.

SET

While using the SET command, keep the following restrictions in mind:

- Only a single receiver is allowed.
- Only the sender-receiver combinations listed in Table 23 on page 352 are supported.
- If Debug Tool was invoked due to a computational condition or an attention interrupt, using an assignment to set a variable might not give the results you expect. This is due to the uncertainty of variable values within statements as opposed to their values at statement boundaries.
- Only Formats 1, 5, and 7 of the COBOL SET command are supported.

Additionally, Debug Tool provides a hexadecimal constant that can be used with the SET command, where the hexadecimal value is denoted by an "H" and delimited by quotation marks or apostrophes. For more information on this constant, see “Using Constants in Expressions” on page 173.

For more information, see “SET Command (COBOL)” on page 326 and *IBM COBOL for VSE/ESA Language Reference*.

IF

When using the IF command, keep in mind the following restrictions:

- Only simple relation conditions are supported.
- The NEXT SENTENCE phrase is not supported.
- Only the comparisons shown in Table 21 on page 349 are supported.

For more information, see “IF Command (COBOL)” on page 269 and *IBM COBOL for VSE/ESA Language Reference*.

EVALUATE

When using the EVALUATE command, keep in mind the following restrictions:

- Only a single subject is supported.
- Consecutive WHENs without associated commands are not supported.
- THROUGH/THRU ranges must be specified as literal constants.
- Only simple relation conditions are supported.
- Debug Tool implements the EVALUATE command as a series of IF commands. As a result, only the comparisons shown in Table 21 on page 349 are supported.

For more information, see “EVALUATE Command (COBOL)” on page 260 and *IBM COBOL for VSE/ESA Language Reference*.

PERFORM

When using the PERFORM command, keep in mind the following restrictions:

- Only inline PERFORM commands are supported (but the PERFORM command can be a Debug Tool procedure invocation).
- Only simple relation conditions are supported.
- The AFTER phrase is not supported.
- Index names and floating-point variables cannot be used as the varying identifiers.

- Index names are not supported in the BY-phase.

For more information, see “PERFORM Command (COBOL)” on page 291 and *IBM COBOL for VSE/ESA Language Reference*.

CALL

When using the CALL command, keep in mind the following restrictions:

- The ON OVERFLOW and ON EXCEPTION phrases are not supported. Consequently, END-CALL is not supported.
- Only CALL commands to separately compiled programs are supported. You cannot CALL nested programs, although they can be invoked by GOTO or STEP to a compiled-in CALL command.
- All CALLs are dynamic. The called program is loaded when it is called.

For more information, see “CALL Command” on page 234 and *IBM COBOL for VSE/ESA Language Reference*.

COBOL Command Format

When you are entering commands directly at your terminal, 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.

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 temporary variables to suit your testing needs. The following sections describe these tasks.

Accessing 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. You make the symbol table available to Debug Tool by compiling with the compile-time TEST(SYM) option. (See "Compiling a COBOL Program with the Compile-Time TEST Option" on page 16 for details about the compile-time TEST option.)

Assigning Values to COBOL Variables

Debug Tool provides three COBOL-like commands to use when assigning values to variables—SET, MOVE, and COMPUTE.

Note: All examples concerning SET, MOVE, and COMPUTE refer to the declarations in the COBOL program segment shown in Figure 32. The examples concerning LIST, found in "Displaying Values of COBOL Variables" on page 166, also refer to this program segment.

```
01  GRP.
    02  ITM-1 OCCURS 3 TIMES INDEXED BY INX1.
        03  ITM-2 PIC 9(3) OCCURS 3 TIMES INDEXED BY INX2.
01  B.
    02  A     PIC 9(10).
01  D.
    02  C     PIC 9(10).
01  F.
    02. E     PIC 9(10)     OCCURS 5 TIMES.
77     AA          PIC X(5)     VALUE 'ABCDE'.
77     BB          PIC X(5).
77     XX          PIC 9(9)    COMP.
77     ONE         PIC 99       VALUE 1.
77     TWO         PIC 99       VALUE 2.
77     PTR         POINTER
```

Figure 32. Sample COBOL Variable Declarations

While reading the examples of variable manipulation, refer to these declarations.

SET

SET allows you to assign values to indexes associated with index names. `inx1`, defined in Figure 32 as the index to `itm-1`, can be given the following value:

```
SET inx1 TO 3;
```

This assigns `inx1` a value of three.

You can also set index values as equal to each other, as in the following example:

```
SET inx2 TO inx1;
```

This assigns the value of `inx1` to `inx2`.

With SET, you can set pointers. The following example:

```
SET ptr TO NULL;
```

assigns the value of an invalid address (nonnumeric 0) to `ptr` and:

```
SET ptr TO ADDRESS OF one;
```

assigns the address of `one` to `ptr`.

You can also use H-literals to set pointers. The following example:

```
SET ptr TO H'200000';
```

assigns the hexadecimal value of X'200000' to ptr.

MOVE

MOVE allows you to assign the value of one program, session, or Debug Tool variable or literal to another. The following example:

```
MOVE a OF b TO c OF d;
```

assigns to the program variable c, found in structure d, the value of the program variable a, found in structure b. Note the qualification used in this example.

The following example:

```
MOVE 123 TO itm-2(1,1);
```

assigns the value of 123 to the first table element of itm-2.

You can also use reference modification to assign values to variables as shown in the following two examples:

```
MOVE aa(2:3) TO bb;
```

and

```
MOVE aa TO bb(1:4);
```

COMPUTE

COMPUTE allows you to assign the value of an arithmetic expression to a variable. The following example:

```
COMPUTE xx = (a + e(1)) / c * 2;
```

assigns to variable xx the result of the expression (a + e(1)) / c * 2.

You can also use table elements in such assignments as shown in the following example:

```
COMPUTE itm-2(1,2) = (a + 10) / 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.

Declaring Temporary Variables

You might want or need to declare temporary variables, also known as 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 temporary variables during a Debug Tool session. For more information on COBOL variable names, see *IBM COBOL for VSE/ESA Language Reference*. Only elementary variables with the attributes shown in Table 11 on page 247 can be declared as session variables. They are accessible to other HLLs.

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.

Displaying Values of COBOL Variables

To display the values of variables, issue the LIST command. The LIST command causes Debug Tool to log and display the current values (and names, if requested) of variables. For example, if you want to display the variables aa, bb, one, and their respective values at statement 52 of your program, issue the following command:

```
AT 52 LIST TITLED (aa, bb, one); 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.

Using DBCS Characters

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;
```

The IBM default for DBCS is ON.

The DBCS syntax and continuation rules you must follow to use DBCS variables in Debug Tool commands are the same as those for the COBOL language.

For COBOL you must type a DBCS literal, such as G, in front of a DBCS value in a Monitor or Log window if you want to update the value.

See *IBM COBOL for VSE/ESA Language Reference* for discussions of DBCS usage with COBOL.

Using Debug Tool Variables in COBOL

Debug Tool variables, as shown in Table 6, provide information about your program that you can use during your session. These variables are distinguished by a percent character (**%**) as the first character in their names. To display the values of any of them during your session, issue the LIST command.

Table 6 (Page 1 of 2). Descriptions of Debug Tool Variables

Debug Tool Variable	COBOL Attributes	Description
%GPRn	PICTURE S9(9) USAGE COMP	Represents general-purpose registers.
%FPRn	USAGE COMP-1	Represents single-precision floating-point registers.
%LPRn	USAGE COMP-2	Represents double-precision floating-point registers.
%EPRn	n/a	Represents extended-precision floating-point registers; not valid in COBOL programs.
%ADDRESS	USAGE POINTER	Contains the address of the location where your program was interrupted.
%AMODE	PICTURE S9(4) USAGE COMP	Contains the current AMODE of the suspended program (either 24 or 31).
%BLOCK	PICTURE X(j)	Contains the name of the current block.
%CAAADDRESS	USAGE POINTER	Contains the address of the CAA control block associated with the suspended program.
%CONDITION	PICTURE X(j)	Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or LE/VSE condition.
%COUNTRY	PICTURE X(j)	Contains the current country code.
%CU	PICTURE X(j)	Contains the name of the primary entry point of the current compilation unit. Equivalent to %PROGRAM.
%EPA	USAGE POINTER	Contains the address of the primary entry point in the currently interrupted program.
%HARDWARE	PICTURE X(j)	Identifies the type of hardware where the application is running.
%LINE	PICTURE X(j)	Contains the current line number. For COBOL, %LINE does not return a relative verb (within the line) for labels. Equivalent to %STATEMENT.
%LOAD	PICTURE X(j)	Contains the name of the phase of the current program.
%NLANGUAGE	PICTURE X(j)	Contains the national language currently being used.

Table 6 (Page 2 of 2). Descriptions of Debug Tool Variables

Debug Tool Variable	COBOL Attributes	Description
%PATHCODE	PICTURE S9(4) USAGE COMP	Contains an integer value identifying the type of change occurring when the program flow changes.
%PLANGUAGE	PICTURE X(j)	Contains the current programming language.
%PROGRAM	PICTURE X(j)	Contains the name of the primary entry point of the current program. Equivalent to %CU.
%RC	PICTURE S9(4) USAGE COMP	Contains a return code whenever a Debug Tool command ends.
%RUNMODE	PICTURE X(j)	Contains a string identifying the presentation mode of Debug Tool.
%STATEMENT	PICTURE X(j)	Equivalent to %LINE.
%SUBSYSTEM	PICTURE X(j)	Contains the name of the underlying subsystem, if any, where the program is executing.
%SYSTEM	PICTURE X(j)	Contains the name of the operating system supporting the program.

Debug Tool variables representing general and floating-point registers can be used as the targets of assignment commands. Detailed descriptions of the Debug Tool variables follow.

%GPR0, %GPR1,...,%GPR15
Variables that represent general purpose registers at the point of interruption in a COBOL program. You can use them as targets of assignments:

```
MOVE name_table TO %GPR5;
```

When modifying register values, use care that you do not change the base register.

Notes:

1. If you change a %GPRn register, the change is reflected when you resume program execution.
2. Although assigning new values to variables %GPR12 and %GPR13 does not result in an error, when any subsequent action is taken the newly set values are reset to their previous values.

%FPR0, %FPR2, %FPR4, %FPR6
Represent short-precision floating-point registers. These variables are defined as USAGE COMP-1. You can use them as targets of assignments:

```
MOVE 3.14152 TO %FPR0;
```

%LPR0, %LPR2, %LPR4, %LPR6
Represent long-precision floating-point registers. These variables are defined as USAGE COMP-2. Similar to the short-precision floating-point registers (%FPRs), you can use these registers as targets of assignments.

%EPR0, %EPR4
Represent the extended-precision floating-point registers. These variables are not defined for COBOL programs.

%ADDRESS
Contains the address of the location where the COBOL program was interrupted.

You can use the OFFSET table in the compiler listing to determine statement numbers. To determine the offset, you can issue the following command:

```
LIST %ADDRESS - %EPA
```

%ADDRESS might not locate a statement in your COBOL program in all instances. When an error occurs outside of the program, in some instances, %ADDRESS contains the actual interrupt address. This occurs only if Debug Tool is unable to locate the last statement that was executed before control left the program.

%AMODE
Contains the current AMODE of the suspended program. Possible values are 24 or 31.

%BLOCK
Contains the name of the current block. To display the name of the current block, you can use the LIST command or issue:

```
DESCRIBE PROGRAM;
```

You can change or override the value of %BLOCK using the QUALIFY command.

%CAAADDRESS
Contains the CAA control block associated with the suspended program.

%CONDITION
Contains the name (or number) of the condition identification when Debug Tool is entered due to an HLL or LE/VSE condition.

%COUNTRY
Contains the current country code.

%CU
Contains the name of the primary entry point of the current program.

You can change or override the value of %CU by using the QUALIFY command.

%CU is equivalent to %PROGRAM.

%EPA
Contains the address of the primary entry point of the currently interrupted COBOL program.

%HARDWARE
Identifies the type of hardware where the application program is running. A possible value is 370/ESA.

%LINE
Contains the current line number. This value can include a period, since the current line can be a statement other than the first statement on a source line.

If the program is at the entry or exit of a block, %LINE contains ENTRY or EXIT, respectively.

If the line number cannot be determined (for example, a run-time line number does not exist or the address where the program is interrupted is not in the program), %LINE contains an asterisk (*). Also, for COBOL, %LINE does not return a relative verb (within the line) for labels.

%LINE is equivalent to %STATEMENT.

%LOAD
Contains the name of the current qualifying phase and is used when an unqualified reference to a program or variable is made. If the currently qualified phase is the one initially loaded, %LOAD contains a single asterisk (*).

Whenever control is transferred to Debug Tool, %LOAD is set to the name of the currently executing phase (or to an asterisk in the case of the initial phase). You can change or override the value of %LOAD by using the QUALIFY command.

Note: For phases to be recognized by Debug Tool, they must be loaded using LE/VSE services.

%NLANGUAGE
Indicates the national language currently being used. It is treated as a string in COBOL. Its possible values are:

ENGLISH
UENGLISH
JAPANESE

%PATHCODE
Contains an integer value identifying the kind of path change taking place when Debug Tool is entered because of a path breakpoint. Its possible values are:

-1 Debug Tool is not in control as the result of a path or attention situation.

0 An attention interrupt occurred.

1 A block has been entered.

2 A block is about to be exited.

3 Control has reached a label coded in the program (a paragraph name or section name).

4 Control is being transferred as a result of a CALL or INVOKE. The invoked routine's parameters, if any, have been prepared.

5 Control is returning from a CALL or INVOKE. If Register 15 contains a return code, it has already been stored.

6 Some logic contained by an inline PERFORM is about to be executed. (Out-of-line PERFORM ranges must start with a paragraph or section name, and are identified by `%PATHCODE = 3`.)

7 The logic following an IF...THEN is about to be executed.

8 The logic following an ELSE is about to be executed.

9 The logic following a WHEN within an EVALUATE is about to be executed.

10 The logic following a WHEN OTHER within an EVALUATE is about to be executed.

11 The logic following a WHEN within a SEARCH is about to be executed.

12 The logic following an AT END within a SEARCH is about to be executed.

13 The logic following the end of one of the following structures is about to be executed:

- An IF statement (with or without an ELSE clause)
- An EVALUATE or SEARCH
- A PERFORM.

14 Control is about to return from a declarative procedure such as USE AFTER ERROR. (Declarative procedures must start with section names, and are identified by `%PATHCODE = 3`.)

15 The logic associated with one of the following phrases is about to be run:

- [NOT] ON SIZE ERROR
- [NOT] ON EXCEPTION
- [NOT] ON OVERFLOW
- [NOT] AT END (other than SEARCH AT END)
- [NOT] AT END-OF-PAGE
- [NOT] INVALID KEY.

16 The logic following the end of a statement containing one of the following phrases is about to be run:

- [NOT] ON SIZE ERROR
- [NOT] ON EXCEPTION
- [NOT] ON OVERFLOW
- [NOT] AT END (other than SEARCH AT END)
- [NOT] AT END-OF-PAGE
- [NOT] INVALID KEY.

Note: Values in the range 3–16 can be assigned to %PATHCODE only if your program was compiled with an option supporting path hooks.

%PLANGUAGE
Indicates the programming language currently being used.

%PROGRAM
Contains the name of the primary entry point of the current program.

You can change or override the value of %PROGRAM by using the QUALIFY command.

%PROGRAM is equivalent to %CU.

%RC
Contains a return code whenever a Debug Tool command ends.

%RC initially has a value of zero unless the log file cannot be opened, in which case it has a value of -1.

The %RC return code is a Debug Tool variable. It is not related to the return code that can be found in Register 15.

%RUNMODE
Contains a string identifying the presentation mode of Debug Tool. Possible values are:

SCREEN
BATCH

%STATEMENT
Contains the current statement number. This value can include a period, as the current statement can be one other than the first statement in a source line.

If the program is at the entry or exit of a block, %STATEMENT contains ENTRY or EXIT, respectively.

If the statement number cannot be determined (for example, if a run-time statement number does not exist or the address where the program is interrupted is not in the program), %STATEMENT contains an asterisk (*).

%STATEMENT is equivalent to %LINE.

%SUBSYSTEM
Contains the name of the underlying subsystem, if any, where the program is running. Possible values are:

CICS
NONE

%SYSTEM
Contains the name of the operating system supporting the program. The only possible value is: VSE.

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.

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.

When both comparands are stated as simple references, all combinations listed in Table 21 on page 349 are supported.

Displaying the Results of 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);
```

See the *IBM COBOL for VSE/ESA Language Reference* for discussions of COBOL expression evaluation.

Expressions are evaluated according to COBOL rules applying to the options specified in "The Debugging Environment Provided for COBOL Programs" on page 160. Conditions are the same ones that exist for program statements.

Using Constants in 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 *IBM COBOL for VSE/ESA Language Reference* are valid in Debug Tool, with the following restriction:

- When you specify a hexadecimal (`X'n'`) constant, no padding takes place. If you need a fullword value, you must specify a full word.

  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).

Additionally, Debug Tool allows the use of a hexadecimal constant. 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

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 `01234F`.

Using the %STORAGE Function

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'
```

For more information about the functions described above, including the proper syntax, see “Debug Tool's Built-in Functions” on page 134.

Using Qualification for 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.

Using Qualifiers

Qualifiers are combinations of phases, 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:

```
PHASE_NAME::>CU_NAME:>BLOCK_NAME:>object;
```

PHASE_NAME is the name of the phase. It is required only when the program consists of multiple phases and you want to change the qualification to other than the current phase. PHASE_NAME can also be the Debug Tool variable %LOAD.

CU_NAME is the name of the compilation unit. The CU_NAME must be the fully qualified compilation unit name. 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.

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.

The following two screens are samples of two similar COBOL programs (blocks):

```
MAIN
⋮
   01  VAR1.
       02  VAR2.
           O3  VAR3     PIC XX.
   01  VAR4     PIC  99.

  ****************MOVE commands entered here****************
```

```
SUBPROG
⋮
   01  VAR1.
       02  VAR2.
           O3  VAR3     PIC XX.
   01  VAR4     PIC  99.
   01  VAR5     PIC  99.

  ****************LIST commands entered here****************
```

You can distinguish between the `main` and `subprog` blocks using qualification. If you enter the following MOVE commands when `main` is the currently executing block:

```
MOVE 8 TO var4;
MOVE 9 TO subprog:>var4;
MOVE 'A' TO var3 OF var2 OF var1;
MOVE 'B' TO subprog:>var3 OF var2 OF var1;
```

and the following LIST commands when `subprog` is the currently executing block:

```
LIST TITLED var4;
LIST TITLED main:>var4;
LIST TITLED var3 OF var2 OF var1;
LIST TITLED main:>var3 OF var2 OF var1;
```

each LIST command results in the following output (without the commentary) in your Log window:

```
VAR4 = 9;    /*  var4 with no qualification refers to a variable     */
             /*  in the currently executing block (subprog).         */
             /*  Therefore, the LIST command displays the value of 9.*/

MAIN:>VAR4 = 8     /*  var4 is qualified to main.                    */
                   /*  Therefore, the LIST command displays 8,       */
                   /*  the value of the variable declared in main.   */

VAR3 OF VAR2 OF VAR1 = 'B';
               /*  In this example, although the data qualification  */
               /*  of var3 is OF var2 OF var1, the                   */
               /*  program qualification defaults to the currently   */
               /*  executing block and the LIST command displays     */
               /*  'B', the value declared in subprog.               */

VAR3 OF VAR2 OF VAR1 = 'A'
               /*  var3 is again qualified to var2 OF var1           */
               /*  but further qualified to main.                    */
               /*  Therefore, the LIST command displays              */
               /*  'A', the value declared in main.                  */
```

The above method of qualifying variables is necessary for command files.

Changing the Point of View

The point of view is usually the currently executing block. You can also get to inaccessible data by changing the point of view using the SET QUALIFY command. The SET keyword is optional. For example, if the point of view (current execution) is in `main` and you want to issue several commands using variables declared in `subprog`, you can change the point of view by issuing the following:

```
QUALIFY BLOCK subprog;
```

You can then issue commands using the variables declared in `subprog` without using qualifiers. Debug Tool does not see the variables declared in procedure `main`. For example, the following assignment commands are valid with the `subprog` point of view:

```
MOVE 10 TO var5;
```

However, if you want to display the value of a variable in `main` while the point of view is still in `subprog`, you must use a qualifier, as shown in the following example:

```
LIST (main:>var-name);
```

The above method of changing the point of view is necessary for command files.

Chapter 11. Using Debug Tool with PL/I Programs

This chapter provides information on the way Debug Tool interacts with PL/I.

It covers such areas as the debugging environment provided by Debug Tool, the Debug Tool subset of PL/I statements, Debug Tool evaluation of PL/I expressions, methods of program qualification, and changing the point of view among several phases.

This chapter also discusses variables: accessing program variables, declaring temporary variables, displaying values of PL/I variables, assigning values to PL/I variables, using Debug Tool variables in PL/I, and support of PL/I freeform DBCS input when testing with Debug Tool.

Debug Tool Commands

To make testing PL/I programs easier, Debug Tool allows you to write debugging commands in a manner resembling PL/I statements. It does this by providing an *interpretive subset* of PL/I language statements. This interpretive subset is a list of commands recognized by Debug Tool that either closely resemble or duplicate the syntax and action of the appropriate PL/I statement. This not only allows you to work with familiar commands, but also simplifies the insertion into your source code of program patches developed while in your Debug Tool session.

The interpretive subset of PL/I statements recognized by Debug Tool is shown in Table 24 on page 353. This subset of statements is valid only when the current programming language is PL/I.

For specific usage notes concerning each command, see the appropriate section of Part 3, “Debug Tool Reference” on page 193.

For explanations of statement usage and keyword meaning, see *IBM PL/I for VSE/ESA Language Reference*.

PL/I Language Statements

PL/I statements are entered as Debug Tool *commands*. 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.

© Copyright IBM Corp. 1995, 1996

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.

Using PL/I Variables with Debug Tool

Debug Tool can process all program variables that are valid in PL/I. It allows you to assign and display the values of variables during your session. It also allows you to declare temporary variables with the recognized PL/I declarations to suit your testing needs.

Accessing 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) (see “Compiling a PL/I Program with the Compile-Time TEST Option” on page 19 for more information).

Displaying Values of PL/I Variables or Expressions

To display the values of variables or expressions, issue 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. See “LIST Command” on page 272 for more information.

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.

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 at any time.

Variables that are based on:

- An OFFSET variable,
- An expression,
- A pointer that either is based or defined,
- A parameter, or
- A 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 DX 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 attribute that specifies something other than a constant
- Variables that are members of a based structure declared with the REFER options.

Structures

You cannot directly 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 ) );
```

You might want or need to declare temporary variables, also known as session variables, during your Debug Tool session. Debug Tool supports all PL/I scalar session variables. In addition, arrays and structures may be declared.

The relevant variable assignment commands are similar to their counterparts in the PL/I language. The rules used for forming variable names in PL/I also apply to the declaration of temporary variables during a Debug Tool session. For more information on PL/I variable names, see *IBM PL/I for VSE/ESA Language Reference*.

Refer to Table 11 on page 247 for variables whose attributes will let them be properly used by other programming languages.

You cannot initialize temporary variables in declarations. However, you can use an assignment statement or function call to initialize a temporary variable.

To declare a floating-point variable called `maxi`, enter the following PL/I declaration:

```
dcl maxi float dec(6) ;
```

In Debug Tool you can declare coded arithmetic, string, event, label, and locator variables, as well as arrays and structures of these.

If you declare a temporary variable with the same name as a programming variable, the temporary 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.

For more on qualification, see “Using Qualification for PL/I” on page 189. For more on declarations, see “DECLARE Command (PL/I)” on page 251.

LIST STORAGE

For LIST STORAGE *address*, *address* can be a POINTER, a hexadecimal PX constant, or the ADDR built-in function.

Assigning Values to PL/I Variables

To assign a value to a PL/I variable, you use an assignment command. See "Assignment Command (PL/I)" on page 207 for syntax information. Assignment commands assign a value to the left operand. The left operand must be a modifiable variable.

PL/I contains two types of assignment operators: simple and compound. A simple assignment operator gives the value of the right operand to the left operand.

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 `ndex` plus 2 to the variable `ndex`:

```
ndex = ndex + 2;
```

Using Debug Tool Variables in PL/I

Debug Tool variables, as shown in Table 7, provide information about your program that you can use during your session. These variables are distinguished by a percent character (**%**) as the first character in their names. To display the values of any of them during your session, use the LIST command.

Table 7 (Page 1 of 2). PL/I Attributes for Debug Tool Variables

Debug Tool Variable	PL/I Attributes	Description
%GPRn	FIXED BIN(31,0)	Represents general-purpose registers.
%FPRn	FLOAT DEC(6)	Represents single-precision floating-point registers.
%LPRn	FLOAT DEC(16)	Represents double-precision floating-point registers.
%EPRn	FLOAT DEC(33)	Represents extended-precision floating-point registers.
%ADDRESS	POINTER	Contains the address of the location where your program was interrupted.
%AMODE	FIXED BIN(15,0)	Contains the current AMODE of the suspended program (either 24 or 31).
%BLOCK	CHAR(j)	Contains the name of the current block.
%CAAADDRESS	POINTER	Contains the address of the CAA control block associated with the suspended program.
%CONDITION	CHAR(j)	Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or LE/VSE condition.

Table 7 (Page 2 of 2). PL/I Attributes for Debug Tool Variables

Debug Tool Variable	PL/I Attributes	Description
%COUNTRY	CHAR(j)	Contains the current country code.
%CU	CHAR(j)	Contains the name of the current compilation unit. Equivalent to %PROGRAM.
%EPA	POINTER	Contains the address of the primary entry point in the currently interrupted program.
%HARDWARE	CHAR(j)	Identifies the type of hardware where the application is running.
%LINE	CHAR(j)	Contains the current line number. Equivalent to %STATEMENT.
%LOAD	CHAR(j)	Contains the name of the phase of the current program.
%NLANGUAGE	CHAR(j)	Contains the national language currently being used.
%PATHCODE	FIXED BIN(15,0)	Contains an integer value identifying the type of change occurring when Debug Tool is entered because of a path breakpoint.
%PLANGUAGE	CHAR(j)	Contains the current programming language.
%PROGRAM	CHAR(j)	Contains the name of the primary entry point of the current program. Equivalent to %CU.
%RC	FIXED BIN(15,0)	Contains a return code whenever a Debug Tool command ends.
%RUNMODE	CHAR(j)	Contains a string identifying the presentation mode of Debug Tool.
%STATEMENT	CHAR(j)	Contains the current statement number. Equivalent to %LINE.
%SUBSYSTEM	CHAR(j)	Contains the name of the underlying subsystem, if any, where the program is executing.
%SYSTEM	CHAR(j)	Contains the name of the operating system supporting the program.

You can use all Debug Tool variables in expressions. Additionally, the variables representing general and floating-point registers are modifiable and can be used as the targets of assignment commands.

Note: When modifying register values, do not modify the base register.

Detailed descriptions of the Debug Tool variables follow.

%GPR0, %GPR1,...,%GPR15

Represent general-purpose registers at the point of interruption in a PL/I program. You can use them in expressions:

```
list (%GPR5 + 10);
```

and as targets of assignments:

```
%GPR5 = nametable;
```

Notes:

- If you change a %GPRn register, the change is reflected when you resume program execution.
- Only %GPR12 can be used at external entry.
- Assigning new values to variables %GPR12 and %GPR13 does not result in an error, however, subsequent processing by Debug Tool will reset them to their previous values.
- It is possible to cause the program base register to be lost if you change that general-purpose register.

%FPR0, %FPR2, %FPR4, %FPR6
Represent single-precision floating-point registers and are equivalent to FLOAT DEC(6) or FLOAT BIN(21) variables. You can use them in expressions:

```
x = %FPR4 / 6.3;
```

and as targets of assignments:

```
%FPR0 = 3.14152;
```

%LPR0, %LPR2, %LPR4, %LPR6
Represent the double-precision floating-point registers and are equivalent to FLOAT DEC(16) or FLOAT BIN(53) variables. Similar to the single-precision floating-point registers (%FPRs), you can use these registers in expressions and as targets of assignments.

%EPR0, %EPR4
Represent the extended-precision floating-point registers, and are equivalent to FLOAT DEC(33) or FLOAT BIN(109) variables. Similar to the single-precision floating-point registers (%FPRs), you can use these registers in expressions and as targets of assignments.

%ADDRESS
Contains the address of the location where the program was interrupted.

%AMODE
Contains the current AMODE of the suspended program. Possible values are 24 or 31.

%BLOCK
Contains the name of the current block. To display the name of the current block, you can use the LIST command or issue:

```
DESCRIBE PROGRAM;
```

You can change or override the value of %BLOCK by using the SET QUALIFY command.

%CAAADDRESS
Contains the address of the CAA control block associated with the suspended program.

%CONDITION
Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or LE/VSE condition.

%COUNTRY
Contains the current country code.

%CU
Contains the name of the primary entry point of the current program.

You can change or override the value of %CU by using the QUALIFY command.

%CU is equivalent to %PROGRAM.

%EPA
Contains the address of the primary entry point of the currently interrupted program.

%HARDWARE
Identifies the type of hardware where the application program is running. A possible value is 370/ESA.

%LINE
Contains the current line (statement) number. This value can include a period since the current line can be a statement other than the first statement on a source line. For example, if `%LINE = 5.5`, the current statement is the fifth statement on the fifth source line.

If the program is at the entry or exit of a block, %LINE contains ENTRY or EXIT respectively.

If the line number cannot be determined (for example, if a run-time line number does not exist or the address where the program is interrupted is not in the program), %LINE contains an asterisk (*).

%LINE is equivalent to %STATEMENT.

%LOAD
Contains the name of the currently qualified phase and is used when an unqualified reference to a program or variable is made. If the currently qualified phase is the one initially loaded, %LOAD contains a single asterisk (*).

Whenever control is transferred to Debug Tool, %LOAD is set to the name of the currently executing phase (or to an asterisk in the case of the initial phase). You can change or override the value of %LOAD by using the SET QUALIFY command.

For phases to be recognized by Debug Tool, they must have been loaded by a language call and not through a direct operating system load command.

%NLANGUAGE
Indicates the national language currently in use. Its possible values include:

ENGLISH
UENGLISH
JAPANESE

%PATHCODE
Contains an integer value identifying the kind of path change taking place when Debug Tool is entered because of a path breakpoint. Possible values are:

-1 Debug Tool is not in control as the result of a path or attention situation.

0 An attention interrupt occurred.

1 A block has been entered.

2 A block is about to be exited.

3 Control has reached a label constant.

4 Control is being sent somewhere else as the result of a CALL or a function reference.

5 Control is returning from a CALL invocation or a function reference. Register 15, if it contains a return code, has not yet been stored.

6 Some logic contained in a complex DO statement is about to be executed.

7 The logic following an IF..THEN is about to be executed.

8 The logic following an ELSE is about to be executed.

9 The logic following a WHEN within a select-group is about to be executed.

10 The logic following an OTHERWISE within a select-group is about to be executed.

Values in the range 3–10 can only be assigned to %PATHCODE if your program was compiled with an option supporting path hooks.

%PLANGUAGE
Indicates the programming language currently in use.

%PROGRAM
The name of the primary entry point of the current program.

You can change or override the value of %PROGRAM by using the QUALIFY command.

%PROGRAM is equivalent to %CU.

%RC
Contains a return code whenever a Debug Tool command ends.

%RC initially has a value of zero unless the log file cannot be opened, in which case it has a value of -1.

The %RC return code is a Debug Tool variable. It is not related to the return code that can be found in Register 15.

%RUNMODE
Contains a string identifying the presentation mode of Debug Tool. Possible values are:

SCREEN
BATCH

%STATEMENT
Contains the current statement number. This value can include a period since the current statement can be one other than the first statement in a source line.

If the program is at the entry or exit of a block, %STATEMENT contains ENTRY or EXIT, respectively.

If the statement number cannot be determined (for example, if a run-time statement number does not exist or the address where the program is interrupted is not in the program), %STATEMENT contains an asterisk (*).

%STATEMENT is equivalent to %LINE.

%SUBSYSTEM
Contains the name of the underlying subsystem, if any, where the program is executing. Possible values are:

CICS
NONE

%SYSTEM
Contains the name of the operating system supporting the program. The only possible value is: VSE.

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 restrictions as noted in "Unsupported PL/I Language Elements" on page 354.

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.

Note: A PX constant allows the input of a hexadecimal value in the format '000abcde'PX, where 000ABCDE is a valid hexadecimal number.

Using DBCS Characters - Freeform Input

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:

```
LIST NAMES  db<.c.skk.w>ord
```

will search for

```
<.D.B.C.Skk.W.O.R.D>
```

This will result in different behavior depending upon the language. For example, the following will find a<kk>b in C and <.Akk.b> in PL/I.

```
LIST NAMES a<kk>*
```

where <kk> is shiftout-kanji-shiftin.

Freeform will be added to the parser and will be in effect while the current programming language is PL/I.

PL/I Built-In Functions

Debug Tool supports the following PL/I built-in functions:

Table 8. PL/I Built-In Functions

ABS	CSTG[2]	LOG1	REAL
ACOS	CURRENTSTORAGE	LOG2	REPEAT
ADDR	DATAFIELD	LOW	SAMEKEY
ALL	DATE	MPSTR	SIN
ALLOCATION	DATETIME	NULL	SIND
ANY	DIM	OFFSET	SINH
ASIN	EMPTY	ONCHAR	SQRT
ATAN	ENTRYADDR	ONCODE	STATUS
ATAND	ERF	ONCOUNT	STORAGE
ATANH	ERFC	ONFILE	STRING
BINARYVALUE	EXP	ONKEY	SUBSTR
BINVALUE[1]	GRAPHIC	ONLOC	SYSNULL
BIT	HBOUND	ONSOURCE	TAN
BOOL	HEX	PLIRETV	TAND
CHAR	HIGH	POINTER	TANH
COMPLETION	IMAG	POINTERADD	TIME
COS	LBOUND	POINTERVALUE	TRANSLATE
COSD	LENGTH	PTRADD[3]	UNSPEC
COSH	LINENO	PTRVALUE[4]	VERIFY
COUNT	LOG		

Notes:

1. Abbreviation of BINARYVALUE.
2. Abbreviation of CURRENTSTORAGE.
3. Abbreviation of POINTERADD.
4. Abbreviation of POINTERVALUE.

Using SET WARNING Command with Built-Ins

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.

Using Debug Tool Functions with PL/I

Debug Tool provides built-in functions for use during a debugging session. These functions allow greater access to your programming environment and greater control over your debugging session. Using these functions, you can reference storage, translate the values of operands to hexadecimal characters, or access a variable or parameter during a specific instance of a recursive procedure.

Using %GENERATION

You can use %GENERATION to access a specific generation of a controlled variable in your program. For example, if you have a program that allocates three generations of the controlled variable `contvar` and you want to examine the contents of the first generation of `contvar`, you can enter:

```
LIST %GENERATION(contvar,1);
```

If you want to assign the contents of the second generation of `contvar` to the latest generation, you can enter:

```
%GENERATION(contvar,ALLOCATION(contvar)) = %GENERATION(contvar,2);
```

Using %HEX

When used with the LIST command, %HEX allows you to display the value of an operand as a hexadecimal character string. For example, if you want to examine the internal representation of the packed decimal variable `zvar1` whose external representation is 235, you can enter:

```
LIST %HEX(zvar1);
```

The hexadecimal value of `235C` is displayed in the Log window.

Using %STORAGE

%STORAGE allows you to reference storage by address and length. By using %STORAGE 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 ('00022222'PX, 8)
   LIST 'Storage has changed at Hex address 22222'
```

Using %RECURSION

%RECURSION allows you to access an automatic variable or a parameter in a specific instance of a recursive function. When you use %RECURSION, remember that:

- If the expression has a value of 1, the oldest generation is referenced. The higher the value of the expression, the more recent the generation of the variable Debug Tool references.
- %RECURSION can be used like a Debug Tool variable.

Using %INSTANCES

%INSTANCES returns the maximum value of %RECURSION (that is, the most recent recursion number) for a given block. %INSTANCES can be used like a Debug Tool variable.

%INSTANCES and %RECURSION can be used together to determine the number of times a function is recursively called. They can also give you access to an automatic variable or parameter in a specific instance of a recursive procedure. Assume, for example, your program contains these statements:

```
PRMAIN: procedure;
...
RECFN: procedure(cnt) returns (fixed bin(31,0));
  dcl 1 cnt fixed bin (31,0);
  cnt = cnt - 1;
  if cnt = 0 then
    call PLITEST;
  return(0);
end RECFN;
```

At this point, the call to PLITEST gives control to Debug Tool, and you are prompted for commands. If you enter:

```
LIST %INSTANCES(cnt);
```

Your Log window displays the number of times RECFN was interactively called.

If you enter:

```
%RECURSION(i, 1);
```

you receive the value of `'i'` at the first call of RECFN.

If necessary, you can use qualification to specify the parameter. For example, if the current point of execution is in `%block2`, and `%block3` is a recursive function containing the variable `x`, you can write an expression using `x` by qualifying the variable, as follows:

```
%RECURSION(prmain:>%block3:>x, %INSTANCES(prmain:>%block3:>x, y+3)) = 10;
```

For the proper syntax of the functions described above, see “Debug Tool's Built-in Functions” on page 134.

Using Qualification for PL/I

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 `x = 5;` 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.

Using Qualifiers

Qualifiers are combinations of phases, compile units, or blocks punctuated by a combination of greater-than signs (>), and colons, that precede the “object” they qualify. For example, the following is a fully qualified object:

```
PHASE_NAME::>CU_NAME:>BLOCK_NAME:>object;
```

PHASE_NAME is the name of the phase. It is required only when the program consists of multiple phases and you want to change the qualification to other than the current phase. PHASE_NAME can also be the Debug Tool variable %LOAD.

CU_NAME is the name of the compilation unit. The CU_NAME must be the fully qualified compilation unit name. 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.

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.

If variable names are unique, or defined as GLOBAL, they do not need to be qualified at the block level.

The following two screens are samples of two similar PL/I programs (blocks):

```
PRMAIN: PROCEDURE;
:
   dcl 1 VAR1,
         2 VAR2,
           3 VAR3     CHAR(2),
       1 VAR4         FIXED BIN(15,0);

  /* Assignment commands entered here */
```

```
SUBPROG: PROCEDURE;
:
   dcl 1 VAR1,
         2 VAR2,
           3 VAR3     CHAR(2),
       1 VAR4         FIXED BIN(15,0),
       1 VAR5         CHAR(2);

  /* LIST commands entered here */
```

You can distinguish between the `prmain` and `subprog` blocks using qualification. If you enter the following assignment commands when `prmain` is the currently executing block:

```
var4 = 8;
subprog:>var4 = 9;
var1.var2.var3 = 'A';
subprog:>var1.var2.var3 = 'B';
```

and the following LIST commands when `subprog` is the currently executing block:

```
LIST TITLED var4;
LIST TITLED prmain:>var4;
LIST TITLED var1.var2.var3;
LIST TITLED prmain:>var1.var2.var3;
```

each LIST command results in the following output (without the comments) in your Log window:

```
VAR4 = 9;    /*  var4 with no qualification refers to a variable     */
             /*  in the currently executing block (subprog).         */
             /*  Therefore, the LIST command displays the value of 9.*/

PRMAIN:>VAR4 = 8   /*  var4 is qualified to prmain.                  */
                   /*  Therefore, the LIST command displays 8,       */
                   /*  the value of the variable declared in prmain. */

VAR1.VAR2.VAR3 = 'B';
               /*  In this example, although the data qualification  */
               /*  of var3 is var1.var2.var3, the program            */
               /*  qualification defaults to the currently           */
               /*  executing block and the LIST command displays     */
               /*  'B', the value declared in subprog.               */

PRMAIN:>VAR1.VAR2.VAR3 = 'A'
               /*  var3 is again qualified to var1.var2.var3         */
               /*  but further qualified to prmain.                  */
               /*  Therefore, the LIST command displays              */
               /*  'A', the value declared in prmain.                */
```

The above method of qualifying variables is necessary for command files.

Changing the Point of View

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 `prmain` and you want to issue several commands using variables declared in `subprog`, you can change the point of view by issuing the following:

```
QUALIFY BLOCK subprog;
```

You can then issue commands using the variables declared in `subprog` without using qualifiers. Debug Tool does not see the variables declared in procedure `prmain`. For example, the following assignment commands are valid with the `subprog` point of view:

```
var5 = 10;
```

However, if you want to display the value of a variable in `prmain` while the point of view is still in `subprog`, you must use a qualifier, as shown in the following example:

```
LIST (main:>var-name);
```

The above method of changing the point of view is necessary for command files.

Part 3. Debug Tool Reference

© Copyright IBM Corp. 1995, 1996

Chapter 12. Using Debug Tool Commands

This chapter describes Debug Tool's windowed interfaces, command usage modes, alternate methods of command input, variables, and common syntax elements. It also gives you task-oriented information such as interpreting checklist boxes, entering commands, getting help, qualifying variables, and changing the point of view.

Command Modes and Language Support

Commands can be issued in two modes: full-screen and batch. In addition, some commands are valid only in certain programming languages or operation modes. Unless otherwise noted, Debug Tool commands are valid in all modes, and for all supported languages.

Entering Commands

This section provides information for entering commands in Debug Tool. It explains the command format, the character set and case, abbreviating or truncating keywords, continuing multiline commands, the significance of blanks, using comments or constants, and retrieving commands from the Log or Source windows.

Command Format

For input typed directly at the terminal, input is free-form, optionally starting in column 1. Separate multiple commands on a line with semicolons. The terminating semicolon (;) is optional for a single command, or the last command in a sequence of commands.

For input that comes from a primary commands or USE file, all of the Debug Tool commands must be terminated with a semicolon except for the C `block` command. The commands must conform to the syntax of the current programming language. For example, for COBOL they must start in or beyond column 8 and not continue beyond column 72.

Character Set and Case

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

Using 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).

© Copyright IBM Corp. 1995, 1996

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`.

For more details on using DBCS characters, see “SET DBCS” on page 306.

Using 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 operations: bitwise or (|), logical or (||), and bitwise assignment or (|=).
- There are alternate code points for the following C characters: vertical bar (|), left brace ({), right brace (}), left bracket ([), and right bracket (]). Although alternate code points will be accepted as input for the braces and brackets, the primary code points will always be logged. See *LE/VSE C Run-Time Programming Guide* for an explanation of the alternate and primary code points in C.

Using 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 Keywords

When you issue the Debug Tool commands, you can truncate most command keywords. You cannot truncate:

- reserved keywords for the different programming languages,
- 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.
- PROCEDURE can only be abbreviated as PROC.

The 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 COMMENT and the current programming language setting is C, the "=" must be abutted to the reference, as in "COMMENT<no space>= 2;" not "COMMENT<space>= 2;". If you want to define a procedure named USE, you must enter "USE<no space>: procedure;" not "USE<space>: procedure;".

When you truncate, you need only enter enough characters of the command to distinguish the command from all other valid Debug Tool commands. You should *not* use truncations in a commands file or compile them into programs because

they may become ambiguous in a subsequent release. The following shows examples of Debug Tool command truncations.

If you enter the following command...	It will be interpreted as...
A 3	AT 3
G	GO
Q B B	QUALIFY BLOCK B
Q Q	QUERY QUALIFY
Q	QUIT

If you specify a truncation that is also a variable in your program, the keyword is chosen if this is the only ambiguity. For example, `LIST A` does not display the value of variable A, but executes the LIST AT command, listing your current AT breakpoints. To display the value of A, issue `LIST (A)`.

In addition, ambiguous commands that cannot be resolved cause an error message and are not performed (that is, there are two commands that could be interpreted by the truncation specified). For example, `D A A;` is an ambiguous truncation since it could either be `DESCRIBE ATTRIBUTES a;` or `DISABLE AT APPEARANCE;`. Instead, you would have to enter `DE A A;` if you wanted `DESCRIBE ATTRIBUTES a;` or `DI A A;` if you wanted `DISABLE AT APPEARANCE;`. There are, of course, other variations that would work as well (for example, `D ATT A;`).

Continuation (Full-screen 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 very vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -
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, if you want to enter "i--", you could enter "(i--)" or "i--;". When the current programming language setting is 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.

Using File Input

The rules for line continuation when input comes from a commands file are language-specific:

- When the current programming language setting is 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 very vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv\
very long string");
```

- When the current programming language setting is COBOL, columns 1 - 6 are ignored by Debug Tool and input can be continued from one line to the next if the SBCS hyphen (-) is used in column 7 of the next line. Command text must begin in column 8 or later and end in or before column 72.

  In literal string continuation, 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:

  ```
  123456 LIST (" this is a very very very vvvvvvvvvvvvvvvvvvvvvvvv"
  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

Significance of Blanks

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.

Comments

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.

- 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.

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.

Constants

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. See “C Expressions” on page 145 or “Using Constants in Expressions” on page 173 for more information.

Additionally, Debug Tool allows the use of hexadecimal constants in COBOL and PL/I.

The COBOL *H constant* is a fullword 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. The COBOL hexadecimal notation for non-numeric literals, such as MOVE X'C1C2C3C4' TO NON-PTR-VAR, should be used for all other situations where a hexadecimal value is needed.

For example, to display the contents at a given address in hexadecimal format, specify:

```
LIST STORAGE (H'20CD0');
```

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);
```

For COBOL only: 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";
```

Retrieving Commands from the Log and Source Windows

When the SCREEN setting is ON, you can retrieve commands from your Log and Source windows and have Debug Tool insert them on the command line.

To retrieve a line, move the cursor to the desired line in the Log or Source window, modify it (delete the leading blank, for example), and press Enter. The input line appears on the command line so you can further modify it as necessary. Press Enter to issue the command.

When retrieving long or multiple Debug Tool commands, a full-screen pop-up window is displayed, with the command as typed in so far. However, trailing blanks on the last line are removed. The window can be expanded by placing the cursor below the pop-up window and pressing Enter. See also “RETRIEVE Command (Full-Screen Mode)” on page 298.

Online Command Syntax Help

Command syntax help is available to you. That is, if you are uncertain about the proper syntax or exact keywords required by a command, type a question mark (?) on the command line and press Enter. For example, in COBOL, if you issue ?, Debug Tool displays the output in the following format:

```
THE NEXT WORD MAY BE ONE OF:
;               DISABLE         INPUT           QUIT
AT              ENABLE          LIST            RETRIEVE
CALL            END             MONITOR         RUN
CLEAR           EVALUATE        MOVE            SCROLL
COMMENT         FIND            PANEL           SET
COMPUTE         GO              PERFORM         STEP
CURSOR          GO TO           PROCEDURE NAME  TRIGGER
DECLARATION     GOTO            QUALIFY         USE
DESCRIBE        IF              QUERY           WINDOW
```

The above output sample is meant to illustrate a point and might not appear exactly as shown.

Note: DECLARE (or DECLARATION) is not a command but a method of making an interactive variable or tag declaration.

If you are in the process of entering a command and want to verify what the next command element should be, you can enter as much of the command as you know followed by a question mark. For example, let's assume you are issuing a form of the SCROLL command (Full-Screen Mode only) and you want to know the possible command elements, enter:

```
SCROLL ?
```

Debug Tool displays the output in the following format:

```
The partially parsed command is:
 SCROLL
The next word may be one of:
BOTTOM       RIGHT
DOWN         TO
LEFT         TOP
NEXT         UP
```

The COMMENT command followed by ? does not invoke the syntax help.

Common Syntax Elements

Several syntax elements are used in many Debug Tool commands. They are described in this subsection. Some of the following syntax elements are generic and do not include a syntax diagram.

Block_Name

A *block_name* identifies:

- A C function or a block statement
- A COBOL nested program contained within a complete COBOL program
- A PL/I block

The current block qualification can be changed using the SET QUALIFY BLOCK command.

For COBOL Only

Enclose the block name in double (“) or single (‘) quotes if it is case sensitive. If the name is not inside quotes, Debug Tool will convert the name to upper case.

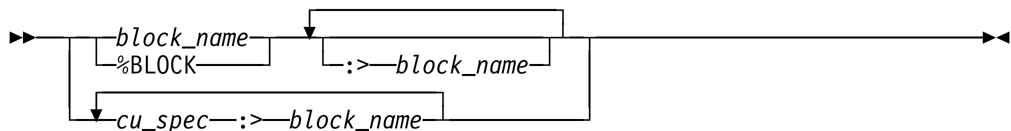
If a name contains an internal double quote, you should enclose the name in single quotes. Similarly, if the name contains an internal single quote, you should enclose the name in double quotes.

End of For COBOL Only

You can only use *block_name* for blocks known in the current enclave.

Block_Spec

A *block_spec* identifies a block in the program being debugged.



block_name
A valid block name; see "Block_Name."

%BLOCK
Represents the currently qualified block. See Table 4 on page 126.

cu_spec
A valid compile unit specification; see "CU_Spec."

You can only use *block_name* for blocks known in the current enclave.

Compile_Unit_Name

A *compile_unit_name* identifies:

- A C source file
- A COBOL program
- The external procedure name of a PL/I program.

For COBOL Only

Enclose the compile unit name in double (") or single (') quotes if it is case sensitive. If the name is not inside quotes, Debug Tool will convert the name to upper case.

End of For COBOL Only

For PL/I only

The compile unit name can optionally be enclosed in single quotes (').

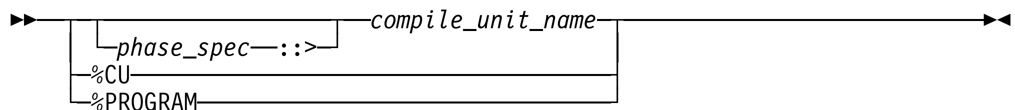
End of For PL/I only

If the compile unit name is not a valid identifier in the current programming language, it must be entered as a character string constant in the current programming language.

The current compile unit qualification can be changed using the SET QUALIFY CU command.

CU_Spec

A *cu_spec* identifies a compile unit in the application being debugged. In PL/I, the compile unit name is the same as the outer-most procedure name in the program.



phase_spec
A valid phase specification; see "Load_Spec" on page 203. If omitted, the current phase qualification is used.

compile_unit_name
A valid compile unit name; see "Compile_Unit_Name."

%CU
Represents the currently qualified compile unit. See Table 4 on page 126. %CU is equivalent to %PROGRAM.

%PROGRAM
Is equivalent to %CU. See Table 4 on page 126.

You can only use *cu_spec* to specify compile units in an enclave that is currently running. You can therefore only qualify variable names, function names, labels, and statement_ids to blocks within compile units in the current enclave.

Expression

An *expression* is a combination of *references* (see "References" on page 203 for more information) and operators that result in a value. For example, it can be a single constant, a program, session, or Debug Tool variable, a built-in function reference, or a combination of constants, variables, and built-in function references, or operators and punctuation (such as parentheses).

Particular rules for forming an expression depend on the current programming language setting and what release level of the language run-time library under which Debug Tool is running. For example, if you upgrade your version of the HLL compiler without upgrading your version of Debug Tool, certain application programming interface inconsistencies might exist.

For more about expressions with each particular HLL, see:

Chapter 9, "Using Debug Tool with C Programs" on page 138,

"Debug Tool Evaluation of COBOL Expressions" on page 172, or

"PL/I Expressions" on page 186.

You can only use expressions for variables contained in the current enclave.

Phase_Name

A *phase_name* is the name of a sublibrary member that has been loaded by a supported HLL load service.

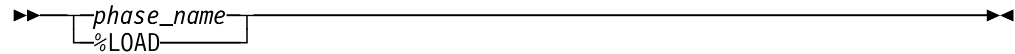
For C, escape sequences in phase names that are specified as strings are not processed if the string is part of a qualification statement.

If omitted from a name that allows it as a qualifier, the current phase qualification is assumed. It can be changed using the SET QUALIFY LOAD command.

If two enclaves contain duplicate phases, references to compile units in the phases will be ambiguous, and will be flagged as errors. However, if the compile unit is in the currently executing phase, that phase is assumed and no check for ambiguity will be performed. Therefore, for Debug Tool, phase names must be unique.

Load_Spec

A *load_spec* identifies a phase in the program being debugged.



phase_name
A valid phase name; see "Phase_Name" on page 202. This can be specified as a string constant in the current programming language, for example, a string literal in C or a character literal in COBOL. If not specified as such, it must be a valid identifier in the current programming language.

%LOAD
Represents the currently qualified phase. See Table 4 on page 126.

References

A *reference* is a subset of an *expression* that resolves to an area of storage; that is, a possible target of an assignment statement. For example, it can be a program, session, or Debug Tool variable, an array or array element, or a structure or structure element, and any of these can be pointer-qualified (in programming languages that allow it). Any identifying name in a reference can be optionally qualified by containing structure names and names of blocks where the item is visible. It is optionally followed by subscript and substring modifiers, following the rules of the current programming language.

The specification of a qualified reference includes all containing structures and blocks as qualifiers, and can optionally begin with a phase name qualifier. For example, when the current programming language setting is C, `phs::>cu:>proc:>struc1.struc2.array[23]`.

When the current programming language setting is C, the term *lvalue* is used in place of reference.

COBOL uses structure qualification (IN or OF keyword) and can have optional subscripting and substringing of the form:
`array OF struc2 OF struc1(subscript)(starting_position:length)`.

Particular rules for forming a reference depend on the current programming language setting and what release level of the language run-time library Debug Tool is running under. For example, if you upgrade your version of the HLL compiler without upgrading your version of Debug Tool, certain application programming interface inconsistencies might exist.

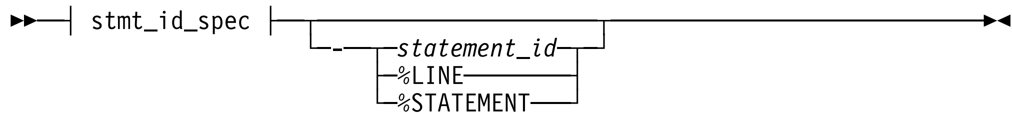
Statement_Id

A *statement_id* identifies an executable statement in a manner appropriate for the current programming language. This can be a statement number, sequence number, or source line number. The statement id is an *integer* or *integer.integer* (where the first *integer* is the line number and the second *integer* is the relative statement number). For example, you can specify `3`, `3.0`, or `3.1` to signify the first relative statement on line 3. C, COBOL, and PL/I allow multiple statements or verbs within a source line.

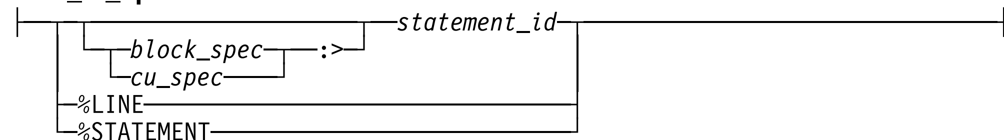
You can only use statement identifiers for statements that are known in the current enclave.

Statement_Id_Range and Stmt_Id_Spec

A *statement_id_range* identifies a source statement id or range of statement ids. *Stmt_id_spec* identifies a statement id specification.



stmt_id_spec:



block_spec
A valid block specification; see "Block_Spec" on page 200. The default is the currently qualified block.

Note: For the currently supported programming languages, block qualification is extraneous and will be ignored. This is because statement identifiers are unique within a compile unit.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. The default is the currently qualified compile unit.

statement_id
A valid statement identifier number; see "Statement_Id" on page 203.

%LINE
Represents the currently suspended source statement or line. See Table 4 on page 126. %LINE is equivalent to %STATEMENT.

%STATEMENT
Is equivalent to %LINE. See Table 4 on page 126.

Specifying a Range of Statements: A range of statements can be identified by specifying a beginning and ending statement id, separated by a hyphen (-). When the current programming language setting is COBOL, blanks are required around the hyphen (-). Blanks are optional for C and PL/I. Both statement ids must be in the same block, the second statement cannot occur before the first in the source program, and they cannot be equal.

A single statement id is also an acceptable statement id range and is considered to begin and end at the same statement. This consists of only one statement or verb even in a multistatement line.

Statement_Label

A *statement_label* identifies a statement using its source label. The specification of a qualified statement label includes all containing compile unit names or block names, and can optionally begin with a phase name qualifier. For example, `phs::>proc1:>proc2:>block1:>start`.

The form of a label depends on the current programming language:

- In C, labels must be valid identifiers.
- In COBOL, labels must be valid identifiers and can be qualified with the section name.
- In PL/I, labels must be valid identifiers, which can include a label variable.

You can only use statement labels for labels that are known in the current enclave.

Chapter 13. Debug Tool Commands

This chapter describes the syntax and usage of each Debug Tool command.

See "How to Read the Syntax Diagrams" on page xvi for an explanation of the syntax notation used to define the commands.

ANALYZE Command (PL/I)

The ANALYZE command displays the process of evaluating an expression and the data attributes of any intermediate results. To display the results of the expression, use the LIST command.

```
►►──ANALYZE──EXPRESSION──(──expression──)──;─────────────────────────►◄
```

EXPRESSION
Requests that the accompanying *expression* be evaluated from the following points of view:

- What are the attributes of each element during the evaluation of the expression?
- What are the dimensions and bounds of the elements of the expression, if applicable?
- What are the attributes of any intermediate results that will be created during the processing of the expression?

expression
A valid Debug Tool PL/I expression.

Usage Notes:

- If SET SCREEN ON is in effect, and you want to issue ANALYZE EXPRESSION for an expression in your program, you can bring the expression from the Source window up to the command line by typing over any character in the line that contains the expression. Then, edit the command line to form the desired ANALYZE EXPRESSION command.
- If SET WARNING ON is in effect, Debug Tool displays messages about PL/I computational conditions that may be raised when evaluating the expression. See "SET WARNING (C and PL/I)" on page 325 for specific information.
- Although the PL/I compiler supports the concatenation of GRAPHIC strings, Debug Tool does not.

© Copyright IBM Corp. 1995, 1996

Example:

This example is based on the following program segment:

```
DECLARE lo_point FIXED BINARY(31,5);
DECLARE hi_point FIXED BINARY(31,3);
DECLARE offset FIXED DECIMAL(12,2);
DECLARE percent CHARACTER(12);
lo_point = 5.4; hi_point = 28.13; offset = -6.77;
percent = '18';
```

The following is an example of the information prepared by issuing ANALYZE EXPRESSION. Specifically, the following shows the effect that mixed precisions and scales have on intermediate and final results of an expression:

```
ANALYZE EXPRESSION ( (hi_point - lo_point) + offset / percent )
>>> Expression Analysis <<<
 ( HI_POINT - LO_POINT ) + OFFSET / PERCENT
|   HI_POINT - LO_POINT
|  |   HI_POINT
|  |   FIXED BINARY(31,3) REAL
|  |   LO_POINT
|  |   FIXED BINARY(31,5) REAL
|   FIXED BINARY(31,5) REAL
|   OFFSET / PERCENT
|  |   OFFSET
|  |   FIXED DECIMAL(12,2) REAL
|  |   PERCENT
|  |   CHARACTER(12)
|   FIXED DECIMAL(15,5) REAL
 FIXED BINARY(31,17) REAL
```

Assignment Command (PL/I)

The Assignment command assigns the value of an expression to a specified reference.

```
►►──reference──=──expression──;──────────────────────────────────►◄
```

reference
A valid Debug Tool PL/I reference. See “References” on page 203.

expression
A valid Debug Tool PL/I expression.

Usage Notes:

- The PL/I repetition factor is not supported by Debug Tool.

  For example, the following is not valid: `rx = (16)'01'B;`

- If Debug Tool was invoked because of a computational condition or an attention interrupt, using an assignment to set a variable might not give the expected results. This is because Debug Tool cannot determine variable values within statements, only at statement boundaries.

- The PL/I assignment statement option BY NAME is not valid in the Debug Tool.

Examples:

- Assign the value 6 to variable x.

```
x = 6;
```

- Assign to the Debug Tool variable %GPR5 the address of name_table.

```
%GPR5 = ADDR (name_table);
```

- Assign to the prg_name variable the value of Debug Tool variable %PROGRAM.

```
prg_name = %PROGRAM;
```

AT Command

The AT command defines a breakpoint or a set of breakpoints. By defining breakpoints, you can temporarily suspend program execution and use Debug Tool to perform other tasks. By specifying an AT-condition, you instruct Debug Tool when to gain control. You can also specify in the AT command what action Debug Tool should take when the AT-condition occurs.

A breakpoint for the specified AT-condition remains established until either another AT command establishes a new action for the same AT-condition or a CLEAR command removes the established breakpoint. An informational message is issued when the first case occurs. Some breakpoints might become obsolete during a debug session and will be cleared automatically by Debug Tool. See the usage notes for more details.

The various forms of the AT command are summarized in Table 9.

Table 9 (Page 1 of 2). Summary of AT Commands

Command	Description
AT ALLOCATE	gives Debug Tool control when storage for a named controlled variable or aggregate is dynamically allocated by PL/I.
AT APPEARANCE	gives Debug Tool control: • For C and PL/I, when the specified compile unit is found in storage • For COBOL, the first time the specified compile unit is called
AT CALL	gives Debug Tool control on an attempt to call the specified entry point.
AT CHANGE	gives Debug Tool control when either the specified variable value or storage location is changed.
AT CURSOR	defines a statement breakpoint by cursor pointing.
AT DELETE	gives Debug Tool control when a phase is deleted.
AT ENTRY/EXIT	defines a breakpoint at the specified entry point or exit.
AT GLOBAL	gives Debug Tool control for every instance of the specified AT-condition.
AT LABEL	gives Debug Tool control at the specified statement label.
AT LINE	gives Debug Tool control at the specified line.
AT LOAD	gives Debug Tool control when the specified phase is loaded.
AT OCCURRENCE	gives Debug Tool control on a language or LE/VSE condition or exception.

Table 9 (Page 2 of 2). Summary of AT Commands	
AT PATH	gives Debug Tool control at a path point.
AT Prefix	defines a statement breakpoint via the Source window prefix area.
AT STATEMENT	gives Debug Tool control at the specified statement.
AT TERMINATION	gives Debug Tool control when the application program is terminated.

Usage Notes:

- To set breakpoints at specific locations in a program, Debug Tool depends on that program being loaded into storage. If you issue an AT command for a specific ENTRY, EXIT, LABEL, LINE, or STATEMENT breakpoint and the program is not known by Debug Tool, a warning message is issued and the breakpoint is not set.
- To set a global breakpoint, you can specify an asterisk (*) with the AT command or you can specify an AT GLOBAL command. For example, if you want to set a global AT ENTRY breakpoint, specify:

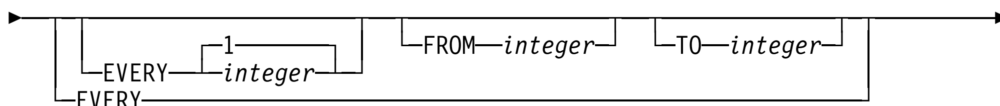
```
AT ENTRY *;
or
AT GLOBAL ENTRY;
```

- AT CHANGE, AT ENTRY, AT EXIT, AT LABEL, AT LINE, or AT STATEMENT breakpoints (when entered for a specific block, label, line, or statement) are automatically cleared when the containing compile unit is removed from storage.
- AT CHANGE breakpoints are automatically cleared when the containing blocks are no longer active or if the relevant variables are in dynamic storage that is freed by a language construct in the program (for example, a C call to `free()`).
- Clearing of a breakpoint is independent of whether the breakpoint is ENABLEd or DISABLEd.
- When multiple AT conditions are raised at the same statement or line, Debug Tool processes them in a predetermined order.

Every_Clause

Most forms of the AT command contain an optional *every_clause* that controls whether the specified action is taken based on the number of times a situation has occurred. For example, you might want an action to occur only every 10th time a breakpoint is reached.

The syntax for *every_clause* is:



EVERY *integer*
Specifies how frequently the breakpoint is taken. For example, EVERY 5 means that Debug Tool is invoked every fifth time the AT-condition is met. The default is EVERY 1.

FROM *integer*
Specifies when Debug Tool invocations are to begin. For example, FROM 8 means that Debug Tool is not invoked until the eighth time the AT-condition is met. If the FROM value is not specified, its value is equal to the EVERY value.

TO *integer*
Specifies when Debug Tool invocations are to end. For example, TO 20 means that after the 20th time this AT-condition is met, it should no longer invoke Debug Tool. If the TO value is not specified, the *every_clause* continues indefinitely.

Usage Notes:

- FROM *integer* cannot exceed TO *integer* and all integers must be ≥ 1.
- EVERY by itself is the same as EVERY 1 FROM 1.
- The EVERY, FROM, and TO clauses can be specified in any order.

Examples:

- Break every third time statement 50 is reached, beginning with the 48th time and ending after the 59th time. The breakpoint action is performed the 48th, 51st, 54th, and 57th time statement 50 is reached.

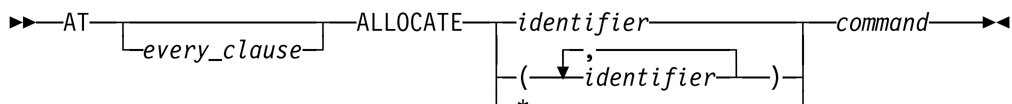
```
AT EVERY 3 FROM 48 TO 59 STATEMENT 50;
```

- At the fifth change of structure field `member` of the structure named `mystruct`, print a message saying that it has changed and list its new value. In addition, clear the CHANGE breakpoint. The current programming language setting is C.

```
AT FROM 5 CHANGE mystruct.member {
  LIST ("mystruct.member has changed.
            It is now", mystruct.member);
  CLEAR AT CHANGE mystruct.member;
}
```

AT ALLOCATE (PL/I)

AT ALLOCATE gives Debug Tool control when storage for a named controlled variable or aggregate is dynamically allocated by PL/I. When the AT ALLOCATE breakpoint occurs, the allocated storage has not yet been initialized; initialization, if any, occurs when control is returned to the program.



every_clause
As described under “Every_Clause” on page 209.

identifier
The name of a PL/I controlled variable whose allocation causes an invocation of Debug Tool. If the variable is the name of a structure, only the major structure name can be specified.

* Sets a breakpoint at every ALLOCATE.

command
: A valid Debug Tool command.

Examples:

- When the major structure `area_name` is allocated, display the address of the storage that was obtained.

```
AT ALLOCATE area_name LIST ADDR (area_name);
```

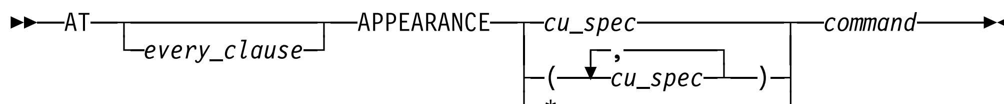
- List the changes to `temp` where the storage for `temp` has been allocated.

```
DECLARE temp CHAR(80) CONTROLLED INITIAL('abc');

AT ALLOCATE temp;
  BEGIN;
    AT CHANGE temp;
      BEGIN;
        LIST (temp);
        GO;
      END;
    GO;
  END;
GO;

temp = 'The first time.';
temp = 'The second time.';
temp = 'The second time.';
```

When `temp` is allocated the value of `temp` has not yet been initialized. When it is initialized to `'abc'` by the INITIAL phrase, the first AT CHANGE is recognized and `'abc'` is listed. The three assignments to `temp` cause the value to be set again but the third assignment doesn't change the value. This example results in one ALLOCATE breakpoint and three CHANGE breakpoints.

AT APPEARANCE

Gives Debug Tool control when the specified compile unit is found in storage. This is usually the result of a new phase being loaded. However, for phases with the main compile unit in COBOL, the breakpoint does not occur until the compile unit is first entered after being loaded.



every_clause
: As described under “Every_Clause” on page 209.

cu_spec
: A valid compile unit specification; see “CU_Spec” on page 201.

* Sets a breakpoint at every APPEARANCE of any compile unit.

command
: A valid Debug Tool command.

Usage Notes:

- In a CICS environment, if an AT APPEARANCE breakpoint is set for a program that is loaded via XCTL or LINK, the breakpoint will not be raised.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- If this breakpoint is set in a parent enclave it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- If the compile unit is qualified with a phase name, the AT APPEARANCE breakpoint will only be recognized for the compile unit that is contained in the specified phase. For example, if a compile unit `cux` which is in phase `phasey` appears, the breakpoint `AT APPEARANCE phasex::>cux` will not be triggered.
- If the compile unit is *not* qualified with a phase name, the current phase qualification is used.
- Debug Tool gains control when the specified compile unit is first recognized by Debug Tool. This can occur when a program is reached that contains a reference to that compile unit. This occurs late enough that the program can be operated on (setting breakpoints, for example), but early enough that the program has not yet been executed. In addition, for C, static variables can also be referenced.
- AT APPEARANCE is helpful when setting breakpoints in unknown compile units. You can set breakpoints at locations currently unknown to Debug Tool by using the proper qualification and embedding the breakpoints in the command list associated with an APPEARANCE breakpoint. However, there can be only one APPEARANCE breakpoint set at any time for a given compile unit and you must include all breakpoints for that unknown compile unit in a single APPEARANCE breakpoint.
- For C, AT APPEARANCE is not triggered for compile units that reside in a loaded phase since the compile units are known at the time of the load.
- For C and PL/I, an APPEARANCE breakpoint is triggered when Debug Tool finds the specified compile unit in storage. To be triggered, however, the APPEARANCE breakpoint must be set before the compile unit is loaded.

  At the time the APPEARANCE breakpoint is triggered, the compile unit you are monitoring has not become the currently-running compile unit. The compile unit that is current when the new compile unit appears in storage, triggering the APPEARANCE breakpoint, remains the current compile unit until execution passes to the new compile unit.
- For COBOL, an APPEARANCE breakpoint is triggered when Debug Tool finds the specified compile unit in storage. To be triggered, however, the APPEARANCE breakpoint must be set before the compile unit is called.

At the time the APPEARANCE breakpoint is triggered, the compile unit you are monitoring has not become the currently-running compile unit. The compile unit that is current when the new compile unit appears in storage, triggering the APPEARANCE breakpoint, remains the current compile unit until execution passes to the new compile unit.

Examples:

- Establish an entry breakpoint when compile unit cu is found in storage. The current programming language setting is C.

```
AT APPEARANCE cu {
  AT ENTRY a;
  GO;
}
```

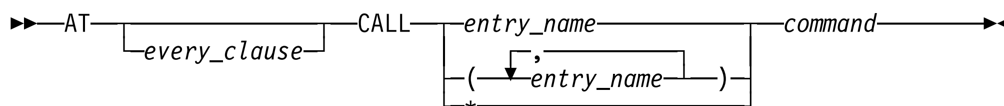
- Defer the AT EXIT and AT LABEL breakpoints until compile unit cuy is first entered after being loaded into storage. The current programming language setting is COBOL.

```
AT APPEARANCE cuy PERFORM
  AT EXIT cuy:>blocky LIST ('Exiting blocky.');
  AT LABEL cuy:>lab1 QUERY LOCATION;
END-PERFORM;
```

  If cuy is later deleted from storage, the breakpoints that are dependent on cuy are automatically cleared. However, if cuy is then loaded again, the APPEARANCE breakpoint for cuy is triggered and the AT EXIT and AT LABEL breakpoints are redefined.

AT CALL

Gives Debug Tool control when the application code attempts to call the specified entry point. Using CALL breakpoints, you can simulate the execution of unfinished subroutines, create dummy or *stub* programs, or set variables to mimic resultant values, allowing you to test sections of code before the whole is complete.



every_clause
: As described under “Every_Clause” on page 209.

entry_name
: A valid external entry point name constant or zero (0); however, 0 can only be specified if the current programming language setting is C or PL/I.

*
: Sets a breakpoint at every CALL of any entry point.

command
: A valid Debug Tool command.

Usage Notes:

- AT CALL intercepts the call itself, not the subroutine entry point. C, COBOL, and PL/I programs compiled with the compile-time TEST(PATH) option identify call targets even if they are unresolved. For more information on the compile-time TEST option, see:

  "Compiling a C Program with the Compile-Time TEST Option" on page 12,

  "Compiling a COBOL Program with the Compile-Time TEST Option" on page 16, or

  "Compiling a PL/I Program with the Compile-Time TEST Option" on page 19.

- A breakpoint set with AT CALL for a call to a C, or PL/I built-in function is never triggered.
- CALL statements within an INITIAL attribute on a PL/I variable declaration will not trigger AT CALL breakpoints.
- AT CALL generally intercepts only calls to entry points known to Debug Tool at compile time. Calls to entry variables are not intercepted, except when the current programming language setting is either C or COBOL (compiled with the run-time TEST option).
- AT CALL `0` intercepts calls to unresolved entry points when the current programming language setting is C or PL/I (compiled with the run-time TEST option).
- AT CALL allows you to intercept or bypass the target program by using GO BYPASS or GOTO. If resumed by a normal GO or STEP, execution resumes by performing the call.
- If this breakpoint is set in a parent enclave it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- For a CICS application on Debug Tool, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application on Debug Tool, it is cleared at the end of a process.
- For COBOL, remember to enclose the entry_name in double (") or single (') quotes if it is case sensitive.
- To be able to set CALL breakpoints in C, you must compile your program with either the PATH or ALL suboption of the compile-time TEST option. The default is PATH.
- If your C program has unresolved entry points or entry variables, issue AT CALL 0.
- To be able to set CALL breakpoints in COBOL, you must compile your program with either the PATH or ALL suboption of the compile-time TEST option.

  AT CALL 0 is not supported for use with COBOL programs. However, COBOL is able to identify CALL targets even if they are unresolved, and also identify entry variables and intercept them. Therefore, not all external references need be resolved for COBOL programs.

- To be able to set CALL breakpoints in PL/I, you must compile your program with either the PATH or ALL suboptions of the compile-time TEST option. AT CALL 0 is supported and is invoked for unresolved external references.

Examples:

- Intercept all calls and request input from the terminal.

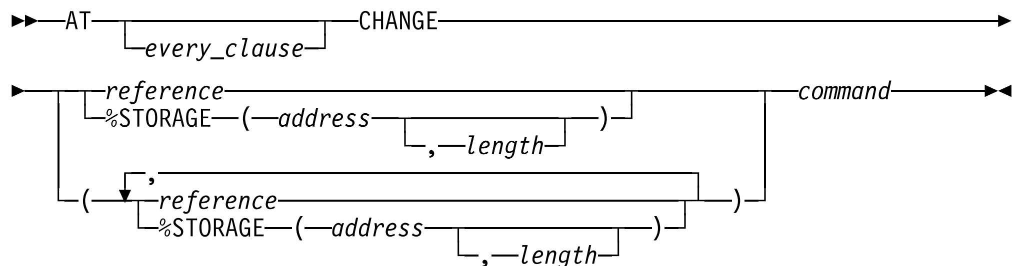
```
AT CALL *;
```

- If the program invokes function `badsubr`, intercept the call, set variable `varbl` to 50, and then bypass the target function. The current programming language setting is C.

```
AT CALL badsubr {
  varbl = 50;
  GO BYPASS;
}
```

AT CHANGE

Gives Debug Tool control when either the application program or Debug Tool command changes the specified variable value or storage location.



every_clause
: As described under "Every_Clause" on page 209.

reference
: A valid Debug Tool reference in the current programming language; see "References" on page 203.

%STORAGE
: A built-in function that provides an alternative way to select an AT CHANGE subject.

address
: The starting address of storage to be watched for changes. This must be a hex constant:

- *0x* in C
- *H* in COBOL (using either double (") or single (') quotes)
- A PX constant in PL/I.

length
: The number of bytes of storage being watched for changes. This must be a positive integer constant. The default value is 1.

command
A valid Debug Tool command.

Usage Notes:

- Data is watched only in storage; hence a value that is being kept in a register due to compiler optimization cannot be watched. In addition, the Debug Tool variables %GPRn, %FPRn, %LPRn, and %EPRn cannot be watched.
- Only entire bytes are watched; bits or bit strings within a byte cannot be singled out.
- Since AT CHANGE breakpoints are identified by storage address and length, it is not possible to have two AT CHANGE breakpoints for the same area (address and length) of storage. That is, an AT CHANGE command replaces a previous AT CHANGE command if the storage address and length are the same. However, any other overlap is ignored and the breakpoints are considered to be for two separate variables. For example, if the storage address is the same, but the length is different, the AT CHANGE command *will not* replace the previous AT CHANGE.
- When more than one AT CHANGE breakpoint is triggered at a time, AT CHANGE breakpoints will be triggered in the order that they were originally set. However, if the triggering of one breakpoint causes a variable watched by a different breakpoint to change, the ordering of the triggers will not necessarily be according to when they were originally entered. For example:

  ```
  AT CHANGE y LIST y;
  AT CHANGE x y = 4;
  GO;
  ```

  If the next statement to be executed in your program causes the value of x to change, the CHANGE x breakpoint will be triggered when Debug Tool gains control. Processing of CHANGE x causes the value of y to change. If you type "GO;" after being informed that CHANGE x was triggered, Debug Tool will trigger the CHANGE y breakpoint (before returning control to your program).

  In this case, the CHANGE y breakpoint was set first, but the CHANGE x breakpoint was triggered first (it caused the CHANGE y breakpoint to be triggered).
- %STORAGE is a Debug Tool built-in function that is available only in the CHANGE breakpoint commands.
- For a CICS application, the CHANGE %STORAGE breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- The referenced variables must exist when the AT CHANGE breakpoint is defined. One way to ensure this is to embed the AT CHANGE in an AT ENTRY.
- An AT CHANGE breakpoint gets removed automatically when the specified variable is no longer defined. AT CHANGEs for C static variables are removed when the phase defining the variable is removed

from storage. For C storage that is allocated using `malloc()` or `calloc()`, this occurs when the dynamic storage is freed using `free()`.

- Changes are not detected immediately, but only at the completion of any command that has the potential of changing storage or variable values. If you issue a Debug Tool command that modifies a variable being watched, the CHANGE condition is raised immediately. You can force more or less frequent checking by using the SET CHANGE command.
- C AT CHANGE breakpoint requirements
  - The variable must be an `lvalue` or an array.
  - The variable must be declared in an active block if the variable is a parameter or has a storage class of `auto`.
  - If you specify the address of the storage containing the variable, it must be specified with a hexadecimal constant.
  - A CHANGE breakpoint defined for a static variable is automatically removed when the file in which the variable was declared is no longer active. A CHANGE breakpoint defined for an external variable is automatically removed when the phase where the variable was declared is no longer active.
- COBOL AT CHANGE breakpoint requirements
  - AT CHANGE using a storage address should not reference a data item that follows a variable-size element or subgroup within a group. COBOL dynamically remaps the group when a variable-size element changes size.
  - If you specify the address of the storage containing the variable, it must be with an `H` constant, delimited by either quotation marks or apostrophes. The `H` constant can only be used where an address or POINTER variable can be used. The COBOL hexadecimal notations for nonnumeric literals should be used for all other situations. For details on the `H` constant, see “Using Constants in Expressions” on page 173.
  - Be careful when examining a variable whose allocated storage follows that of a variable-size element. COBOL dynamically remaps the storage for the element any time it changes size. This could alter the address of the variable you want to examine.
  - You cannot set a CHANGE breakpoint for a COBOL file record before the file is opened.
  - The variable, when in the local storage section, must be declared in an active block.
- PL/I AT CHANGE breakpoint requirements
  - CHANGE breakpoint is removed for based or controlled variables when they are FREEd and for parameters and AUTOMATIC variables when the block in which they are declared is no longer active.
  - CHANGE monitors only structures with single scalar elements. Structures containing more than one scalar element are not supported.

- The variable must be a valid reference for the current block.
- The breakpoint is automatically removed after the referenced variable ceases to exist. The CHANGE breakpoint is removed for based or controlled variables when they are FREEd and for parameters and AUTOMATIC variables when the block in which they were declared is no longer active.
- A CHANGE breakpoint monitors the storage allocated to the current generation of a controlled variable. If you subsequently allocate new generations, they are not automatically monitored.
- If you specify the address of storage containing the variable, you must do so with a PX constant, delimited by single quotation marks. The PX constant can only be used where an address or pointer variable can be used.

Examples:

- Identify the current location each time variable `varbl1` or `varbl2` is found to have a changed value. The current programming language setting is COBOL.

```
AT CHANGE (varbl1, varbl2) PERFORM
  QUERY LOCATION;
  GO;
END-PERFORM;
```

- When storage at the hex address 22222 changes, print a message in the log. Eight bytes of storage are to be watched. The current programming language setting is C.

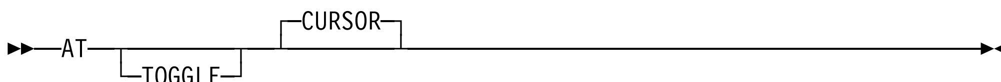
```
AT CHANGE %STORAGE (0x00022222, 8)
  LIST "Storage has changed at hex address 22222";
```

- Set two breakpoints when storage at the hex address 1000 changes. The variable `x` is defined at hex address 1000 and is 20 bytes in length. In the first breakpoint, 20 bytes of storage are to be watched. In the second breakpoint, 50 bytes of storage are to be watched. The current programming language setting is C.

```
AT CHANGE %STORAGE (0x00001000, 20)     /* Breakpoint 1 set */
AT CHANGE %STORAGE (0x00001000, 50)     /* Breakpoint 2 set */
AT CHANGE x       /* Replaces breakpoint 1, since x is at  */
                  /* hex address 1000 and is 20 bytes long */
```

AT CURSOR (Full-Screen Mode)

Provides a cursor controlled method for setting a statement breakpoint. It is most useful when assigned to a PF key.



TOGGLE
Specifies that if the cursor-selected statement already has an associated statement breakpoint then the breakpoint is removed rather than replaced.

Usage Notes:

- AT CURSOR does not allow specification of an *every_clause* or a *command*, and must not have a semicolon coded.
- The cursor must be in the Source window and positioned on a line where an executable statement begins. An AT STATEMENT command for the first executable statement in the line is generated and executed (or cleared if one is already defined and TOGGLE is specified).

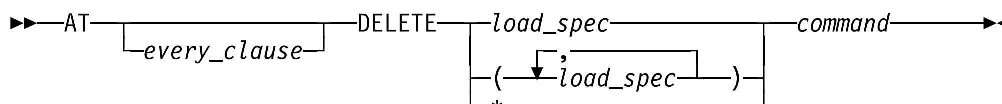
Example:

Define a PF key to toggle the breakpoint setting at the cursor position.

```
SET PF10 = AT TOGGLE CURSOR;
```

AT DELETE

Gives Debug Tool control when a phase is removed from storage by an LE/VSE delete service, such as on completion of a successful C `release()`, COBOL CANCEL, or PL/I RELEASE.



every_clause
: As described under “Every_Clause” on page 209.

load_spec
: A valid phase specification; see “Load_Spec” on page 203.

*
: Sets a breakpoint at every DELETE of any phase.

command
: A valid Debug Tool command.

Usage Notes:

- Debug Tool gains control for deletes that are affected by the LE/VSE delete service. If a phase is deleted using the VSE CDDELETE macro, Debug Tool is not informed. This can cause errors if Debug Tool attempts to operate on any part of the deleted phase.
- AT DELETE cannot specify the initial phase.
- If this breakpoint is set in a parent enclave it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.

Examples:

- Each time a phase is deleted, request input from the terminal.

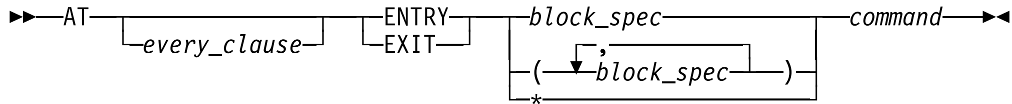
```
AT DELETE *;
```

- Stop watching variable `var1:>x` when phase `myphs` is deleted.

```
AT DELETE myphs CLEAR AT CHANGE (var1:>x);
```

AT ENTRY/EXIT

Defines a breakpoint at the specified entry point or exit in the specified block.



every_clause
As described under "Every_Clause" on page 209.

block_spec
A valid block specification; see "Block_Spec" on page 200.

* Sets a breakpoint at every ENTRY or EXIT of any block.

command
A valid Debug Tool command.

Usage Notes:

- AT ENTRY/EXIT can only be set for programs that are currently fetched or loaded. If you want to set an entry or exit breakpoint for a currently unknown compile unit, see "AT APPEARANCE" on page 211.
- An ENTRY or EXIT breakpoint set for a compile unit that becomes nonactive (one that is not in the current enclave), is suspended until the compile unit becomes active. An ENTRY/EXIT breakpoint set for a compile unit that is deleted from storage is suspended until the compile unit is restored. A suspended breakpoint cannot be triggered or operated on with breakpoint commands.
- For a CICS application running with Debug Tool, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- Both ENTRY and EXIT breakpoints for blocks in a fetched or loaded program are removed when that program is released.

Examples:

- At the entry of program `subrx`, initialize variable `ix` and continue program execution. The current programming language setting is COBOL.

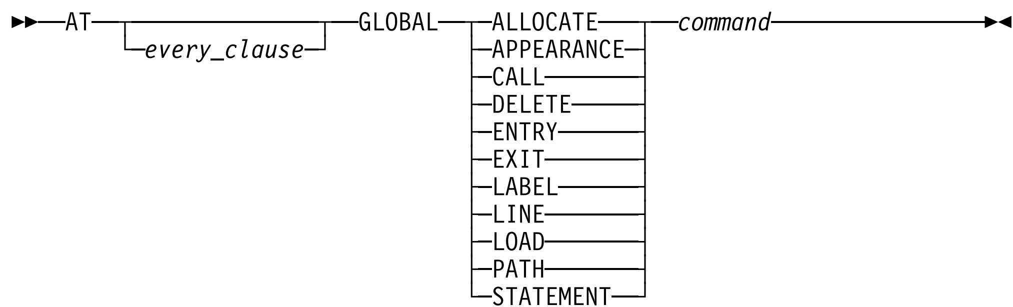
```
AT ENTRY subrx PERFORM
  SET ix TO 5;
  GO;
END-PERFORM;
```

- At exit of `main` print a message and trigger the SIGUSR1 condition. The current programming language setting is C.

```
AT EXIT main {
  puts("At exit of the program");
  TRIGGER SIGUSR1;
  GO;
}
```

AT GLOBAL

Gives Debug Tool control for every instance of the specified AT-condition. These breakpoints are independent of their nonglobal counterparts (except for AT PATH, which is identical to AT GLOBAL PATH). Global breakpoints are always performed before their specific counterparts.



every_clause
: As described under “Every_Clause” on page 209.

command
: A valid Debug Tool command.

You should use GLOBAL breakpoints where you don't have specific information of where to set your breakpoint. For example, you want to halt at entry to block `Abcdefg_Unknwn` but cannot remember the name, you can issue `AT GLOBAL ENTRY` and Debug Tool will halt every time a block is being entered. If you want to halt at every function call, you can issue `AT GLOBAL CALL`.

Usage Notes:

- To set a global breakpoint, you can specify an asterisk (*) with the AT command or you can specify an AT GLOBAL command.
- Although you can define GLOBAL breakpoints to coexist with singular breakpoints of the same type at the same location or event, COBOL does not allow you to define two or more single breakpoints of the same type for the same location or event. The last breakpoint you define replaces any previous breakpoint.

Examples:

- If you want to set a global AT ENTRY breakpoint, specify:

```
AT ENTRY *;
or
AT GLOBAL ENTRY;
```

- At every statement or line, display a message identifying the statement or line. The current programming language setting is COBOL.

```
AT GLOBAL STATEMENT LIST ('At Statement:', %STATEMENT);
```

- If you enter (for COBOL):

```
AT EXIT table1 PERFORM
LIST TITLED (age, pay);
GO;
END-PERFORM;
```

then enter:

```
AT EXIT table1 PERFORM
LIST TITLED (benefits, scale);
GO;
END-PERFORM;
```

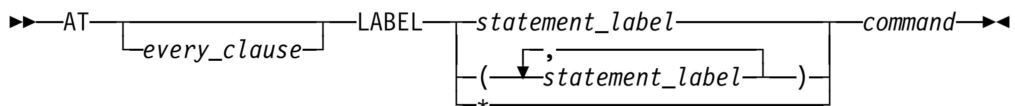
only `benefits` and `scale` are listed when your program reaches the exit point of block `table1`. The second AT EXIT replaces the first because the breakpoints are defined for the same location. However, if you define the following GLOBAL breakpoint:

```
AT GLOBAL EXIT PERFORM
LIST TITLED (benefits, scale);
GO;
END-PERFORM;
```

in conjunction with the first EXIT breakpoint, when your program reaches the exit from `table1`, all four variables (`age`, `pay`, `benefits`, and `scale`) are listed with their values, because the GLOBAL EXIT breakpoint can coexist with the EXIT breakpoint set for `table1`.

AT LABEL

Gives Debug Tool control when execution has reached the specified statement label or group of labels. For C and PL/I, if there are multiple labels associated with a single statement, you can specify several labels and Debug Tool gains control at each label. For COBOL, AT LABEL lets you specify several labels, but for any group of labels that are associated with a single statement, Debug Tool gains control for that statement only once.



every_clause
: As described under “Every_Clause” on page 209.

statement_label
: a valid source label constant; see “Statement_Label” on page 205.

*   Sets a breakpoint at every LABEL.

command
A valid Debug Tool command.

Usage Notes:

- For COBOL *statement_label* can have either of the following forms:
  - *name*

    This form can be used in COBOL for reference to a section name or for a COBOL paragraph name that is not within a section or is in only one section of the block.
  - *name1* OF *name2* or *name1* IN *name2*

    This form must be used for any reference to a COBOL paragraph (name1) that is within a section (name2), if the same name also exists in other sections in the same block. You can specify either OF or IN, but Debug Tool always uses OF for output.

  Either form can be prefixed with the usual block, compile unit, and phase name qualifiers.
- For C or PL/I, you can set a LABEL breakpoint at each label located at a statement. This is the only circumstance where you can set more than one breakpoint at the same location.
- A LABEL breakpoint set for a nonactive compile unit (one that is not in the current enclave), is *suspended* until the compile unit becomes active. A LABEL breakpoint set for a compile unit that is deleted from storage is suspended until the compile unit is restored. A suspended breakpoint cannot be triggered or operated on with breakpoint commands.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- You cannot set LABEL breakpoints at, for example, PL/I label variables.
- LABEL breakpoints for label constants in a fetched or loaded phase are removed when that program is released.
- To be able to set LABEL breakpoints in C or PL/I, you must compile your program with either the PATH and SYM suboptions or the ALL suboption of the compile-time TEST option.

  You can set breakpoints for more than one label at the same location. Debug Tool is entered for each specified label.
- To be able to set LABEL breakpoints in COBOL, you must compile your program with either the STMT, PATH, or ALL suboption and the SYM suboption of the compile-time TEST option.

  When defining specific LABEL breakpoints Debug Tool sets a breakpoint for each label specified, unless there are several labels on the same statement. In this case, only the last LABEL breakpoint defined is set.

Examples:

- Set a breakpoint at label `create` in the currently qualified block.

```
AT LABEL create;
```

- At program label `para OF sect1` display variable names `x` and `y` and their values, and continue program execution. The current programming language setting is COBOL.

```
AT LABEL para OF sect1 PERFORM
  LIST TITLED (x, y);
  GO;
END-PERFORM;
```

- Set a breakpoint at labels `label1` and `label2`, even though both labels are associated to the same statement. The current programming language setting is C.

```
AT LABEL label1 LIST 'Stopped at label1';  /* Label1 is first  */
AT LABEL label2 LIST 'Stopped at label2';  /* Label2 is second */
```

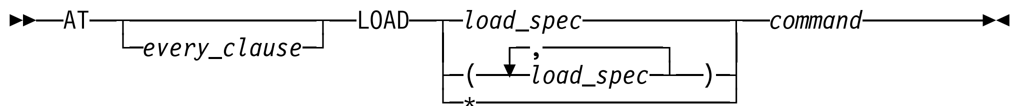
AT LINE

See “AT STATEMENT” on page 230.

AT LOAD

Gives Debug Tool control when the specified phase is brought into storage. For example, on completion of a successful C `fetch()`, a PL/I FETCH, or during a COBOL dynamic CALL. Once the breakpoint is raised for the specified phase, it is not raised again unless either the phase is released and fetched again or another phase with the specified name is fetched.

You can set LOAD breakpoints regardless of what compile-time options are in effect.



every_clause
: As described under “Every_Clause” on page 209.

load_spec
: A valid phase specification; see “Load_Spec” on page 203.

* Sets a breakpoint at every LOAD of any phase.

command
: A valid Debug Tool command.

Usage Notes:

- Debug Tool gains control for loads that are affected by the LE/VSE load service. If a phase is loaded using the VSE CDLOAD macro or EXEC CICS LOAD, Debug Tool is not informed.
- AT LOAD can be used to detect the loading of specific language library phases; however, the loading of language library phases does not trigger an AT GLOBAL LOAD or AT LOAD *.
- AT LOAD cannot specify the initial phase because it is already loaded when Debug Tool is invoked.
- A LOAD breakpoint is triggered when a new enclave is entered.
- If this breakpoint is set in a parent enclave it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- For a CICS application executing with Debug Tool, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- At the triggering of a LOAD breakpoint for C and PL/I, Debug Tool has enough information about the loaded phase to set breakpoints and examine variables of static and extern storage classes.
- At the triggering of a LOAD breakpoint for COBOL, Debug Tool does not have enough information about the loaded phase to set breakpoints in blocks contained within the phase. At the triggering of an APPEARANCE breakpoint, however, you can set such breakpoints.

Examples:

- Print a message when phase `myphs` is loaded. The current programming language setting is either C or COBOL.

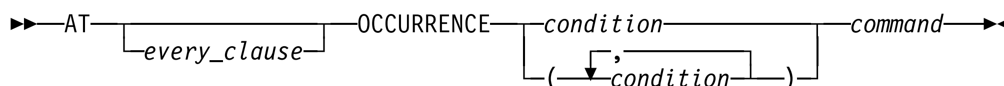
```
AT LOAD myphs LIST ("Phase myphs has been loaded");
```

- Establish an entry breakpoint when phase `a` is fetched and then resume execution. The current programming language setting is C.

```
AT LOAD a {
  AT ENTRY a;
  GO;
}
```

AT OCCURRENCE

Gives Debug Tool control on a language or LE/VSE condition or exception.



every_clause
As described under "Every_Clause" on page 209.

condition
A valid condition or exception. This can be either an LE/VSE symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.

Following are the C condition constants; they must be uppercase and not abbreviated:

SIGABND	SIGINT	SIGTERM
SIGABRT	SIGIOERR	SIGUSR1
SIGFPE	SIGSEGV	SIGUSR2
SIGILL		

PL/I condition constants can be used as well. See “ON Command (PL/I)” on page 287 for information about valid condition names.

There are no COBOL condition constants. Instead, an LE/VSE symbolic feedback code must be used, for example, CEE347. For symbolic feedback codes for LE/VSE callable services, see *LE/VSE Programming Reference*.

command
A valid Debug Tool command.

Program conditions and condition handling vary from language to language. The methods the OCCURRENCE breakpoint uses to adapt to each language are described below.

For C:

When a C or LE/VSE condition occurs during your session, the following series of events takes place:

1. Debug Tool is invoked before any C signal handler.
2. If you set an OCCURRENCE breakpoint for that condition, Debug Tool processes that breakpoint and executes any commands you have specified. If you did not set an OCCURRENCE breakpoint for that condition, and:
   - If the current test-level setting is ALL, Debug Tool prompts you for commands or reads them from a commands file.
   - If the current test-level setting is ERROR, and the condition has an error severity level (that is, anything but SIGUSR1, SIGUSR2, SIGINT, or SIGTERM), Debug Tool gets commands by prompting you or by reading from a commands file.
   - If the current test-level setting is NONE, Debug Tool ignores the condition and returns control to the program.

You can set OCCURRENCE breakpoints for equivalent C signals and LE/VSE conditions. For example, you can set AT OCCURRENCE CEE345 and AT OCCURRENCE SIGSEGV during the same debugging session. Both indicate an addressing exception and, if you set both breakpoints, no error occurs. However, if you set OCCURRENCE breakpoints for a condition using both its C and LE/VSE designations, the LE/VSE breakpoint is the only breakpoint triggered. Any command list associated with the C condition is not executed. Table 19 on page 347 lists the LE/VSE conditions and their C equivalents. Also see *LE/VSE Programming Guide*.

You can use OCCURRENCE breakpoints to control your program's response to errors.

Usage Notes:

- If the application program also has established an exception handler for the condition then that handler is entered when Debug Tool releases control, unless return is by use of GO BYPASS or GOTO or a specific statement.
- OCCURRENCE breakpoints for COBOL IGZ conditions can only be set after a COBOL run-time phase has been initialized.
- For C and PL/I, certain LE/VSE conditions map to C SIGxxx values and PL/I condition constants. It is possible to enter two AT OCCURRENCE breakpoints for the same condition. For example, one could be entered with the LE/VSE condition name and the other could be entered with the C SIGxxx condition constant. In this case, the AT OCCURRENCE breakpoint for the LE/VSE condition name is triggered and the AT OCCURRENCE breakpoint for the C condition constant is not. However, if an AT OCCURRENCE breakpoint for the LE/VSE condition name is not defined, the corresponding mapped C or PL/I condition constant is triggered.
- If this breakpoint is set in a parent enclave it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- For COBOL, Debug Tool detects LE/VSE conditions. If an LE/VSE condition occurs during your session, the following series of events takes place:
  1. Debug Tool is invoked before any condition handler.
  2. If you set an OCCURRENCE breakpoint for that condition, Debug Tool processes that breakpoint and executes any commands you have specified. If you have not set an OCCURRENCE breakpoint for that condition, and:
     - If the current test-level setting is ALL, Debug Tool prompts you for commands or reads them from a commands file.
     - If the current test-level setting is ERROR, and the condition has a severity level of 2 or higher, Debug Tool gets commands by prompting you or by reading from a commands file.
     - If the current test-level setting is NONE, Debug Tool ignores the condition and returns control to the program.

  You can use OCCURRENCE breakpoints to control your program's response to errors.

  See *LE/VSE Debugging Guide and Run-Time Messages* for a list of LE/VSE conditions.
- For PL/I, Debug Tool detects LE/VSE and PL/I conditions. If a condition occurs, Debug Tool is invoked before any condition handler. If you have issued an ON command or set an OCCURRENCE breakpoint for the specified condition, Debug Tool runs the associated commands. See “ON Command (PL/I)” on page 287.

If there is no AT OCCURRENCE or ON set, then:

- If the current test-level setting is ALL, Debug Tool prompts you for commands or reads them from a commands file.
- If the current test-level setting is ERROR, and the condition has an error severity level of 2 or higher, Debug Tool gets commands by prompting you or by reading from a commands file.
- If the current test-level setting is NONE, Debug Tool ignores the condition and returns control to the program.

Once Debug Tool returns control to the program, any relevant PL/I ON-unit is run. PL/I condition handling is described in *IBM PL/I for VSE/ESA Language Reference*. Also see *LE/VSE Programming Guide*.

Examples:

- When a data exception occurs, query the current location. The current programming language setting is either C or COBOL.

```
AT OCCURRENCE CEE347 QUERY LOCATION;
```

- When the SIGSEGV condition is raised, set an error flag and call a user termination routine. The current programming language setting is C.

```
AT OCCURRENCE SIGSEGV {
  error = 1;
  terminate (error);
}
```

- Suppose SIGFPE maps to CEE347 and the following breakpoints are defined. The current programming language setting is C.

```
AT OCCURRENCE SIGFPE LIST "SIGFPE condition";
AT OCCURRENCE CEE347 LIST "CEE347 condition";
```

If the LE/VSE condition CEE347 is raised, the CEE347 breakpoint is triggered.

However, if a breakpoint had not been defined for CEE347 and the CEE347 condition is raised, the SIGFPE breakpoint is triggered (since it is mapped to CEE347).

AT PATH

Gives Debug Tool control when the flow of control changes (at a path point). AT PATH is identical to AT GLOBAL PATH.

every_clause
As described under “Every_Clause” on page 209.

command
A valid Debug Tool command.

Usage Notes:

- For an explanation of path points and possible values for %PATHCODE, which vary according to the language of your program, see "Using Debug Tool Variables in C" on page 140 "Using Debug Tool Variables in COBOL" on page 167, or "Using Debug Tool Variables in PL/I" on page 181.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- For C, COBOL and PL/I, you can set PATH breakpoints if you compiled with the PATH suboption. For more information, see Chapter 2, "Preparing to Debug Your Program" on page 12.
- For COBOL and PL/I, you can set PATH breakpoints at any time (default is PATH), but setting of other breakpoints is different for each suboption of the compile-time TEST option. For more information, see "Compiling a COBOL Program with the Compile-Time TEST Option" on page 16 or "Compiling a PL/I Program with the Compile-Time TEST Option" on page 19.

Examples:

- Whenever a path point has been reached, display the five most recently processed breakpoints and conditions.

```
AT PATH LIST LAST 5 HISTORY;
```

- Whenever a path point has been reached, display a message and query the current location. The current programming language setting is COBOL.

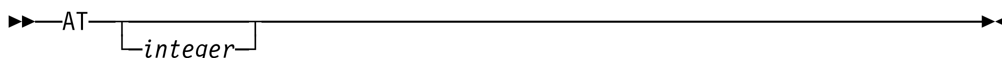
```
AT PATH PERFORM
  LIST "Path point reached";
  QUERY LOCATION;
  GO;
END-PERFORM;
```

- Whenever a path point has been reached, the value of %PATHCODE contains the code representing the type of path point stopped at. If the program is stopped at the entry to a block, display the %PATHCODE.

```
AT PATH LIST %PATHCODE;
```

AT Prefix (Full-Screen Mode)

Sets a statement breakpoint when you issue this command via the Source window prefix area. When one or more breakpoints have been set on a line, the prefix area for that line is highlighted.



integer
Selects a relative statement (for C and PL/I) or a relative verb (for COBOL) within the line. The default value is 1.

Example:

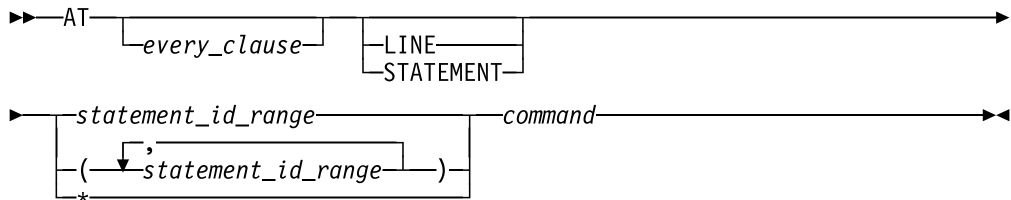
Set a breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
AT 3
```

No space is needed as a delimiter between the keyword and the integer; hence, AT 3 is equivalent to AT3.

AT STATEMENT

Gives Debug Tool control at each specified statement or line within the given set of ranges.



every_clause
: As described under "Every_Clause" on page 209.

statement_id_range
: A valid statement id or statement id range; see "Statement_Id_Range and Stmt_Id_Spec" on page 204.

*****
: Sets a breakpoint at every STATEMENT or LINE.

command
: A valid Debug Tool command.

Usage Notes:

- A STATEMENT breakpoint set for a nonactive compile unit (one that is not in the current enclave), is *suspended* until the compile unit becomes active. A STATEMENT breakpoint set for a compile unit that is deleted from storage is suspended until the compile unit is restored. A suspended breakpoint cannot be triggered or operated on with breakpoint commands.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.
- You can specify the first relative statement on each line in any one of three ways. If, for example, you want to set a STATEMENT breakpoint at the first relative statement on line three, you can enter `AT 3`, `AT 3.0`, or `AT 3.1`. However, Debug Tool logs them differently according to the current programming language as follows:
  - **For C**

    The first relative statement on a line is specified with "0". All of the above breakpoints are logged as `AT 3.0`.

– **For COBOL or PL/I**

The first relative statement on a line is specified with "1". All of the above breakpoints are logged as `AT 3.1`.

Examples:

- Set a breakpoint at statement or line number 23. The current programming language setting is COBOL.

```
AT 23 LIST 'About to close the file';
```

- Set breakpoints at statements 5 through 9 of compile unit `mycu`. The current programming language setting is C.

```
AT STATEMENT "mycu":>5 - 9;
```

- Set breakpoints at lines 19 through 23 and at statements 27 and 31.

```
AT LINE (19 - 23, 27, 31);
```

or

```
AT LINE (27, 31, 19 - 23);
```

AT TERMINATION

Gives Debug Tool control when the application program is terminated.

```
►►──AT──TERMINATION──command────────────────────────────────────►◄
```

command
A valid Debug Tool command.

Usage Notes:

- AT TERMINATION does not allow specification of an *every_clause* because termination can only occur once.
- If Debug Tool has been initialized for any reason, the following default form of this command is automatically in effect:

```
AT TERMINATION;
```

This definition causes control to be given to your terminal (or primary commands file) when the program ends. This termination breakpoint can be replaced or cleared at any time with the AT TERMINATION or CLEAR AT TERMINATION command.

- If this breakpoint is set in a parent enclave, it can be triggered and operated on with breakpoint commands while the application is in a child enclave.
- When Debug Tool gains control, normal execution of the program is complete; however, a CALL or function invocation from Debug Tool can continue to perform program code. When the AT TERMINATION breakpoint gives control to Debug Tool:
  - Fetched phases have not been released
  - Files have not been closed
  - Language-specific termination has been invoked yet no action has been taken

  In C, the user `atexit()` lists have already been called.

In PL/I, the FINISH condition was already raised.

- You are allowed to enter any command with AT TERMINATION. However, normal error messages are issued for any command that cannot be completed successfully because of lack of information about your program.
- The TERMINATION breakpoint is set automatically at Debug Tool initialization. It remains in effect for the entire Debug Tool session. Changes made to this breakpoint in one enclave will remain in effect when control is passed to another enclave.
- You can enter `DISABLE AT TERMINATION;` or `CLEAR AT TERMINATION;` at any time to disable or clear the breakpoint. It remains disabled or cleared until you re-enable or reset it.
- For a CICS application, this breakpoint is cleared at the end of the last process in the application. For a non-CICS application, it is cleared at the end of a process.

Examples:

- When the program ends, check the Debug Tool environment to see what files have not been closed.

```
AT TERMINATION DESCRIBE ENVIRONMENT;
```

- When the program ends, display the message "Program has ended" and end the Debug Tool session. The current programming language setting is C.

```
AT TERMINATION {
  LIST "Program has ended";
  QUIT;
}
```

BEGIN Command (PL/I)

BEGIN and END delimit a sequence of one or more commands to form one longer command. The BEGIN and END keywords cannot be abbreviated.

```
                ┌─────────┐
►►──BEGIN──;────▼─command─┴──END──;──────────────────────────►◄
```

command
A valid Debug Tool command.

Usage Notes:

- The BEGIN command is most helpful when used in AT, IF, or ON commands.
- The BEGIN command does not imply a new block or name scope. It is equivalent to a PL/I simple DO.

Examples:

- Set a breakpoint at statement 320 listing the value of variable x and assigning the value of 2 to variable a.

```
AT 320 BEGIN;
  LIST (x);
  a = 2;
END;
```

- When the PL/I condition FIXEDOVERFLOW is raised—that is, when the length of the result of a fixed-point arithmetic operation exceeds the maximum length allowed—list the value of variable x and assign the value of 2 to variable a. The current programming language setting is PL/I.

```
ON FIXEDOVERFLOW BEGIN; LIST (x); a=2; END;
```

block Command (C)

The block command allows you to group any number of Debug Tool commands into one command. When you enclose Debug Tool commands within a single set of braces ({}), everything within the braces is treated as a single command. You can place a block anywhere a command is allowed.

command
: A valid Debug Tool command.

Usage Notes:

- Declarations are not allowed within a nested block.
- The C block command does not end with a semicolon. A semicolon after the closing brace is treated as a Null command.

Example:

Establish an entry breakpoint when phase a is fetched.

```
AT LOAD a {
  AT ENTRY a;
  GO;
}
```

break Command (C)

The break command allows you to terminate and exit a loop (that is, do, for, and while) or switch command from any point other than the logical end. You can place a break command only in the body of a looping command or in the body of a switch command. The break keyword must be lowercase and cannot be abbreviated.

In a looping statement, the `break` command ends the loop and moves control to the next command outside the loop. Within nested statements, the `break` command ends only the smallest enclosing `do`, `for`, `switch`, or `while` commands.

In a `switch` body, the `break` command ends the execution of the `switch` body and gives control to the next command outside the `switch` body.

Examples:

- The following example shows a `break` command in the action part of a `for` command. If the `i`-th element of the array `string` is equal to `'\0'`, the `break` command causes the `for` command to end.

```
for (i = 0; i < 5; i++) {
   if (string[i] == '\0')
      break;
   length++;
}
```

- The following `switch` command contains several `case` clauses and one `default` clause. Each clause contains a function call and a `break` command. The `break` commands prevent control from passing down through subsequent commands in the `switch` body.

```
char key;

key = '-';
AT LINE 15 switch (key)
{
   case '+':
      add();
      break;
   case '-':
      subtract();
      break;
   default:
      printf("Invalid key\n");
      break;
}
```

CALL Command

The CALL command invokes either a procedure, entry name, or program name, or it requests that an LE/VSE run-time dump be produced. The C equivalent for CALL is a function reference. PL/I subroutines or functions cannot be called dynamically during a Debug Tool session. The CALL keyword cannot be abbreviated.

In COBOL, the CALL command cannot be issued when Debug Tool is at initialization.

The various forms of the CALL command are summarized in Table 10.

Table 10 (Page 1 of 2). Summary of CALL Commands

CALL %DUMP	invokes the LE/VSE dump service to obtain a formatted dump.
CALL entry_name (COBOL)	invokes an entry name in the application program (COBOL).

Table 10 (Page 2 of 2). Summary of CALL Commands	
CALL procedure	invokes a procedure that has been defined with the PROCEDURE command.

CALL %DUMP

Invokes the LE/VSE dump service to obtain a formatted dump.

```
►►──CALL──%DUMP──┬─────────────────────────────────────┬──;──────────►◄
                 └─(──options_string──┬───────────┬──)─┘
                                      └─,──title──┘
```

title
Specifies the identification printed at the top of each page of the dump. It must be a fixed-length character string, conforming to the current programming language syntax for a character string constant (that is, enclosed in quotes according to the rules of that programming language). The string length cannot exceed 80 bytes.

options_string
A fixed-length character string, conforming to the current programming language syntax for a character string constant, which specifies the type, format, and destination of dump information. The string length cannot exceed 247 bytes.

Options are declared as a string of keywords separated by blanks or commas. Some options have suboptions that follow the option keyword and are contained in parentheses. The options can be specified in any order, but the last option declaration is honored if there is a conflict between it and any preceding options.

The *options_string* can include the following:

TRACEBACK
Requests a traceback of active procedures, blocks, condition handlers, and library phases on the call chain. The traceback shows transfers of control from either calls or exceptions. The traceback extends backwards to the main program of the current thread.

TRACEBACK can be abbreviated as TRACE.

NOTRACEBACK
Suppresses traceback.

NOTRACEBACK can be abbreviated as NOTRACE.

FILES
Requests a complete set of attributes of all files that are open and the contents of the buffers used by the files.

FILES can be abbreviated as FILE.

NOFILES
Suppresses file attributes of files that are open.

NOFILES can be abbreviated as NOFILE.

VARIABLES
Requests a symbolic dump of all variables, arguments, and registers.

Variables include arrays and structures. Register values are those saved in the stack frame at the time of call. There is no way to print a subset of this information.

Variables and arguments are printed only if the symbol tables are available. A symbol table is generated if a program is compiled using the compile options shown below for each language:

Language	Compile Option
C	TEST(SYM)
COBOL	TEST or TEST(hook,SYM)
PL/I	TEST(,SYM)

The variables, arguments, and registers are dumped starting with Debug Tool. The dump proceeds up the chain for the number of routines specified by the STACKFRAME option.

VARIABLES can be abbreviated as VAR.

NOVARIABLES
Suppresses dump of variables, arguments, and registers.

NOVARIABLES can be abbreviated as NOVAR.

BLOCKS
Produces a separate hexadecimal dump of control blocks used in LE/VSE and member language libraries.

Global control blocks and control blocks associated with routines on the call chain are printed. Control blocks are printed for Debug Tool. The dump proceeds up the call chain for the number of routines specified by the STACKFRAME option.

If FILES is specified, this is used to produce a separate hexadecimal dump of control blocks used in the file analysis.

BLOCKS can be abbreviated as BLOCK.

NOBLOCKS
Suppresses the hexadecimal dump of control blocks.

NOBLOCKS can be abbreviated as NOBLOCK.

STORAGE
Dumps the storage used by the program.

The storage is displayed in hexadecimal and character format. Global storage and storage associated with each routine on the call chain is printed. Storage is dumped for Debug Tool. The dump proceeds up the call chain for the number of routines specified by the STACKFRAME option. Storage for all file buffers is also dumped if the FILES option is specified.

STORAGE can be abbreviated as STOR.

NOSTORAGE
Suppresses storage dumps.

NOSTORAGE can be abbreviated as NOSTOR.

STACKFRAME(n|ALL)
Specifies the number of stack frames dumped from the call chain.

If STACKFRAME(ALL) is specified, all stack frames are dumped. No stack frame storage is dumped if STACKFRAME(0) is specified.

The particular information dumped for each stack frame depends on the VARIABLE, BLOCK, and STORAGE option declarations specified. The first stack frame dumped is the one associated with Debug Tool, followed by its caller, and proceeding backwards up the call chain.

STACKFRAME can be abbreviated to SF.

PAGESIZE(n)
Specifies the number of lines on each page of the dump.

This value must be greater than 9. A value of zero (0) indicates that there should be no page breaks in the dump.

PAGESIZE can be abbreviated to PAGE. The default setting is PAGESIZE(60).

FNAME(s)
Specifies the filename of the file where the dump report is written.

The default filename CEEDUMP is used if this option is not specified.

If the filename supplied is not valid, or the file specified by the filename is not defined in your job control, the output is written to SYSLST.

CONDITION
Specifies that for each condition active on the call chain, the following information is dumped from the Condition Information Block (CIB):

- The address of the CIB
- The message associated with the current condition token
- The message associated with the original condition token, if different from the current one
- The location of the error
- The machine state at the time the condition manager was invoked
- The ABEND code and REASON code, if the condition occurred because of an ABEND.

The particular information that is dumped depends on the condition that caused the condition manager to be invoked. The machine state is included only if a hardware condition or ABEND occurred. The ABEND and REASON codes are included only if an ABEND occurred.

CONDITION can be abbreviated as COND.

NOCONDITION
Suppresses dump condition information for active conditions on the call chain.

NOCONDITION can be abbreviated as NOCOND.

ENTRY
Includes in the dump a description of the Debug Tool routine that called the LE/VSE dump service and the contents of the registers at the point of the call. For the currently supported programming languages, ENTRY is extraneous and will be ignored.

NOENTRY
Suppresses the description of the Debug Tool routine that called the LE/VSE dump service and the contents of the registers at the point of the call.

The defaults for the preceding options are:

CONDITION
FILES
FNAME(CEEDUMP)
NOBLOCKS
NOENTRY
NOSTORAGE
PAGESIZE(60)
STACKFRAME(ALL)
THREAD(CURRENT)
TRACEBACK
VARIABLES

Usage Notes:

- As Debug Tool utilises C for its own file I/O, any dump produced will include information for an active C run-time environment.
- If incorrect options are used, a default dump is written.
- Debug Tool does not analyze any of the CALL %DUMP options, but just passes them along to the LE/VSE dump service. Some of these options might not be very appropriate, because the call is being made from Debug Tool rather than from your program.

  See *LE/VSE Programming Reference* for additional details on the CEE5DMP dump options.
- Control might not be returned to Debug Tool after the dump is produced, depending on the option string specified.
- COBOL does not do anything if the FILES option is specified; the BLOCKS option gives the file information instead.

  For detailed descriptions of dump output for the different HLLs, see *LE/VSE Debugging Guide and Run-Time Messages*.
- Using a small n (like 1 or 2) with the STACKFRAME option will not produce useful results because only the Debug Tool stack frames appear in your dump. Larger values of n or ALL should be used to ensure that application stack frames are shown.

Examples:

- Request a formatted dump that traces active procedures, blocks, condition handlers, and library phases. Identify the dump as "Dump after read".

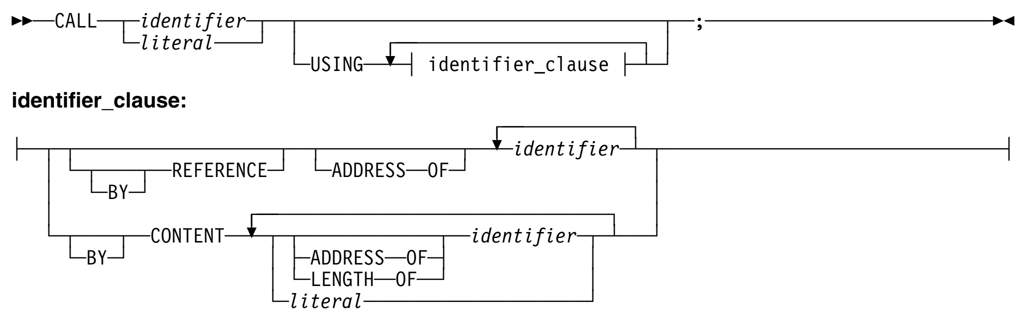
```
CALL %DUMP ("TRACEBACK", "Dump after read");
```

- Call the LE/VSE dump service to obtain a formatted dump including traceback information, file attributes, and buffers.

```
CALL %DUMP ("TRACEBACK FILES");
```

CALL entry_name (COBOL)

Invokes an entry name in the application program. The entry name must be a valid external entry point name (that is, callable from other compile units).



identifier
: A valid Debug Tool COBOL identifier.

literal
: A valid COBOL literal.

Usage Notes:

- If you have a COBOL entry point name that is the same as a Debug Tool procedure name, the procedure name takes precedence when using the CALL command. If you want the entry name to take precedence over the Debug Tool procedure name, you must qualify the entry name when using the CALL command.
- You can use the CALL entry_name command to change program flow dynamically. You can pass parameters to the called module.
- The CALL follows the same rules as CALLs within the COBOL language.
- The COBOL ON OVERFLOW and ON EXCEPTION phrases are not supported, so END-CALL is not supported.
- Only CALLs to separately compiled programs are supported; nested programs are not CALLable by this Debug Tool command (they can of course be invoked by GOTO or STEP to a compiled-in CALL).
- All CALLs are dynamic, that is, the CALLed program (whether specified as a *literal* or as an *identifier*) is loaded when it is CALLed.
- See *IBM COBOL for VSE/ESA Language Reference* for an explanation of the following COBOL keywords: ADDRESS, BY, CONTENT, LENGTH, OF, REFERENCE, USING.

Example:

Call the entry name `sub1` passing the variables `a`, `b`, and `c`.

```
CALL "sub1" USING a b c;
```

CALL procedure

Invokes a procedure that has been defined with the PROCEDURE command.

```
►►──CALL──procedure_name──;────────────────────────────────────►◄
```

procedure_name
The name given to a sequence of Debug Tool commands delimited by a PROCEDURE command and a corresponding END command.

Usage Notes:

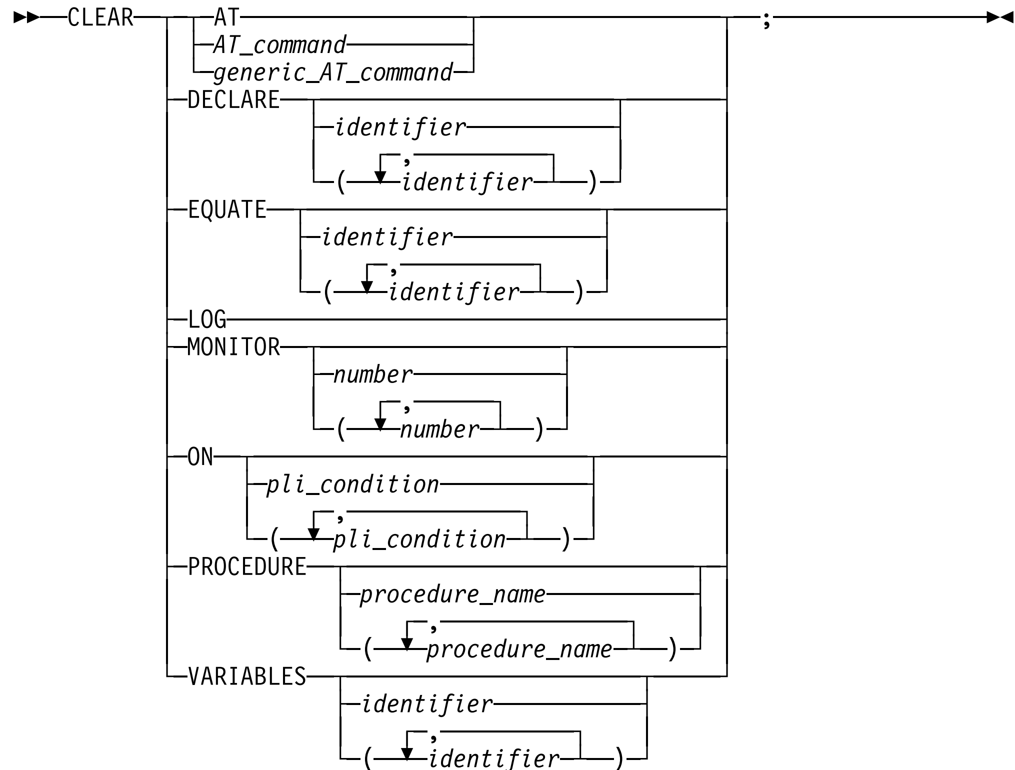
- Since the Debug Tool procedure names are always uppercase, the procedure name is converted to uppercase even for programming languages that have mixed-case symbols.
- The CALL keyword is required even for programming languages that do not use CALL for subroutine invocations.
- The CALL command is restricted to calling procedures in the currently executing enclave.

Example: Create and call the procedure named `proc1`.

```
proc1: PROCEDURE;
  LIST (r, c);
END;
AT 54 CALL proc1;
```

CLEAR Command

The CLEAR command removes the actions of previously issued Debug Tool commands. Some breakpoints are removed automatically when Debug Tool determines that they are no longer meaningful. For example, if you set a breakpoint in a fetched or loaded compile unit, the breakpoint is discarded when the compile unit is released.



AT

Removes all breakpoints from previously issued AT commands (including GLOBAL breakpoints).

AT_command

A valid AT command that includes at least one operand. See Table 9 on page 208 for a list of valid AT commands. The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted.

generic_AT_command

A valid AT command without operands. It can be one of the following:

ALLOCATE	DELETE	OCCURRENCE
APPEARANCE	ENTRY	PATH
CALL	EXIT	STATEMENT
CHANGE	LABEL	TERMINATION.
CURSOR	LOAD	

(the LINE keyword can be used in place of STATEMENT)

DECLARE

Removes previously defined variables and tags. If no *identifier* follows DECLARE, all session variables and tags are cleared. DECLARE is equivalent to VARIABLES.

identifier

The name of a session variable or tag declared during the Debug Tool session. This operand must follow the rules for the current programming language.

EQUATE
Removes previously defined symbolic references. If no *identifier* follows EQUATE, all existing SET EQUATE synonyms are cleared.

identifier
The name of a previously defined reference synonym declared during the Debug Tool session using SET EQUATE. This operand must follow the rules for the current programming language.

LOG
Erases the log file and clears out the data being retained for scrolling.

MONITOR
Clears the commands defined for MONITOR. If no *number* follows MONITOR, the entire list of commands affecting the monitor window is cleared; the Monitor window is empty.

number
A positive integer that refers to a monitored command. If a list of integers is specified, all commands represented by the specified list are cleared.

ON (PL/I)
Removes the effect of an earlier ON command. If no *pli_condition* follows ON, all existing ON commands are cleared.

pli_condition
Identifies an exception condition for which there is an ON command defined.

PROCEDURE
Clears previously defined Debug Tool procedures. If no *procedure_name* follows PROCEDURE, all inactive procedures are cleared.

procedure_name
The name given to a sequence of Debug Tool commands delimited by a PROCEDURE command and a corresponding END command. The procedure must be currently in storage and not active.

VARIABLES
Removes previously defined variables and tags. If no *identifier* follows VARIABLES, all session variables and tags are cleared. VARIABLES is equivalent to DECLARE.

identifier
The name of a session variable or tag declared during the Debug Tool session. This operand must follow the rules for the current programming language.

Usage Notes:

- Only an AT LINE or AT STATEMENT breakpoint can be cleared with a CLEAR AT CURSOR command.
- To clear every single breakpoint in the Debug Tool session, issue CLEAR AT followed by CLEAR AT TERMINATION.
- To clear a global breakpoint, you can specify an asterisk (*) with the CLEAR AT command or you can specify a CLEAR AT GLOBAL command.

If you have only a global breakpoint set and you specify CLEAR AT ENTRY without the asterisk (*) or GLOBAL keyword, you get a message saying there are no such breakpoints.

Examples:

- Remove the LABEL breakpoint set in the program at label `create`.

```
CLEAR AT LABEL create;
```

- Remove previously defined variables `x`, `y`, and `z`.

```
CLEAR DECLARE (x, y, z);
```

- Remove the effect of the ninth command defined for MONITOR.

```
CLEAR MONITOR 9;
```

- Remove the structure type definition `tagone` (assuming all variables declared interactively using the structure tag have been cleared). The current programming language setting is C.

```
CLEAR VARIABLES struct tagone;
```

- Establish some breakpoints with the AT command and then remove them with the CLEAR command (checking the results with the LIST command).

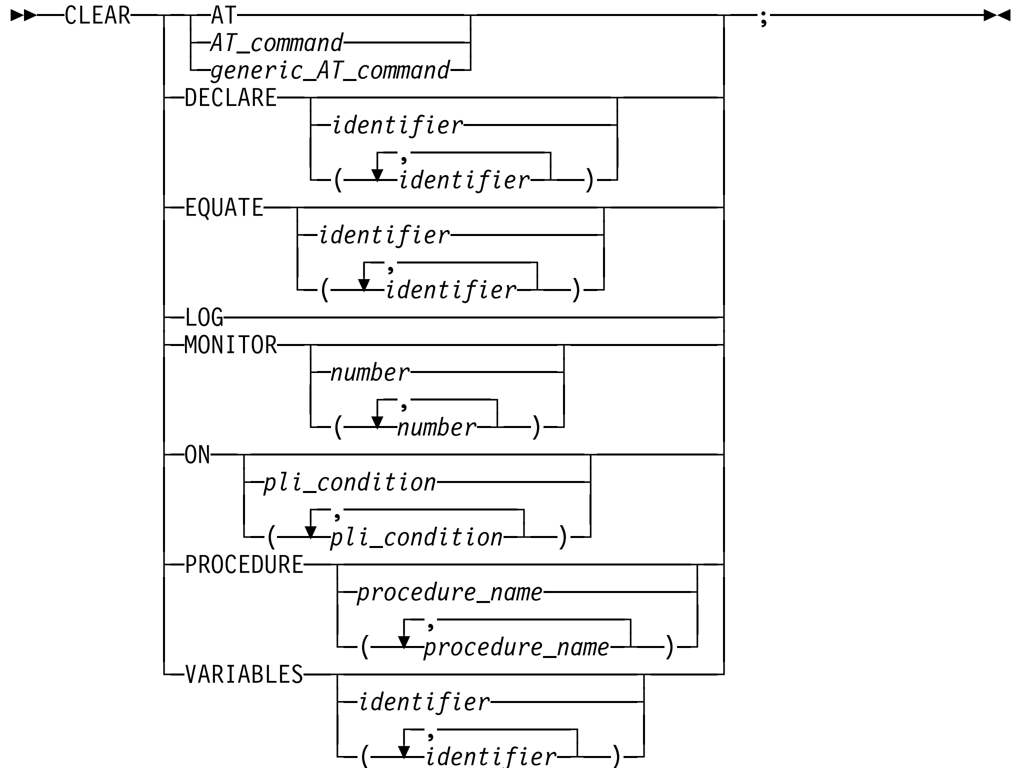
```
AT 50;
AT 56;
AT 55 LIST (r, c);
LIST AT;
CLEAR AT 50;
LIST AT;
CLEAR AT;
LIST AT;
```

- If you want to clear an AT ENTRY * breakpoint, specify:

```
CLEAR AT ENTRY *;
or
CLEAR AT GLOBAL ENTRY;
```

CLEAR Prefix (Full-Screen Mode)

Clears a breakpoint when you issue this command via the source window prefix area.



integer
Selects a relative statement (for C and PL/I) or a relative verb (for COBOL) within the line to remove the breakpoint if there are multiple statements on that line. The default value is 1.

Example:

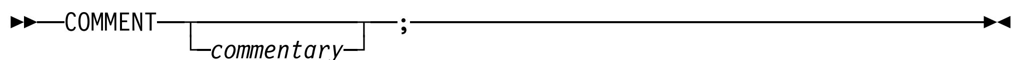
Clear a breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
CLEAR 3
```

No space is needed as a delimiter between the keyword and the integer; hence, CLEAR 3 is equivalent to CLEAR3.

COMMENT Command

The COMMENT command can be used to insert commentary in to the session log. The COMMENT keyword cannot be abbreviated.



commentary
Commentary text not including a semicolon. An embedded semicolon is not allowed; text after a semicolon is treated as another Debug Tool command. DBCS characters can be used within the commentary.

The COMMENT command can be used as an executable command, that is it can be the subject of a conditional command, but it is treated as a Null command.

Examples:

- Comment that `varblxx` seems to have the wrong value.

```
COMMENT At this point varblxx seems to have the wrong value;
```

- Combine a commentary with valid Debug Tool commands.

```
COMMENT Entering subroutine testrun; LIST (x); GO;
```

COMPUTE Command (COBOL)

The COMPUTE command assigns the value of an arithmetic expression to a specified reference. The COMPUTE keyword cannot be abbreviated.

```
►►──COMPUTE──reference──=──expression──;──────────────────────────────►◄
```

reference
A valid Debug Tool COBOL numeric reference.

expression
A valid Debug Tool COBOL numeric expression.

Usage Notes:

- If Debug Tool was invoked because of a computational condition or an attention interrupt, using an assignment to set a variable might not give expected results. This is due to the uncertainty of variable values within statements as opposed to their values at statement boundaries.
- COMPUTE assigns a value only to a single receiver; unlike COBOL, multiple receiver variables are not supported.
- Floating-point receivers are not supported; however, floating-point values can be set by using the MOVE command (see “MOVE Command (COBOL)” on page 286).
- The COBOL EQUAL keyword is not supported ("=" must be used).
- The COBOL ROUNDED and SIZE ERROR phrases are not supported, so END-COMPUTE is not supported.
- If the *expression* consists of a single numeric operand, the COMPUTE will be treated as a MOVE and therefore subject to the same rules as the MOVE command.

Examples:

- Assign to variable `x` the value of `a` + 6.

```
COMPUTE x = a + 6;
```

- Assign to the variable `mycode` the value of the Debug Tool variable %PATHCODE + 1.

```
COMPUTE mycode = %PATHCODE + 1;
```

CURSOR Command (Full-Screen Mode)

The CURSOR command moves the cursor between the last saved position on the Debug Tool session panel (excluding the header fields) and the command line.

```
►►──CURSOR──;──────────────────────────────────────────────────►◄
```

Usage Notes:

- The cursor position can be saved by typing the CURSOR command on the command line and moving the cursor before pressing Enter, or by moving the cursor and pressing a PF key with the CURSOR command assigned to it.
- If the CURSOR command precedes any command on the command line, the cursor is moved before the other command is performed. This can be useful in saving cursor movement for commands that are performed repeatedly in one of the windows.
- The CURSOR command is not logged.

Example:

Move the cursor between the last saved position on the Debug Tool session panel and the command line.

```
CURSOR;
```

Declarations

Use declarations to declare temporary variables and tags effective during a Debug Tool session. Session variables remain in effect for the entire debug session, or process in which they were declared. Variables and tags declared with declarations can be used in other Debug Tool commands as if they were declared to the compiler. Declared variables and tags are removed when your Debug Tool session ends or when the CLEAR command is used to remove them. The keywords must be the correct case and cannot be abbreviated.

Language Compatible Attributes

While working in one language, you can declare session variables that you can continue to use after calling in a phase of a different language. Table 11 on page 247 shows how session data attributes are mapped across programming languages. Attributes not shown in the table cannot be mapped to other programming languages.

Remember when declaring session variables that C variable names are case-sensitive. When the current programming language is C, only 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, variable names in mixed or lowercase are mapped to uppercase. These COBOL or PL/I variables can be declared or referenced using any mixture of lowercase and uppercase characters and it makes no difference. However, if the variable is shared with C, within 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).

Variables with incompatible attributes cannot be shared between other programming languages, but they do cause 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`.

Table 11. Attribute Mappings

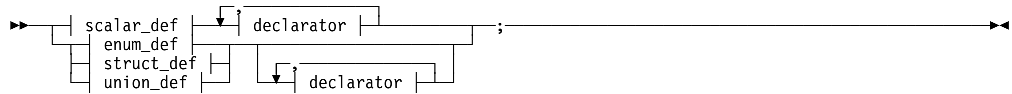
Machine Value	C Value	COBOL Value	PL/I Value
byte	unsigned char	PICTURE X	CHAR(1)
byte string	unsigned char[j]	PICTURE X(j)	CHAR(j)
halfword	signed short int	PICTURE S9(j<=4) USAGE BINARY	FIXED BIN(15,0)
fullword	signed long int	PICTURE S9(4<j<=9) USAGE BINARY	FIXED BIN(31,0)
floating point	float	USAGE COMP-1	FLOAT BIN(21) or FLOAT DEC(6)
long floating point	double	USAGE COMP-2	FLOAT BIN(53) or FLOAT DEC(16)
extended floated point	long double	n/a	FLOAT BIN(109) or FLOAT DEC(33)
fullword pointer	void *	USAGE POINTER	POINTER

Note:

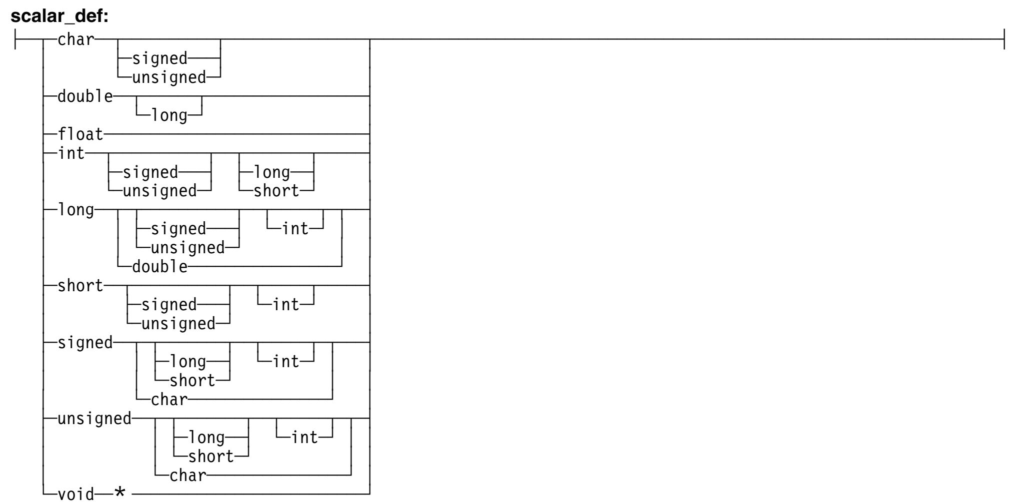
When registering session variables in PL/I, the DECIMAL type is always the default. For example, if C declares a `float`, PL/I registers the variable as a FLOAT DEC(6) rather than a FLOAT BIN(21).

Declarations (C)

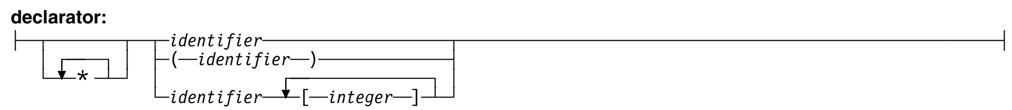
You can also declare `enum`, `struct`, and `union` data types. The syntax is identical to C except that `enum` members can only be initialized to an optionally signed integer constant.



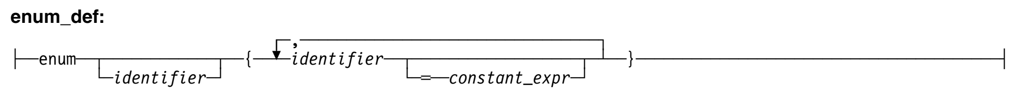
scalar_def:



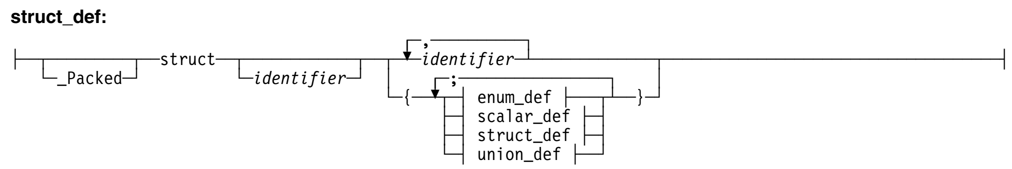
declarator:



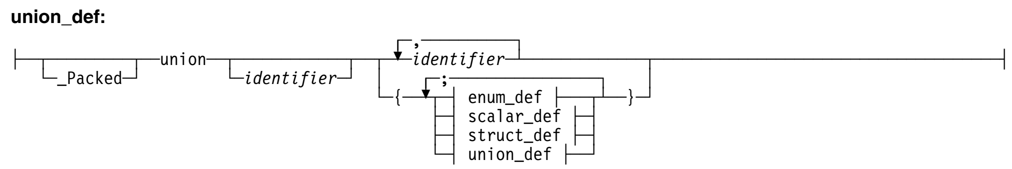
enum_def:



struct_def:



union_def:



* A C indirect operator.

identifier
A valid C identifier.

integer
A valid C array bound integer constant.

constant_expr
A valid C integer constant.

Usage Notes:

- As in C, the keywords can be specified in any order. For example, *unsigned long int* is equivalent to *int unsigned long*. Some permutations are shown in the syntax diagram to make sure that every keyword is shown at least once in the initial position.
- As in C, the identifiers are case-sensitive; that is, "X" and "x" are different names.
- A `structure` definition must have either an *identifier*, a *declarator*, or both specified.
- Initialization is not supported.
- A declaration cannot be used in a command list; for example, as the subject of an `if` command or `case` clause.
- Declarations of the form `struct tag identifier` must have the tag previously declared interactively.
- Only variables with attributes listed in the Table 11 on page 247 table can be declared.
- See *IBM C for VSE/ESA Language Reference* for an explanation the following keywords:

char	short
double	signed
enum	struct
float	union
int	unsigned
long	void
_Packed	

Examples:

- Define two C integers.

```
int myvar, hisvar;
```

- Define an enumeration variable `status` that represents the following values:

Enumeration Constant	Integer Representation
run	0
create	1
delete	5
suspend	6

```
enum statustag {run, create, delete=5, suspend} status;
```

- Define a variable in a `struct` declaration.

```
struct atag {
  char foo;
  int var1;
} avar;
```

- Interactively declare variables using structure tags.

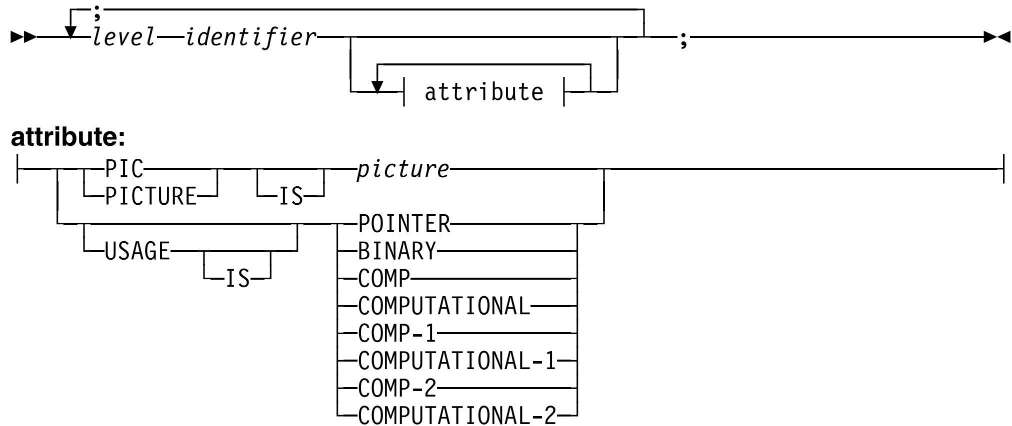
```
struct tagone {int a; int b;} c;
```

  then specify:

```
struct tagone d;
```

Declarations (COBOL)

The syntax for declarations in COBOL is:



level
: 1 or 77.

identifier
: A valid COBOL data name (including DBCS data names).

picture
: A sequence of characters from the set: S X 9 (replication factor is optional).

  If *picture* is not X(*), the COBOL USAGE clause is required.

Usage Notes:

- A declaration cannot be used in a command list; for example, as the subject of an IF command or WHEN clause.
- BINARY and COMP are equivalent.
- Use BINARY or COMP for COMPUTATIONAL-4.
- COMP-1 is short floating point (4 bytes).
- COMP-2 is long floating point (8 bytes).
- Only COBOL PICTURE and USAGE clauses are supported.
- Short forms of COMPUTATIONAL (COMP) are supported.
- Only variables with attributes listed in Table 11 on page 247 can be declared.

- See *IBM COBOL for VSE/ESA Language Reference* for an explanation of the following COBOL keywords:

BINARY	PIC
COMP	PICTURE
COMPUTATIONAL	POINTER
IS	USAGE

Examples:

- Define a variable named `floattmp` to hold a floating-point number.

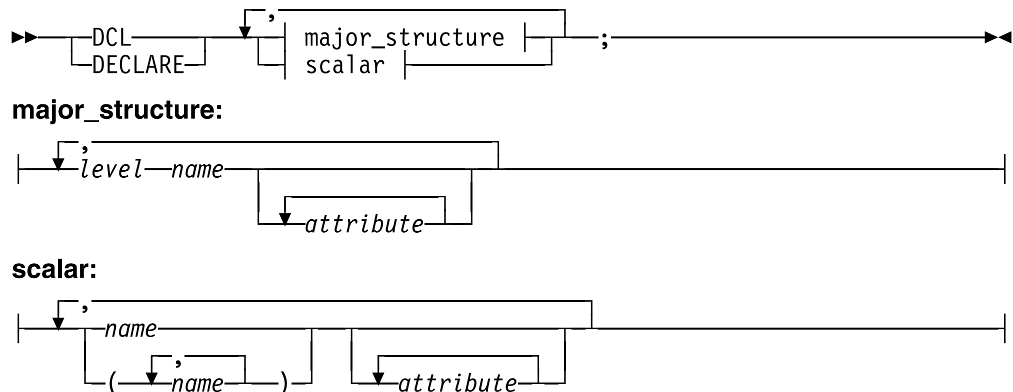
```
01 floattmp USAGE COMP-1;
```

- Define an integer variable name `temp`.

```
77 temp PIC S9(9) USAGE COMP;
```

DECLARE Command (PL/I)

The syntax for declarations in PL/I is:



level
An unsigned positive integer. Level 1 must be specified for major structure names.

name
A valid PL/I identifier. The name must be unique within a particular structure level.

When name conflicts occur, Debug Tool uses session variables before using other variables of the same name that appear in the running programs. Use qualification to refer to the program variable during a Debug Tool session. For example, to display the variable `a` declared with the DECLARE command as well as the variable `a` in the program, issue the LIST command as follows:

```
LIST (a, %BLOCK:a);
```

If a name conflict occurs because the variable was declared earlier with a DECLARE command, the new declaration overrides the previous one.

attribute
A PL/I data or storage attribute.

Acceptable PL/I data attributes are:

BINARY	CPLX	FIXED	LABEL	PTR
BIT	DECIMAL	FLOAT	OFFSET	REAL
CHARACTER	EVENT	GRAPHIC	POINTER	VARYING
COMPLEX				

Acceptable PL/I storage attributes are:

ALIGNED	BASED	UNALIGNED

Only simple factoring of attributes is allowed. DECLAREs such as the following are not allowed:

```
DCL (a(2), b) PTR;
DCL (x REAL, y CPLX) FIXED BIN(31);
```

Also, the precision attribute and scale factor as well as the bounds of a dimension can be specified. If a temporary variable has dimensions and bounds, these must be declared following PL/I Language rules. See *IBM PL/I for VSE/ESA Language Reference* for more details.

Usage Notes:

- DECLARE is not valid as a subcommand. That is, it cannot be used as part of a DO/END or BEGIN/END block.
- Initialization is not supported.
- Program DEFAULT statements do not affect the DECLARE command.
- Only variables with attributes listed in Table 11 on page 247 can be shared.
- See *IBM PL/I for VSE/ESA Language Reference* for an explanation of the following PL/I data and storage attributes:

ALIGNED	CHARACTER	EVENT	LABEL	REAL
BASED	COMPLEX	FIXED	OFFSET	UNALIGNED
BINARY	CPLX	FLOAT	POINTER	VARYING
BIT	DECIMAL	GRAPHIC	PTR	

Examples:

- Declare x, y, and z as variables that can be used as pointers.

```
DECLARE (x, y, z) POINTER;
```

- Declare a as a variable that can represent binary, fixed-point data items of 15 bits.

```
DECLARE a FIXED BIN(15);
```

- Declare d03 as a variable that can represent binary, floating-point, complex data items.

```
DECLARE d03 FLOAT BIN COMPLEX;
```

  This d03 will have the attribute of FLOAT BINARY(21).

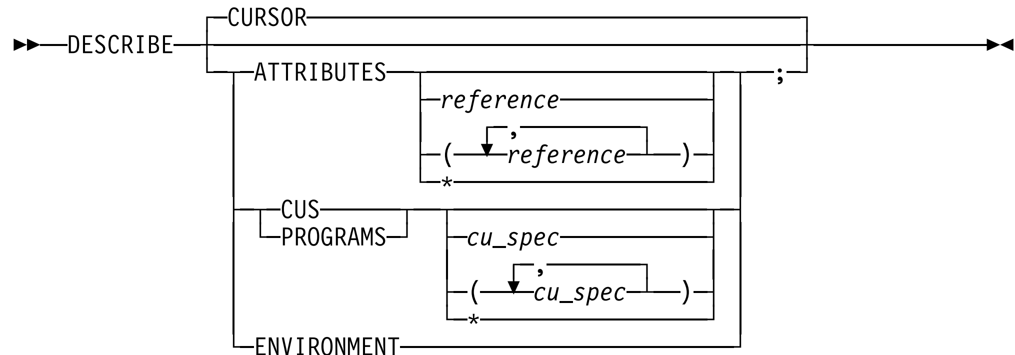
- Declare x as a pointer, and setx as a major structure with structure elements a and b as fixed-point data items.

```
DECLARE x POINTER, 1 setx, 2 a FIXED, 2 b FIXED;
```

This a and b will have the attributes of FIXED DECIMAL(5).

DESCRIBE Command

The DESCRIBE command displays the attributes of references, compile units, and the execution environment.



CURSOR (Full-Screen Mode only)
Provides a cursor-controlled method for describing variables, structures, and arrays. If you have assigned DESCRIBE to a PF key, you can display the attributes of a selected variable by positioning the cursor at that variable and pressing the assigned PF key.

ATTRIBUTES
Displays the attributes of a specified variable or, in C, a tag or expression. The attributes are ordinarily those that appeared in the declaration of a variable or are assumed because of the defaulting rules. DESCRIBE ATTRIBUTES works only for variables accessible to the current programming language. All variables in the currently qualified block are described if no operand is specified.

reference
A valid Debug Tool reference in the current programming language. Note:

- In C, this can be a valid expression, enumeration tag, structure tag, or union tag identifier. For a C expression, the type is the only attribute displayed. You must use enum, struct, or union when referencing the C tag; see “Declarations (C)” on page 247 for more information.
- In COBOL, this can be any user-defined name appearing in the DATA DIVISION. Names can be subscripted or substringed if they are defined as arrays or alphanumeric data.
- In PL/I, if the variable is in a structure, it can have inherited dimensions from a higher level parent. The inherited dimensions appear as if they have been part of the declaration of the variable.
- For more information, see “References” on page 203.

*
Describes all variables in the compile unit.

CUS
Describes the attributes of compile units, including such things as the compile-time options and list of internal blocks. The information returned is dependent on the HLL that the compile unit was compiled under. CUS is equivalent to PROGRAMS.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. The default is the currently qualified compile unit.

* Describes all compile units.

PROGRAMS
Is equivalent to CUS.

ENVIRONMENT
The information returned includes a list of the currently opened files. Names of files that have been opened but are not currently open are excluded from the list. COBOL does not provide any information for DESCRIBE ENVIRONMENT.

Usage Notes:

- Cursor pointing can be used by typing the DESCRIBE CURSOR command on the command line and moving the cursor to a variable in the source window before pressing Enter, or by moving the cursor and pressing a PF key with the DESCRIBE CURSOR command assigned to it.
- When using the DESCRIBE CURSOR command for a variable that is located by the cursor position, the variable's name cannot be split across different lines of the source listing.
- In C and COBOL, expressions containing parentheses () must be enclosed in another set of parentheses when used with the DESCRIBE ATTRIBUTES command. For example, `DESCRIBE ATTRIBUTES ((x + y) / z);`.
- For PL/I, DESCRIBE ATTRIBUTES will return only the top-level names for structures. DESCRIBE ATTRIBUTES * is not supported for PL/I. To get more detail, specify the structure name as the *reference*.

Examples:

- Display the attributes of the enumeration variable `sum`.

```
DESCRIBE ATTRIBUTES enum sum;
```

- Describe the attributes of `argc`, `argv`, `boolean`, `i`, `ld`, and `structure`.

```
DESCRIBE ATTRIBUTES (argc, argv, boolean, i, ld, structure);
```

- Describe the current environment.

```
DESCRIBE ENVIRONMENT;
```

- Display information describing program `myprog`.

```
DESCRIBE PROGRAMS myprog;
```

DISABLE Command

The DISABLE command makes the AT breakpoint inoperative, but does not clear it; you can ENABLE it later without typing the entire command again.

```
►►──DISABLE──AT_command──────────────────────────────────────────►◄
```

AT_command
An enabled AT command. The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted. Valid forms are the same as those allowed with CLEAR AT.

Usage Notes:

- To re-enable a disabled AT command, use the ENABLE command.
- Disabling an AT command does not prevent its replacement by a new (enabled) version if an overlapping AT command is later specified. It also does not prevent removal by a CLEAR AT command.
- Breakpoints already disabled within the range(s) specified in the specific AT command are unaffected; however, a warning message is issued for any specified range found to contain no enabled breakpoints.

Examples:

- Disable the breakpoint that was set by the command AT ENTRY `myprog` CALL `proc1`;.

  ```
  DISABLE AT ENTRY myprog;
  ```

- If statement 25 is in a loop and you set the following breakpoint:

  ```
  AT EVERY 5 FROM 1 TO 100 STATEMENT 25 LIST x;
  ```

  to disable it, enter:

  ```
  DISABLE AT STATEMENT 25;
  ```

  You do not need to reenter the every_clause or the command list. To restore the breakpoint, enter:

  ```
  ENABLE AT STATEMENT 25;
  ```

DISABLE Prefix (Full-Screen Mode)

Disables a statement breakpoint when you issue this command via the Source window prefix area.

```
►►──DISABLE──┬─────────┬─────────────────────────────────────────►◄
             └─integer─┘
```

integer
Selects a relative statement (for C or PL/I) or a relative verb (for COBOL) within the line. The default value is 1.

Example:

Disable the breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
DIS 3
```

For an example of the prefix area, see Figure 20 on page 89.

No space is needed as a delimiter between the keyword and the integer; hence, DIS 3 is equivalent to DIS3.

do/while Command (C)

The `do/while` command performs a command before evaluating the test expression. Due to this order of execution, the command is performed at least once. The `do` and `while` keywords must be lowercase and cannot be abbreviated.

```
►►──do──command──while──(──expression──)──;──────────────────────────►◄
```

command
: A valid Debug Tool command.

expression
: A valid Debug Tool C expression.

The body of the loop is performed before the `while` clause is evaluated. Further execution of the `do/while` command depends on the value of the `while` clause. If the `while` clause does not evaluate to false, the command is performed again. Otherwise, execution of the command ends.

A `break` command can cause the execution of a `do/while` command to end, even when the `while` clause does not evaluate to false.

Example:

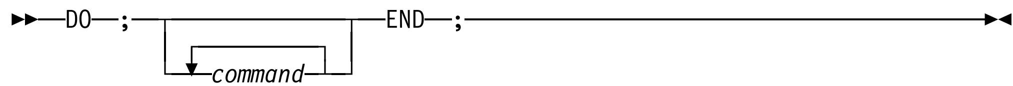
Perform a loop 11 times, and list the value of `ctr` each time through the loop.

```
int ctr;

do {
  ctr++;
  LIST ctr;
} while (ctr <= 10);
```

DO Command (PL/I)

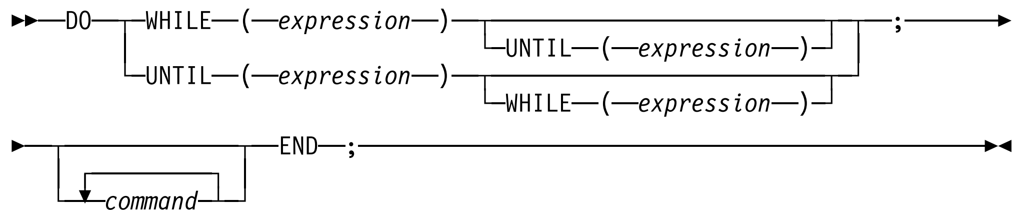
The DO command allows one or more commands to be collected into a group which can (optionally) be repeatedly executed. The DO and END keywords delimit a group of commands collectively called a DO group. The keywords cannot be abbreviated.

Simple



command
A valid Debug Tool command.

Repeating



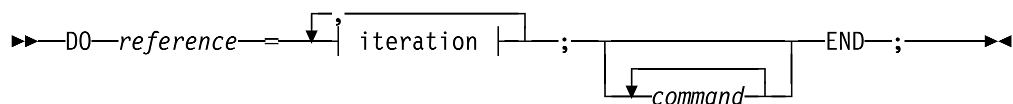
WHILE
Specifies that *expression* is evaluated before each execution of the command list. If the expression evaluates to true, the commands are executed and the DO group begins another cycle; if it evaluates to false, execution of the DO group ends.

expression
A valid Debug Tool PL/I Boolean expression.

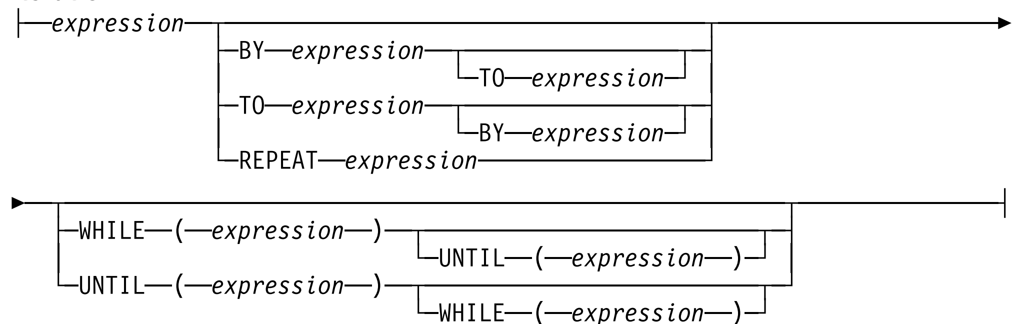
UNTIL
Specifies that *expression* is evaluated after each execution of the command list. If the expression evaluates to false, the commands are executed and the DO group begins another cycle; if it evaluates to true, execution of the DO group ends.

command
A valid Debug Tool command.

Iterative



iteration:



reference
A valid Debug Tool PL/I reference.

expression
A valid Debug Tool PL/I expression.

BY
Specifies that *expression* is evaluated at entry to the DO specification and saved. This saved value specifies the increment to be added to the control variable after each execution of the DO group.

If BY *expression* is omitted from a DO specification and if TO *expression* is specified, *expression* defaults to the value of 1.

If BY 0 is specified, the execution of the DO group continues indefinitely unless it is halted by a WHILE or UNTIL option, or control is transferred to a point outside the DO group.

The BY option allows you to vary the control variable in fixed positive or negative increments.

TO
Specifies that *expression* is evaluated at entry of the DO specification and saved. This saved value specifies the terminating value of the control variable.

If TO *expression* is omitted from a DO specification and if BY *expression* is specified, repetitive execution continues until it is terminated by the WHILE or UNTIL option, or until some statement transfers control to a point outside the DO group.

The TO option allows you to vary the control variable in fixed positive or negative increments.

REPEAT
Specifies that *expression* is evaluated and assigned to the control variable after each execution of the DO group. Repetitive execution continues until it is terminated by the WHILE or UNTIL option, or until some statement transfers control to a point outside the DO group.

The REPEAT option allows you to vary the control variable nonlinearly. This option can also be used for nonarithmetic control variables, such as pointers.

WHILE
Specifies that *expression* is evaluated before each execution of the command list. If the expression evaluates to true, the commands are executed and the DO group begins another cycle; if it evaluates to false, execution of the DO group ends.

UNTIL
Specifies that *expression* is evaluated after each execution of the command list. If the expression evaluates to false, the commands are executed and the DO group begins another cycle; if it evaluates to true, execution of the DO group ends.

command
A valid Debug Tool command.

Examples:

- At statement 25, initialize variable a and display the values of variables x, y, and z.

```
AT 25 DO; %BLOCK:>a = 0; LIST (x, y, z); END;
```

- Execute the DO group until ctr is greater than 4 or less than 0.

```
DO UNTIL (ctr > 4) WHILE (ctr >= 0); ...; END;
```

- Execute the DO group with i having the values 1, 2, 4, 8, 16, 32, 64, 128, and 256.

```
DO i = 1 REPEAT 2*i UNTIL (i = 256); END;
```

- Repeat execution of the DO group with j having values 1 through 20, but only if k has the value 1.

```
DO j = 1 TO 20 BY 1 WHILE (k = 1); END;
```

ENABLE Command

The ENABLE command makes the AT breakpoints operative after they have been disabled.

```
►►──ENABLE──AT_command──────────────────────────────────────────────►◄
```

AT_command
A disabled AT command. The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted. Valid forms are the same as those allowed with CLEAR AT.

Usage Notes:

- To disable an AT command, use the DISABLE command.
- Breakpoints already enabled within the range(s) specified in the specific AT command are unaffected; however, a warning message is issued for any specified range found to contain no disabled breakpoints.

Example:

Enable the previously disabled command AT ENTRY mysub CALL proc1;.

```
ENABLE AT ENTRY mysub;
```

ENABLE Prefix (Full-Screen Mode)

Enables a disabled statement breakpoint when you issue this command via the Source window prefix area.

```
►►──ENABLE──┬─────────┬──────────────────────────────────────────────►◄
            └─integer─┘
```

integer
Selects a relative statement (for C or PL/I) or a relative verb (for COBOL) within the line. The default value is 1.

Example:

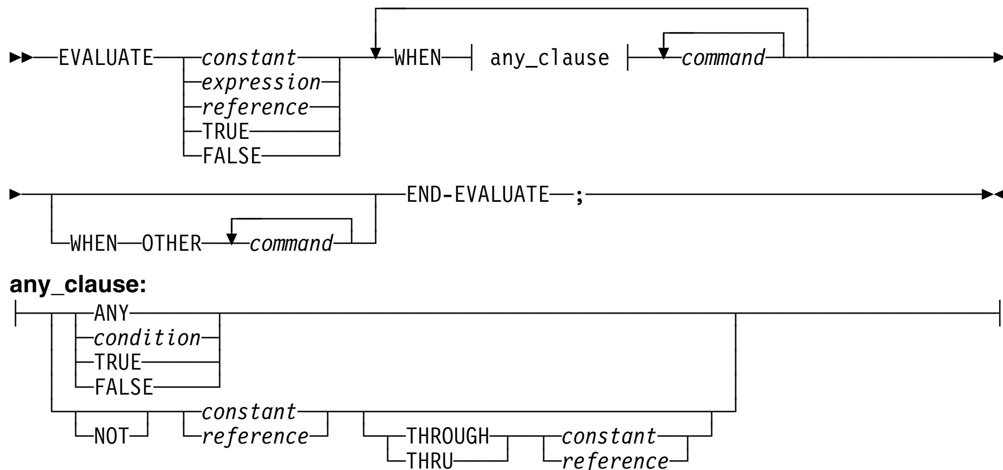
Enable the breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
ENABLE 3
```

No space is needed as a delimiter between the keyword and the integer; hence, ENABLE 3 is equivalent to ENABLE3.

EVALUATE Command (COBOL)

The EVALUATE command provides a shorthand notation for a series of nested IF statements. The keywords cannot be abbreviated.



constant
A valid Debug Tool COBOL constant.

expression
A valid Debug Tool COBOL arithmetic expression.

reference
A valid Debug Tool COBOL reference.

condition
A simple relation condition.

command
A valid Debug Tool command.

Usage Notes:

- Only a single subject is supported.
- Consecutive WHENs without associated commands are not supported.
- THROUGH/THRU ranges can be specified as constants or references.
- See *IBM COBOL for VSE/ESA Language Reference* for an explanation of the following COBOL keywords:

ANY	THROUGH
FALSE	THRU
NOT	TRUE
OTHER	WHEN

Example:

The following example shows an EVALUATE command and the equivalent coding for an IF command.

```
EVALUATE menu-input
  WHEN "0"
    CALL init-proc
  WHEN "1" THRU "9"
    CALL process-proc
  WHEN "R"
    CALL read-parms
  WHEN "X"
    CALL cleanup-proc
  WHEN OTHER
    CALL error-proc
END-EVALUATE;
```

The equivalent IF command.

```
IF (menu-input = "0") THEN
  CALL init-proc
ELSE
  IF (menu-input >= "1") AND (menu-input <= "9") THEN
    CALL process-proc
  ELSE
    IF (menu-input = "R") THEN
      CALL read-parms
    ELSE
      IF (menu-input = "X") THEN
        CALL cleanup-proc
      ELSE
        CALL error-proc
      END-IF;
    END-IF;
  END-IF;
END-IF;
```

Expression Command (C)

The Expression command evaluates the given expression. The expression can be used to either assign a value to a variable or to call a function.

```
►►──expression──;──────────────────────────────────────────►◄
```

expression
A valid Debug Tool C expression. Assignment is affected by including one of the C assignment operators in the expression. No use is made of the value resulting from a stand-alone expression.

Usage Note:

Function invocations in expressions are restricted to functions contained in the currently executing enclave.

Examples:

- Initialize the variables x, y, z and note that function invocations are supported.

```
x = 3 + 4/5;
y = 7;
z = 8 * func(x, y);
```

- Increment y and assign the remainder of the integer division of omega by 4 to alpha.

```
alpha = (y++, omega % 4);
```

FIND Command

The FIND command provides full-screen, and batch mode search capability in source and listing files, and full-screen searching of log and monitor objects as well.

string
The string searched for, conforming to the current programming language syntax for a character string constant. The string length cannot exceed 128 bytes, excluding the quotes.

If *string* is not specified, the string from the previous FIND command is used.

Some examples of possible strings follow:

C	COBOL	PL/I
"ABC"	"A5"	'P76'
"IntLink::*"	'A5'	

CURSOR (Full-Screen Mode)
Specifies that the current cursor position selects the window searched.

LOG (Full-Screen Mode)
Selects the session Log window.

MONITOR (Full-Screen Mode)
Selects the Monitor window.

SOURCE (Full-Screen Mode)
Selects the Source window.

Usage Notes:

- Window defaulting can be controlled by the SET DEFAULT WINDOW command.
- If the current programming language setting is C, the search is case-sensitive. Otherwise, the search is not case-sensitive.
- In full-screen mode, the search begins at the top line displayed in the window or at the location of the last found search argument if a previous FIND was issued for any search string. If the end of the object is reached without finding the search argument, FIND wraps to the top of the object and continues the search. A message notifies you that wrapping has occurred.

  If the search argument is found, the window is scrolled until it is visible. If the search target is DBCS, it is displayed as is. If the search target is not DBCS, it is highlighted as specified by the SET COLOR command and the cursor is placed at the beginning of the target. If the search target is not found, the screen position is unchanged and the cursor is not moved.
- FIND can be made immediately effective in full-screen mode with the IMMEDIATE command.
- In batch mode, the search begins at the first line of the source listing or source file, or at the location of the last found search argument if a previous FIND was issued for the same string. If the end of the listing is reached without finding the search argument, FIND wraps to the top of the listing and continues the search without notification. However, the line number is identified in the output.

  If the search argument is found, the line containing it is displayed with a vertical bar character (|) beneath the search target.
- For PL/I, if the line found is not the first line of the statement, all lines from the start of the statement are displayed, up to and including the target line.
- The full-screen FIND command is not logged; however, the FIND command is logged in batch mode.

Example:

Indicate that you want to search the Monitor window for the name `myvar`. The current programming language setting is either C or COBOL.

```
FIND "myvar" MONITOR;
```

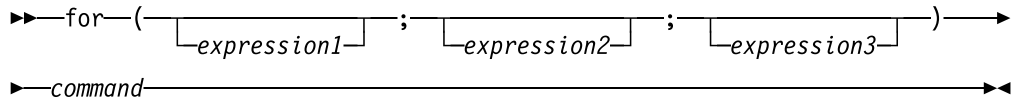
for Command (C)

The `for` command provides iterative looping similar to the C `for` statement. It enables you to do the following:

- Evaluate an expression before the first iteration of the command ("initialization").
- Specify an expression to determine whether the command should be performed again ("controlling part").
- Evaluate an expression after each iteration of the command.

- Perform the command, or `block`, if the controlling part does not evaluate to false.

The `for` keyword must be lowercase and cannot be abbreviated.



expression1,2,3
: A valid Debug Tool C expression.

command
: A valid Debug Tool command.

Debug Tool evaluates *expression1* only before the command is performed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression before the first iteration of the command, you can omit this expression.

Debug Tool evaluates *expression2* before each execution of the command. If this expression evaluates to false, the command does not run and control moves to the command following the `for` command. Otherwise, the command is performed. If you omit *expression2*, it is as if the expression has been replaced by a nonzero constant and the `for` command is not terminated by failure of this expression.

Debug Tool evaluates *expression3* after each execution of the command. You might use this expression to increase, decrease, or reinitialize a variable. If you do not want to evaluate an expression after each iteration of the command, you can omit *expression3*.

A `break` command can cause the execution of a `for` command to end, even when the second expression does not evaluate to false. If you omit the second expression, you must use a `break` command to stop the execution of the `for` command.

Examples:

- The following `for` command lists the value of `count` 20 times. The `for` command initially sets the value of `count` to 1. After each execution of the command, `count` is incremented.

  ```
  for (count = 1; count <= 20; count++)
    LIST TITLED count;
  ```

  Alternatively, the preceding example can be written with the following sequence of commands to accomplish the same task.

  ```
  count = 1;
  while (count <= 20) {
    printf("count = %d\n", count);
    count++;
  }
  ```

- The following `for` command does not contain an initialization expression.

```
for (; index > 10; --index) {
  varlist[index] = var1 + var2;
  printf("varlist[%d] = %d\n", index, varlist[index]);
}
```

GO Command

The GO command causes Debug Tool to start or resume running your program.

```
►►──GO──┬────────┬──;──────────────────────────────────────────►◄
        └─BYPASS─┘
```

BYPASS
Bypasses the user or system action for the AT-condition that caused the breakpoint. It is valid only when Debug Tool is entered for an:

AT CALL breakpoint, or

HLL or LE/VSE condition

Usage Notes:

- If GO is specified in a command list (for example, as the subject of an IF command or WHEN clause), all subsequent commands in the list are ignored.
- If GO is specified within the body of a loop, it causes the execution of the loop to end.
- To suppress the logging of GO commands, use the SET ECHO OFF command.
- GO with no operand specified does not actually resume the program if there are additional AT-conditions that have not yet been processed. See the usage notes for the AT commands on page 209 for an explanation on processing multiple AT-conditions.

Examples:

- Resume execution.

```
GO;
```

- Resume execution and bypass user and system actions for the AT-condition that caused the breakpoint.

```
GO BYPASS;
```

- Your application has abended with a protection exception, so an OCCURRENCE breakpoint has been triggered. Correct the results of the instruction which caused the exception and issue `GO BYPASS;` to continue processing as if the abend had not occurred.

GOTO Command

The GOTO command causes Debug Tool to resume program execution at the specified statement id. The GOTO keyword cannot be abbreviated. If you want Debug Tool to return control to you at a target location, make sure there is a breakpoint at that location.

```
►►──┬─GOTO────┬──statement_id──;──────────────────────────────►◄
    └─GO──TO──┘
```

statement_id
A valid statement id; see “Statement_Id” on page 203.

Usage Notes:

- If GOTO is specified in a command list (for example, as the subject of an IF command or WHEN clause), all subsequent commands in the list are ignored.
- PL/I allows GOTO in a command list on a call to PLITEST or CEETEST.
- For COBOL, the GOTO command follows the COBOL language rules for the GOTO statement.
- In PL/I, out-of-block GOTOs are allowed. However, qualification may be needed.
- Statement GOTO's are not restricted if the program is compiled with minimum optimization.
- Because statements may be removed by the compiler during optimization, specify a reference or statement with the GOTO command that can be reached during program execution. You can issue the LIST STATEMENT NUMBERS command to determine the reachable statements. See “Qualifying Variables and Changing the Point of View” on page 129.

Examples:

- Resume execution at statement 23, where statement 23 is in a currently active block.

  ```
  GOTO 23;
  ```

  If there's no breakpoint at statement 23, Debug Tool will run from statement 23 until a breakpoint is hit.

- Resume execution at statement 45, where statement 45 is in a currently active block.

  ```
  AT 45
  GOTO 45
  ```

GOTO LABEL Command

The GOTO LABEL command causes Debug Tool to resume program execution at the specified statement label. The specified label must be in the same block. If you want Debug Tool to return control to you at the target location, make sure there is a breakpoint at that location.

```
►►──┬─GOTO────┬──┬───────┬──statement_label──;──────────────────────►◄
    └─GO──TO──┘  └─LABEL─┘
```

statement_label
A valid statement label within the currently executing program or, in PL/I, a label variable. See “Statement_Label” on page 205.

Usage Notes:

- In PL/I, out-of-block GOTOs are allowed. However, qualification might be needed. See “Qualifying Variables and Changing the Point of View” on page 129.
- The LABEL keyword is optional when either the target *statement_label* is nonnumeric or if it is qualified (whether the actual label was nonnumeric or not).
- For COBOL, you can use GOTO LABEL only if you compiled your program with either PATH or ALL suboption and the SYM suboption of the compile-time TEST option. A COBOL *statement_label* can have either of the following forms:
  - *name*

    This form can be used in COBOL for reference to a section name or for a COBOL paragraph name that is not within a section or is in only one section of the block.
  - *name1* OF *name2* or *name1* IN *name2*

    This form must be used for any reference to a COBOL paragraph (name1) that is within a section (name2), if the same name also exists in other sections in the same block. You can specify either OF or IN, but Debug Tool always uses OF for output.

  Either form can be prefixed with the usual block, compile unit, and phase qualifiers.
- For C, you can use GOTO LABEL only if you compiled your program with either the PATH or ALL suboption and the SYM suboption of the compile-time TEST option. There are no restrictions on using labels with GOTO LABEL.
- For PL/I, you can use GOTO LABEL only if you compiled your program with either the PATH or ALL suboption and the SYM suboption of the compile-time TEST option. There are no restrictions on using labels with GOTO LABEL and label variables are supported.

Examples:

- Go to the label constant `laba` in block `suba` in program `prog1`.

  ```
  GOTO prog1:>suba:>laba;
  ```

- Go to the label constant `para OF sect1`. The current programming language setting is COBOL.

  ```
  GOTO LABEL para OF sect1;
  ```

if Command (C)

The `if` command lets you conditionally perform a command. You can optionally specify an `else` clause on the `if` command. If the test expression evaluates to false and an `else` clause exists, the command associated with the `else` clause is performed. The `if` and `else` keywords must be lowercase and cannot be abbreviated.

```
►►──if──(──expression──)──command──┬──────────────────┬──────────────────►◄
                                   └─else──command─┘
```

expression
  A valid Debug Tool C expression.

command
  A valid Debug Tool command.

When `if` commands are nested and `else` clauses are present, a given `else` is associated with the closest preceding `if` clause within the same block.

Usage Note:

- An `else` clause should always be included if the `if` clause causes Debug Tool to get more input (for example, an `if` containing USE or other commands that cause Debug Tool to be reinvoked because an AT-condition occurs).

Examples:

- The following example causes `grade` to receive the value "A" if the value of `score` is greater than or equal to 90.

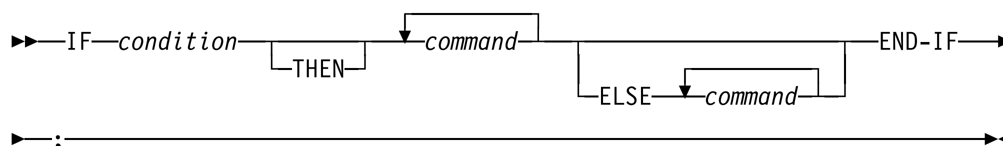
  ```
  if (score >= 90)
     grade = "A";
  ```

- The following example shows a nested `if` command.

  ```
  if (paygrade == 7) {
     if (level >= 0 && level <= 8)
        salary *= 1.05;
     else
        salary *= 1.04;
  }
  else
     salary *= 1.06;
  ```

IF Command (COBOL)

The IF command lets you conditionally perform a command. You can optionally specify an ELSE clause on the IF command. If the test expression evaluates to false and an ELSE clause exists, the command associated with the ELSE clause is performed. The keywords cannot be abbreviated.



condition
A simple relation condition.

command
A valid Debug Tool command.

When IF commands are nested and ELSE clauses are present, a given ELSE or END-IF is associated with the closest preceding IF clause within the same block.

Unlike COBOL, Debug Tool requires terminating punctuation (;) after commands. The END-IF keyword is required.

Usage Notes:

- An ELSE clause should always be included if the IF clause causes Debug Tool to get more input (for example, an IF containing USE or other commands that cause Debug Tool to be reinvoked because an AT-condition occurs).
- The COBOL NEXT SENTENCE phrase is not supported.
- Only the comparison combinations listed in "Allowable Comparisons for the Debug Tool IF Command" on page 349, are supported.

Examples:

- If the value of `field-1` is equal to the value of `field-2`, go to the statement with label constant `label-1`. Execution of the user program continues at `label-1`. If `array-1` does not equal `array-2`, the GOTO is not performed and control is passed to the user program.

```
IF field-1 = field-2 THEN GOTO LABEL label-1; ELSE GO;
```

- Set a breakpoint at statement 23, which will test if variable `j` is equal to 10, display the names and values of variables `rmdr`, `totodd`, and `terms(j)`. If variable `j` is not equal to 10, continue program execution.

```
AT 23 IF j = 10 THEN LIST TITLED (rmdr, totodd, terms(j)); ELSE GO;
```

IF Command (PL/I)

The IF command lets you conditionally perform a command. You can optionally specify an ELSE clause on the IF command. If the test expression evaluates to false and an ELSE clause exists, the command associated with the ELSE clause is performed. The keywords cannot be abbreviated.

expression
A valid Debug Tool PL/I expression.

If necessary, the expression is converted to a BIT string.

command
A valid Debug Tool command.

When IF commands are nested and ELSE clauses are present, a given ELSE is associated with the closest preceding IF clause within the same block.

Usage Note:

- An ELSE clause should always be included if the IF clause causes Debug Tool to get more input (for example, an IF containing USE or other commands that cause Debug Tool to be reinvoked because an AT-condition occurs).

Examples:

- If the value of `array1` is equal to the value of `array2`, go to the statement with label constant `label_1`. Execution of the user program continues at `label_1`. If `array1` does not equal `array2`, the GOTO is not performed and control is passed to the user program.

```
IF array1 = array2 THEN GOTO LABEL label_1; ELSE GO;
```

- Set a breakpoint at statement 23, which will test if variable `j` is equal to 10, display the names and values of variables `rmdr`, `totodd`, and `terms(j)`. If variable `j` is not equal to 10, continue program execution.

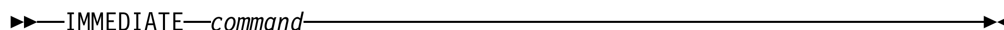
```
AT 23 IF j = 10 THEN LIST TITLED (rmdr, totodd, terms(j)); ELSE GO;
```

IMMEDIATE Command (Full-Screen Mode)

The IMMEDIATE command causes a command within a command list to be performed immediately. It is intended for use with commands assigned to a PF key.

IMMEDIATE can only be entered as an unnested command or within a compound command.

It is recommended that PF key definitions for FIND, RETRIEVE, SCROLL, and WINDOW commands be prefixed with IMMEDIATE. This makes it possible to do things like SCROLL even when entering a group of commands.



command
One of the following Debug Tool commands:

FIND
RETRIEVE
SCROLL commands
BOTTOM

DOWN
LEFT
NEXT
RIGHT
TO
TOP
UP
WINDOW commands
CLOSE
OPEN
SIZE
ZOOM

Usage Note: The IMMEDIATE command is not logged.

Examples:

- Specify that the WINDOW OPEN LOG command be immediately effective.

```
IMMEDIATE WINDOW OPEN LOG;
```

- Specify that the SCROLL BOTTOM command be immediately effective.

```
IMMEDIATE SCROLL BOTTOM;
```

INPUT Command (C and COBOL)

The INPUT command provides input for an intercepted read and is valid only when there is a read pending for an intercepted file. The INPUT keyword cannot be abbreviated.

```
►►──INPUT──text──;──────────────────────────────────────────────►◄
```

text
Specifies text input to a pending read.

Usage Notes:

- The INPUT text consists of everything between the INPUT keyword and the semicolon (or end-of-line). Any leading or trailing blanks are removed by Debug Tool.
- If a semicolon is included as part of the INPUT text, or if the first character of the INPUT text is a quote, the INPUT text must conform to the current programming language syntax for a character string constant (that is, enclosed in quotes, with internal quotes entered according to the rules of that programming language).
- This command is not supported for CICS.
- See “SET INTERCEPT (C and COBOL)” on page 311 for information about setting interception to and from a file.

Example:

You have used SET INTERCEPT ON to make Debug Tool prompt you for input to a sequential file. The prompt and the file's name appears in the Command Log.

Indicate that the phrase "quick brown fox" is input to a pending read. The phrase is written to the file.

```
INPUT quick brown fox;
```

Program input is recorded in your Log window.

A closing semicolon (;) is required for this command. Everything between the INPUT keyword and the semicolon is considered input text. If you want to include a semicolon in your input, or if the first character of your input is a quote, you must enter your input as a valid character string for your programming language.

LIST Command

The LIST command displays information about a program such as values of specified variables, structures, arrays, registers, statement numbers, frequency information, and the flow of program execution. The LIST command can be used to display information in any enclave. All information displayed will be saved in the log file.

The various forms of the LIST command are summarized in Table 12.

Table 12 (Page 1 of 2). Summary of LIST Commands

LIST (blank)	lists Source Identification panel
LIST AT	lists the currently defined breakpoints.
LIST CALLS	lists the dynamic chain of active blocks.
LIST CURSOR	lists the variable pointed to by the cursor.
LIST Expression	lists values of expressions.
LIST FREQUENCY	lists statement execution counts.
LIST LAST	lists a list of recent entries in the history table.
LIST LINE NUMBERS	lists all line numbers that are valid locations for an AT LINE breakpoint.
LIST LINES	lists one or more lines from the current listing or source file displayed in the Source window.
LIST MONITOR	lists the current set of MONITOR commands.
LIST NAMES	lists the names of variables, programs, or Debug Tool procedures.
LIST ON	lists the action (if any) currently defined for the specified PL/I conditions.
LIST PROCEDURES	lists the commands contained in the specified Debug Tool procedure.
LIST REGISTERS	lists the current register contents.

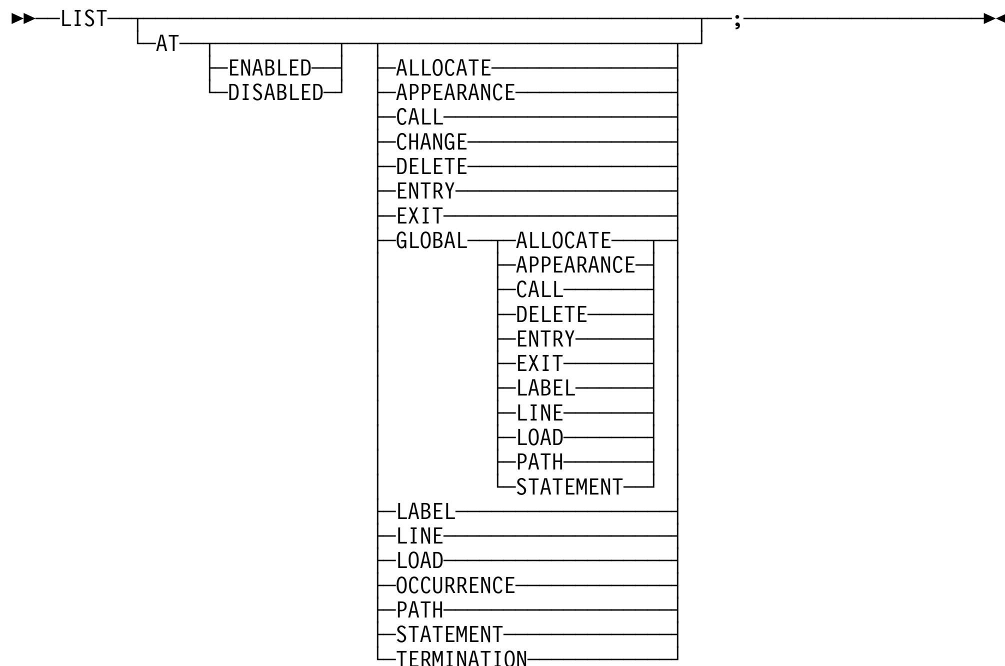
Table 12 (Page 2 of 2). Summary of LIST Commands	
LIST STATEMENT NUMBERS	lists all statement numbers that are valid locations for an AT STATEMENT breakpoint.
LIST STATEMENTS	lists one or more statements from the current listing or source file displayed in the Source window.
LIST STORAGE	provides a dump-format display of storage.

LIST (blank)

Displays the Source Identification Panel, where associations are made between source listings or source files shown in the Source window and their program units, LIST is equivalent to LISTINGS which is equivalent to SOURCES. See "PANEL Command (Full-Screen Mode)" on page 289 for additional information.

LIST AT

Lists the currently defined breakpoints, including the action taken when the specified breakpoint is activated.



AT_command
: The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted. See Table 9 on page 208 for a list of valid AT commands.

ENABLED
: Restricts the list to enabled breakpoints. The default is to list both enabled and disabled breakpoints.

DISABLED
: Restricts the list to disabled breakpoints. The default is to list both enabled and disabled breakpoints.

ALLOCATE
Lists currently defined AT ALLOCATE breakpoints.

APPEARANCE
Lists currently defined AT APPEARANCE breakpoints.

CALL
Lists currently defined AT CALL breakpoints.

CHANGE
Lists currently defined AT CHANGE breakpoints. This displays the storage address and length for all AT CHANGE subjects, and shows how they were specified (if other than by the %STORAGE function).

DELETE
Lists currently defined AT DELETE breakpoints.

ENTRY
Lists currently defined AT ENTRY breakpoints.

EXIT
Lists currently defined AT EXIT breakpoints.

GLOBAL
Lists currently defined AT GLOBAL breakpoints for the specified AT-condition.

LABEL
Lists currently defined AT LABEL breakpoints.

LINE
Lists currently defined AT LINE or AT STATEMENT breakpoints. LINE is equivalent to STATEMENT.

LOAD
Lists currently defined AT LOAD breakpoints.

OCCURRENCE
Lists currently defined AT OCCURRENCE breakpoints.

PATH
Lists currently defined AT PATH breakpoints.

STATEMENT
Is equivalent to LINE.

TERMINATION
Lists currently defined AT TERMINATION breakpoint.

If the AT command type (for example, LOAD) is not specified, LIST AT lists all currently defined breakpoints (both DISABLEd and ENABLEd).

Usage Note:

- To display a global breakpoint, you can specify an asterisk (*) with the LIST AT command or you can specify a LIST AT GLOBAL command. For example, if you want to display an AT ENTRY * breakpoint, specify:

  ```
  LIST AT ENTRY *;
  or
  LIST AT GLOBAL ENTRY;
  ```

  If you have only a global breakpoint set and you specify LIST AT ENTRY without the asterisk (*) or GLOBAL keyword, you get a message saying there are no such breakpoints.

Examples:

- Display information about enabled breakpoints defined at block entries.

  ```
  LIST AT ENABLED ENTRY;
  ```

- Display breakpoint information for all disabled AT CHANGE breakpoints within the currently executing program.

  ```
  LIST AT DISABLED CHANGE;
  ```

- The current programming language setting is C. Here are some assorted LIST AT commands.

  Display breakpoint information for any breakpoint set at line 22 of the program.

  ```
  LIST AT LINE 22;
  ```

  Display breakpoint information for the breakpoint set at occurrence of the SIGSEGV condition.

  ```
  LIST AT OCCURRENCE SIGSEGV;
  ```

  Display breakpoint information for the breakpoint set to activate on a change to the structure `structure.un.m`.

  ```
  LIST AT CHANGE structure.un.m;
  ```

LIST CALLS

Displays the dynamic chain of active blocks. For languages without block structure, this is the CALL chain.

```
►►──LIST──CALLS──;─────────────────────────────────────────────►◄
```

Usage Notes:

- For programs containing interlanguage communication (ILC), routines from previous enclaves are only listed if they are written in a language that is active in the current enclave.
- If the enclave was created with the `system()` function, compile units in parent enclaves will not be listed.

Example:

Display the current dynamic chain of active blocks.

```
LIST CALLS;
```

LIST CURSOR (Full-Screen Mode)

Provides a cursor controlled method for displaying variables, structures, and arrays. It is most useful when assigned to a PF key.

Usage Notes:

- Cursor pointing can be used by typing the LIST CURSOR command on the command line and moving the cursor to a variable in the source window before pressing Enter, or by moving the cursor and pressing a PF key with the LIST CURSOR command assigned to it.
- When using the LIST CURSOR command for a variable that is located by the cursor position, the variable's name cannot be split across different lines of the source listing.

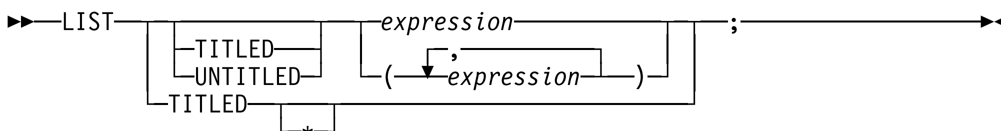
Example:

Display the value of the variable at the current cursor position.

```
LIST CURSOR
```

LIST Expression

Displays values of expressions.



TITLED
Displays each expression in the list with its value. For PL/I, this is the default. For C, this is the default for expressions that are *lvalues*. For COBOL, this is the default *except* for expressions consisting of only a single constant.

If TITLED is issued with no keyword specified, all variables in the currently qualified block are listed.

*** (C and COBOL)**
Lists all variables in the currently qualified compile unit.

UNTITLED
Lists expression values without displaying the expressions themselves. For C, this is the default for expressions that are not *lvalues*. For COBOL, this is the default for expressions consisting of only a single constant.

expression
A valid Debug Tool expression in the current programming language; see "Expression" on page 202.

- For the LIST command, an expression also includes character strings enclosed in either double (") or single (') quotes, depending on the current programming language.
- In C and COBOL, expressions containing parentheses () must be enclosed in another set of parentheses when used with the LIST command. For example, `LIST ((x + y) / z);`.
- In COBOL, an expression can be the GROUP keyword followed by a reference. If specified, the GROUP keyword causes the reference to be displayed as if it were an elementary item. If GROUP is specified for an elementary item, it has no effect. The operand of a GROUP keyword can only be a reference (expressions are not allowed). For example, `LIST TITLED GROUP y;`.

Usage Notes:

- If LIST TITLED * is specified and your compile unit is large, slow performance might result.
- When using LIST TITLED with no parameters within the PL/I compile unit, only the first element of any array will be listed. If the entire array needs to be listed, use LIST and specify the array name (that is, *LIST array*, where `array` is the name of an array).

Examples:

- Display the values for variables `size` and `r` and the expression `c + r`, with their respective names.

```
LIST TITLED (size, r, c + r);
```

- Display the COBOL references as if they were elementary items. The current programming language setting is COBOL.

```
LIST (GROUP x OF z(1,2), GROUP a, w);
```

- Display the value of the Debug Tool variable %ADDRESS.

```
LIST %ADDRESS;
```

LIST FREQUENCY

Lists statement execution counts.

```
►►──LIST──FREQUENCY──┬─statement_id_range──────────┬──;──────────────►◄
                     │    ┌─,──────────────────┐   │
                     ├─(──▼─statement_id_range─┴──)─┤
                     └─*────────────────────────────┘
```

statement_id_range
A valid statement id or statement id range; see "Statement_Id_Range and Stmt_Id_Spec" on page 204.

* Lists frequency for all statements in the currently qualified compile unit. If currently executing at the AT Termination breakpoint where there is no qualification available, it will list frequency for all statements in all the compile units in the terminating enclave where frequency data exists.

Note: See also "SET FREQUENCY" on page 310.

Examples:

- List frequency for statements 1-20.

```
LIST FREQUENCY 1 - 20;
```

- List frequency for all statements in the currently qualified compile unit.

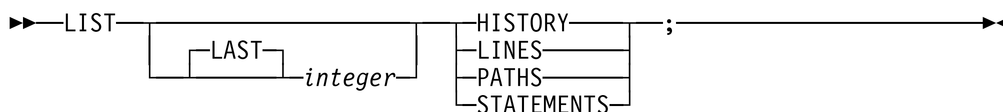
```
LIST FREQUENCY *;
```

- List frequency for all statements in all compilation units.

```
AT TERMINATION LIST FREQUENCY *;
```

LIST LAST

Displays a list of recent entries in the history table.



integer
Specifies the number of most recently processed breakpoints and conditions displayed.

HISTORY
Displays all processed breakpoints and conditions.

LINES
Displays processed statement or line breakpoints. LINES is equivalent to STATEMENTS.

PATHS
Displays processed path breakpoints.

STATEMENTS
Is equivalent to LINES.

Note: See also "SET HISTORY" on page 311.

Examples:

- Display all processed path breakpoints in the history table.

```
LIST PATHS;
```

- Display all program breakpoints and conditions for the last five times Debug Tool gained control.

```
LIST LAST 5 HISTORY;
```

LIST LINE NUMBERS

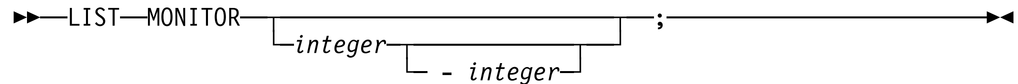
See "LIST STATEMENT NUMBERS" on page 282.

LIST LINES

See "LIST STATEMENTS" on page 283.

LIST MONITOR

Lists all or selected members of the current set of MONITOR commands.



integer
An unsigned integer identifying a MONITOR command. If two integers are specified, the first must not be greater than or equal to the second. If omitted, all MONITOR commands are displayed.

Usage Note:

- When the current programming language setting is COBOL, blanks are required around the hyphen (-). Blanks are optional for C.

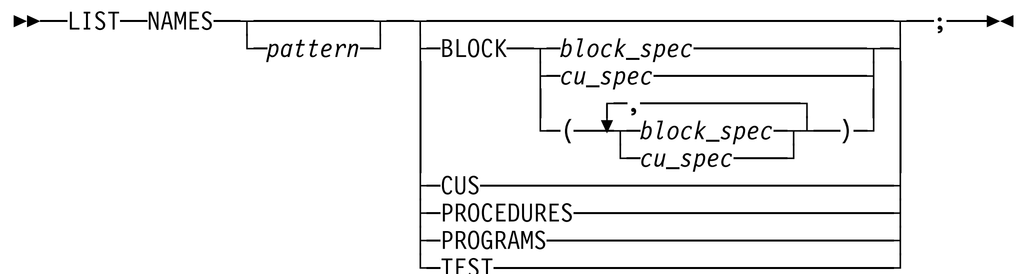
Example:

List the fifth through the seventh commands currently being monitored.

```
LIST MONITOR 5 - 7;
```

LIST NAMES

Lists the names of variables, programs, or Debug Tool procedures. If LIST NAMES is issued with no keyword specified, the names of all program and session variables that can be referenced in the current programming language and that are visible to the currently qualified block are displayed. A subset of the names can be specified by supplying a pattern to be matched.



pattern
The pattern searched for, conforming to the current programming language syntax for a character string constant. The pattern length cannot exceed 128 bytes, excluding the quotes.

If the DBCS setting is ON, the pattern can contain DBCS characters. DBCS shift codes are not considered significant characters in the pattern. Within the pattern, an SBCS or DBCS asterisk represents a string of zero or more

insignificant SBCS or DBCS characters. As many as eight asterisks can be included in the pattern, but adjacent asterisks are equivalent to a single asterisk.

Some examples of possible strings follow:

C	COBOL	PL/I
"ABC"	"A5" 'A5'	'MY'

Pattern matching is not case-sensitive outside of DBCS. Both the pattern and potential names outside of shift codes are effectively uppercased, except when the current programming language setting is C. Letters in the pattern must be the correct case when the current programming language setting is C.

BLOCK
Displays variable names that are defined within one or more specified blocks.

block_spec
A valid block specification; see "Block_Spec" on page 200.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. *cu_spec* can be used to list the variable and function names that are defined within the specified compile unit.

CUS
Displays the compile unit names. CUS is equivalent to PROGRAMS.

PROCEDURES
Displays the Debug Tool procedure names.

PROGRAMS
Is equivalent to CUS.

TEST
Displays the Debug Tool session variable names.

Usage Notes:

- LIST NAMES CUS applies to compile unit names.
- LIST NAMES TEST shows only those session variable names that can be referenced in the current programming language.
- The output of LIST NAMES without any options depends on both the current qualification and the current programming language setting. If the current programming language differs from the programming language of the current qualification, the output of the command shows only those session variable names that can be referenced in the current programming language.
- For structures, the pattern is tested against the complete name, hence "B" is not satisfied by "C OF B OF A" (COBOL).

Examples:

- Display all compile unit names that begin with the letters "MY" and end with "5". The current programming language setting is either C or COBOL.

  ```
  LIST NAMES "MY*5" PROGRAMS;
  ```

- Display the names of all the Debug Tool procedures that can be invoked.

  ```
  LIST NAMES PROCEDURES;
  ```

- Display the names of variables whose names begin with 'R' and are in the mainprog block. The current programming language setting is COBOL.

  ```
  LIST NAMES 'R*' BLOCK (mainprog);
  ```

LIST ON (PL/I)

Lists the action (if any) currently defined for the specified PL/I conditions.

```
►►──LIST──ON──┬───────────────┬──;─────────────────────────────►◄
              └─pli_condition─┘
```

pli_condition
A valid PL/I condition specification. If omitted, all currently defined ON command actions are listed. See “ON Command (PL/I)” on page 287.

Example:

List the action for the ON ZERODIVIDE command.

```
LIST ON ZERODIVIDE;
```

LIST PROCEDURES

Lists the commands contained in the specified Debug Tool PROCEDURE definitions.

```
►►──LIST──PROCEDURES──┬────────────────────┬──;────────────────►◄
                      ├─name───────────────┤
                      │     ┌─,────┐       │
                      └─(───▼─name─┴──)────┘
```

name
A valid Debug Tool procedure name. If no procedure name is specified, the commands contained in the currently running procedure are displayed. If no procedure is currently running, an error message is issued.

Examples:

- Display the commands in the Debug Tool procedure p2.

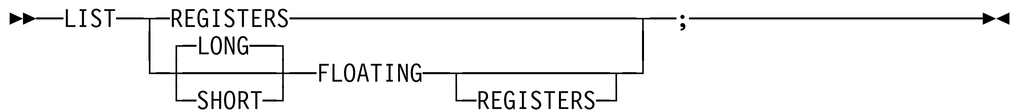
```
LIST PROC p2;
```

- List the procedures abc and proc7.

```
LIST PROCEDURES (abc, proc7);
```

LIST REGISTERS

Displays the current register contents.



REGISTERS
Displays the general-purpose registers

LONG
Displays the decimal value of the long-precision floating-point registers.

SHORT
Displays the decimal value of the short-precision floating-point registers.

FLOATING
Displays the long-precision floating-point registers.

Examples:

- Display the general-purpose registers at the point of a program interruption:

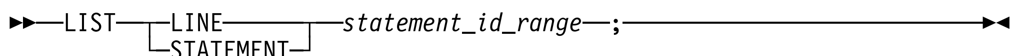
```
LIST REGISTERS;
```

- Display the floating-point registers.

```
LIST FLOATING REGISTERS;
```

LIST STATEMENT NUMBERS

Lists all statement or line numbers that are valid locations for an AT LINE or AT STATEMENT breakpoint.



NUMBERS
Displays the statement numbers that can be used to set STATEMENT breakpoints, assuming the compile options used to generate statement hooks were specified at compile time. The list can also be used for the GOTO command, however, you might not be able to GOTO all of the statement numbers listed.

block_spec
A valid block specification; see "Block_Spec" on page 200. This operand lists all statement or line numbers in the specified block.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. For C programs, *cu_spec* can be used to list the statement numbers that are defined within the specified compile unit before the first function definition.

statement_id_range
A valid statement id or statement id range; see "Statement_Id_Range and Stmt_Id_Spec" on page 204.

Examples:

- List the statement or line numbers in the currently qualified block.

```
LIST STATEMENT NUMBERS;
```

- Display the statement or line number of every statement in block `earnings`.

```
LIST STATEMENT NUMBERS earnings;
```

LIST STATEMENTS

Lists one or more statements or lines from a source file. It can be used in full-screen mode to copy a portion of a source listing or source file to the log.

```
►►──LIST──┬─LINE──────┬──statement_id_range──;──────────────────►◄
          └─STATEMENT─┘
```

statement_id_range
A valid statement id or statement id range in the same block or different blocks; see "Statement_Id_Range and Stmt_Id_Spec" on page 204.

Usage Notes:

- The specified lines are displayed in the same format as they would appear in the full-screen Source window, except that wide lines are truncated.
- You might need to specify a range of line numbers to ensure that continued statements are completely displayed.
- This command is not to be confused with the LIST LAST STATEMENTS command.

Examples:

- List lines 25 through 30 in the source file associated with the currently qualified compile unit.

```
LIST LINES 25 - 30;
```

- List statement 100 from the current program listing file.

```
LIST STATEMENT 100;
```

LIST STORAGE

Displays the contents of storage at a particular address in hex format.

```
►►──LIST──STORAGE──┬─────────────────────────────────────────┬──;──────►◄
                   └─(──┬─address───┬──┬──────────────┬──)─┘
                        └─reference─┘  └─,──integer─┘
```

address
The starting address of storage to be watched for changes. This must be a hex constant: *0x* in C, *H* in COBOL (using either double (") or single (') quotes), or *PX* in PL/I.

reference
A valid Debug Tool reference in the current programming language; see "References" on page 203.

For C, if the referenced variable is an array, Debug Tool displays the storage at the address of that array. However, if the referenced variable is a pointer, Debug Tool displays the storage at the address given by that pointer.

integer
The number of bytes of storage displayed. The default is 16 bytes.

Usage Notes:

- Using Debug Tool, cursor pointing can be used by typing the LIST STORAGE command on the command line and moving the cursor to a variable in the Source window before pressing Enter, or by moving the cursor and pressing a PF key with the LIST STORAGE command assigned to it.
- When using the LIST STORAGE command in Debug Tool for a variable that is located by the cursor position, the variable's name cannot be split across different lines of the source listing.
- If no operand is specified with LIST STORAGE, the command is cursor-sensitive.

Examples:

- Display the first 64 bytes of storage beginning at the address of variable `table`.

```
LIST STORAGE (table, 64);
```

- Display 16 bytes of storage at the address given by pointer `table(1)`.

```
LIST STORAGE (table(1));
```

- Display the 16 bytes contained at locations 20CD0-20CDF. The current programming language setting is COBOL.

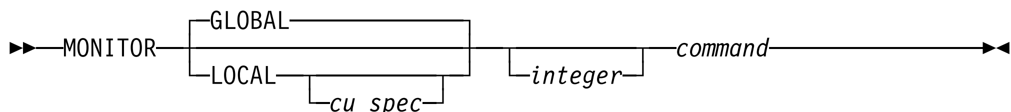
```
LIST STORAGE (H'20CD0');
```

- Display the 16 bytes contained at locations 20CD0-20CDF. The current programming language setting is PL/I.

```
LIST STORAGE ('20CD0'PX);
```

MONITOR Command

The MONITOR command defines or redefines a command whose output is displayed in the Monitor window (full-screen mode), or log file (batch mode). Only DESCRIBE, LIST, Null, and QUERY command values are maintained.



GLOBAL
Specifies that the monitor definition is global. That is, it is not associated with a particular compile unit.

LOCAL
Specifies that the monitor definition is local to a specific compile unit. Using Debug Tool, the specified output is displayed only when the current qualification is within the associated compile unit.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. This specifies the compile unit associated with the monitor definition. The default is the currently qualified compile unit.

integer
An integer in the range 1 to 99, indicating what command in the list is replaced with the specified command and the order that the monitored commands are evaluated. If omitted, the next monitor integer is assigned. An error message is displayed if the maximum number of monitoring commands already exists.

command
A DESCRIBE, LIST, Null, or QUERY command whose output is displayed in the Monitor window, or log file.

Usage Notes:

- A monitor number identifies a global monitor command, a local monitor command, or neither.
- Using Debug Tool, monitor output is presented in monitor number sequence.
- If a number is provided and a command omitted, a null command is inserted on the line corresponding to the number in the Monitor window. This reserves the monitor number.
- You can only specify a monitor number that is at most one greater than the highest existing monitor number.
- The MONITOR command displays up to a maximum of 1000 lines of output in the Monitor window.
- Replacement only occurs if the command identified by the monitor number already exists.
- The MONITOR LIST command does not allow the TITLED, and UNTITLED options.
- When using the MONITOR LIST command, simple references (or C *lvalues*) display identifying information with the values, whereas expressions and literals do not.
- The GLOBAL and LOCAL keywords also affect the default qualification for evaluation of an expression. GLOBAL indicates that the default qualification is the currently executing point in the program. LOCAL indicates that the default qualification is to the compile unit specified.

Examples:

- Replace the 10th command in the monitor list with QUERY LOCATION. This is a global definition; therefore, it is always present in the monitor output.

```
MONITOR 10 QUERY LOCATION;
```

- Add a monitor command that displays the variable `abc` and is local to compile unit `myprog`. The monitor number is the next available number.

```
MONITOR LOCAL myprog LIST abc;
```

MOVE Command (COBOL)

The MOVE command transfers data from one area of storage to another. The keywords cannot be abbreviated.

```
►►──MOVE──┬─reference─┬──TO──reference──;─────────────────────────────►◄
          └─literal───┘
```

reference
A valid Debug Tool COBOL reference.

literal
A valid COBOL literal.

Usage Notes:

- If Debug Tool was invoked because of a computational condition or an attention interrupt, using an assignment to set a variable might not give expected results. This is due to the uncertainty of variable values within statements as opposed to their values at statement boundaries.
- MOVE assigns a value only to a single receiver; unlike COBOL, multiple receiver variables are not supported.
- The COBOL CORRESPONDING phrase is not supported.
- Only the sender/receiver combinations listed in Appendix D, "Using COBOL Reference Information with Debug Tool" on page 349, are supported.

Examples:

- Move the string constant "Hi There" to the variable `field`.

```
MOVE "Hi There" TO field;
```

- Move the value of session variable `temp` to the variable `b`.

```
MOVE temp TO b;
```

- Move a DBCS value to the DBCS variable `dbcs_field`.

```
MOVE G"D B C S  V A L U E" TO dbcs_field;
```

Null Command

The Null command is a semicolon written where a command is expected. It is used for such things as an IF command with no action in its THEN clause.

```
►►──;──────────────────────────────────────────────────────────────►◄
```

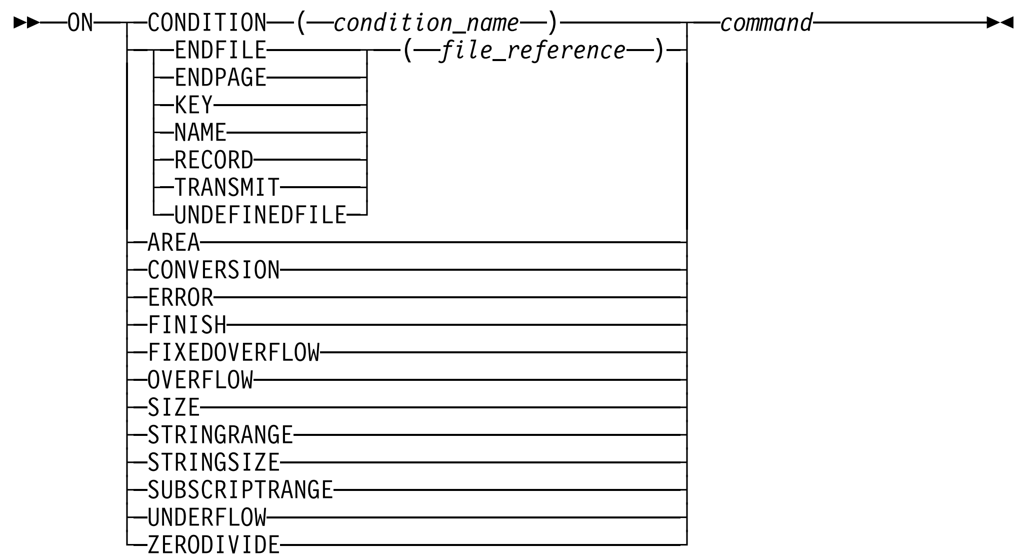
Example:

Do nothing if `array[x]` > 0, otherwise set `a` to 1. The current programming language setting is C.

```
if (array[x] > 0); else a = 1;
```

ON Command (PL/I)

The ON command establishes the actions to be executed when the specified PL/I condition is raised. This command is equivalent to AT OCCURRENCE; see page 225.



condition_name
A valid PL/I CONDITION condition name.

file_reference
A valid PL/I file constant or file variable (can be qualified).

command
A valid Debug Tool command.

Usage Notes:

- You must abide by the PL/I restrictions for the particular condition. See *IBM PL/I for VSE/ESA Language Reference* for an explanation of the restrictions.
- An ON action for a specified PL/I condition remains established until:
  - Another ON command establishes a new action for the same condition. In other words, the breakpoint is replaced.
  - A CLEAR command removes the ON definition.
- The ON command occurs before any existing ON-unit in your application program. The ON-unit is processed after Debug Tool returns control to the language.
- The following are accepted PL/I abbreviations for the PL/I condition constants:

  FIXEDOVERFLOW or FOFL
  OVERFLOW or OFL
  STRINGRANGE or STRG
  STRINGSIZE or STRZ
  SUBSCRIPTRANGE or SUBRG
  UNDEFINEDFILE([file_reference]) or UNDF([file_reference])
  UNDERFLOW or UFL
  ZERODIVIDE or ZDIV

- The preferred form of the ON command is AT OCCURRENCE, however, ON is recognized and processed. ON should be considered a synonym of AT OCCURRENCE. Any ON commands entered are logged as AT OCCURRENCE commands.

Examples:

- Display a message if a division by zero is detected.

```
ON ZERODIVIDE BEGIN;
  LIST 'A zero divide has been detected';
END;
```

- Display and patch the error character when converting character data to numeric.

  Given a PL/I program that contains the following statements:

```
DECLARE i FIXED BINARY(31,0);
  .
  .
  .
i = '1s3';
```

  The following Debug Tool command would display and patch the error character when converting the character data to numeric:

```
ON CONVERSION
  BEGIN;
    LIST (%STATEMENT, ONCHAR);
    ONCHAR = '0';
    GO;
  END;
```

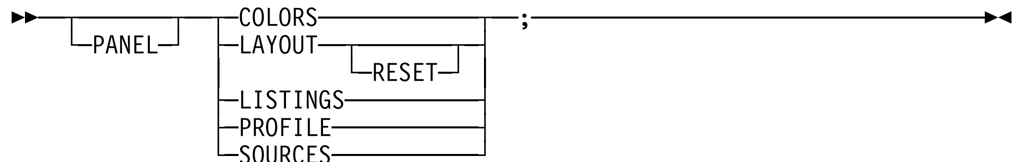
  '1s3' cannot be converted to a binary number so CONVERSION is raised. The ON CONVERSION command lists the offending statement

number and the offending character: `'s'`. The data will be patched by replacing the `'s'` with a character zero, `0`, and processing will continue.

PANEL Command (Full-Screen Mode)

The PANEL command displays special panels. The PANEL keyword is optional.

The PANEL command cannot be used in a command list, any conditional command, or any multiway command.



COLORS
Displays the Color Selection Panel that allows the selection of color, highlighting, and intensity of the various fields of the Debug Tool session panel.

LAYOUT
Displays the Window Layout Selection Panel that controls the configuration of the windows on the Debug Tool session panel.

RESET
Restores the relative sizes of windows for the current configuration, without displaying the window layout panel. For configurations 1 and 4, the three windows are evenly divided. For other configurations, the point where the three windows meet is approximately the center of the screen.

See "Changing Session Panel Window Layout" on page 95 for details on the six configuration options.

LISTINGS
Displays the Source Identification Panel, where associations are made between source listings or source files shown in the Source window and their program units. LISTINGS is equivalent to SOURCES.

Debug Tool provides the Source Identification Panel to maintain a record of compile units associated with your program, as well as their associated source or listing.

You can also make source or listings available to Debug Tool by entering their names on the Source Identification Panel.

The Source Identification Panel associates compile units with the names of their respective listing or source files and controls what appears in the Source window. To explicitly name the compile units being displayed in the Source window, access the Source Identification Panel by entering the PANEL LISTINGS or PANEL SOURCES command. Figure 33 on page 290 is an example of the panel.

```
                   Source Identification Panel
Command ===>

Compile Unit           Listings/Source File             Display
---------------------- -------------------------------- -------
DBKP515                (DBKP515.LIST)                      Y
___________            ____________________________        _

Enter QUIT      to return with current settings saved.
      CANCEL    to return without current settings saved.
      UP/DOWN   to scroll up and down.
```

Figure 33. The Source Identification Panel

Compile Unit
Is the name of a valid compile unit currently known to Debug Tool. New compile units are added to the list as they become known.

Listing/Source File
Is the name of the listing or source file containing the compilation unit to be displayed in the Source window. If the file is a listing, only source program statements are shown. The minimum required is the compile unit name. The default file specification is `pgmname.LIST` (COBOL and PL/I) or `pgmname.C`, where `pgmname` is the name of your program.

Display
Is a flag that specifies whether the listing or source is to be displayed in the Source window.

To display a listing view, take the following steps:

- For a C program, catalog the source program in a sublibrary or write the source program to a SAM ESDS file or a sequential disk file. When you compile the C program, you should use the compile-time INFILE option to specify this sublibrary member name or filename.
- For a COBOL or PL/I program, compile the program and write the compile-time listing to a sublibrary member, a SAM ESDS file, or a sequential disk file.
- Make sure the source or listing file is available and accessible to Debug Tool.
- Set the **Display** field on the Source Identification panel to Y for the compile unit. To save time and avoid displaying listings or source you do not want to see, specify N.

If any of these conditions are not satisfied, the Source window remains empty until control reaches a compile unit where the conditions are satisfied.

You can change the source or source listing associated with a compile unit by entering the new name over the source or source listing file displayed in the LISTING/SOURCE FILE field.

Note: The new name must be followed by at least one blank.

After you modify the panel, return to the Debug Tool session panel either by issuing the QUIT command, or by pressing the QUIT PF key.

PROFILE
Displays the Profile Settings Panel, where parameters of a full-screen Debug Tool session can be set.

SOURCES
Is equivalent to LISTINGS.

Usage Notes:

- All information on the panels invoked by the PANEL command is saved when QUIT is used to leave them. Saving the changes to the specified panels in this manner returns you to your Debug Tool session with the current settings in effect. In addition, `CANCEL` can be used to leave the panels without saving the changes.
- On normal termination, Debug Tool saves certain panel settings in a sublibrary member *userid*.SAFE. See “Customizing Settings” on page 99 for details on changing and saving the settings for each of the panels.
- The PANEL command is not logged.

Examples:

- Display the color and attribute panel.

```
PANEL COLORS;
```

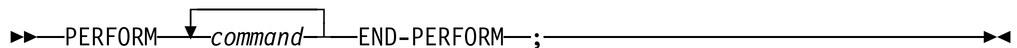
- Reset the relative sizes of the windows for the current layout configuration.

```
PANEL LAYOUT RESET;
```

PERFORM Command (COBOL)

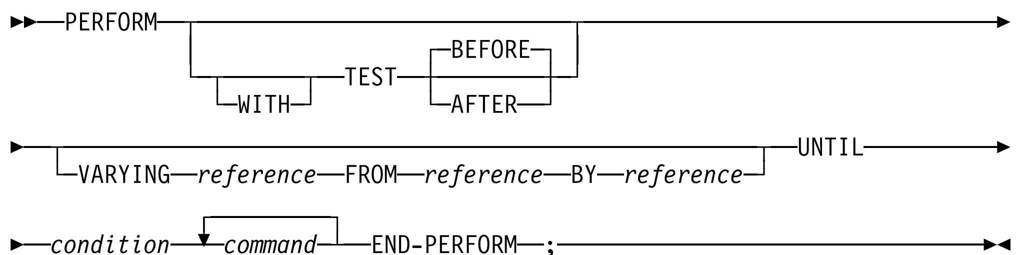
The PERFORM command transfers control explicitly to one or more statements and implicitly returns control to the next executable statement after execution of the specified statements is completed. The keywords cannot be abbreviated.

Simple:



command
A valid Debug Tool command.

Repeating:



reference
A valid Debug Tool COBOL reference.

condition
A simple relation condition.

command
A valid Debug Tool command.

Usage Notes:

- A constant as a *reference* is allowed only on the right side of the FROM and BY keywords.
- Index-names and floating point variables cannot be used as the VARYING *references*.
- Index-names are not supported in the BY phrase.
- Only in-line PERFORMs are supported (but the PERFORMed command can be a Debug Tool procedure invocation).
- The COBOL AFTER phrase is not supported.
- See *IBM COBOL for VSE/ESA Language Reference* for an explanation of the following COBOL keywords:

  AFTER
  BEFORE
  BY
  FROM
  TEST
  UNTIL
  VARYING
  WITH

Examples:

- Set a breakpoint at statement number 10 to move the value of variable `a` to the variable `b` and then list the value of `x`.

```
AT 10 PERFORM
  MOVE a TO b;
  LIST (x);
END-PERFORM;
```

- List the value of `height` for each even value between 2 and 30, including 2 and 30.

```
PERFORM WITH TEST AFTER
  VARYING height FROM 2 BY 2
    UNTIL height = 30
      LIST height;
END-PERFORM;
```

Prefix Commands (Full-Screen Mode)

The Prefix commands apply only to source listing lines and are typed into the prefix area in the Source window. For details, see the section corresponding to the command name.

The various forms of the Prefix commands are summarized in Table 13 on page 293.

Table 13. Summary of Prefix Commands	
AT Prefix	defines a statement breakpoint via the Source window prefix area.
CLEAR Prefix	clears a breakpoint via the Source window prefix area.
DISABLE Prefix	disables a breakpoint via the Source window prefix area.
ENABLE Prefix	enables a disabled breakpoint via the Source window prefix area.
QUERY Prefix	queries what statements have breakpoints via the Source window prefix area.
SHOW Prefix	specifies what relative statement or verb within the line is to have its frequency count shown in the suffix area.

PROCEDURE Command

The PROCEDURE command allows the definition of a group of commands that can be accessed using the CALL procedure command. The CALL command is the only way to perform the commands within the PROCEDURE. PROCEDURE definitions remain in effect for the entire debug session.

The PROCEDURE keyword can only be abbreviated as PROC. PROCEDURE definitions can be subcommands of other PROCEDURE definitions. The name of a nested procedure has only the scope of the containing procedure. Session variables cannot be declared within a PROCEDURE definition.

In addition, a procedure must be defined before it is CALLed.

```
              ┌─────────┐
►►─name─:─PROCEDURE─;──▼─command─┴──END─;───────────────────►◄
```

name
A valid Debug Tool procedure name. It must be a valid identifier in the current programming language. The maximum length is 31 characters.

command
A valid Debug Tool command other than a declaration or PANEL command.

Usage Notes:

- Since the Debug Tool procedure names are always uppercase, the procedure names are converted to uppercase even for programming languages that have mixed-case symbols.
- If a GO or STEP command is issued within a procedure or a nested procedure, any statements following the GO or STEP in that procedure and the containing procedure are ignored. If control returns to Debug Tool, it returns to the statement following the CALL of the containing PROCEDURE.
- It is recommended that procedure names be chosen so that they are valid for all possible programming language settings throughout the entire Debug Tool debugging session.

Examples:

- When procedure proc1 is called, the values of variables x, y, and z are displayed.

```
proc1: PROCEDURE; LIST (x, y, z); END;
```

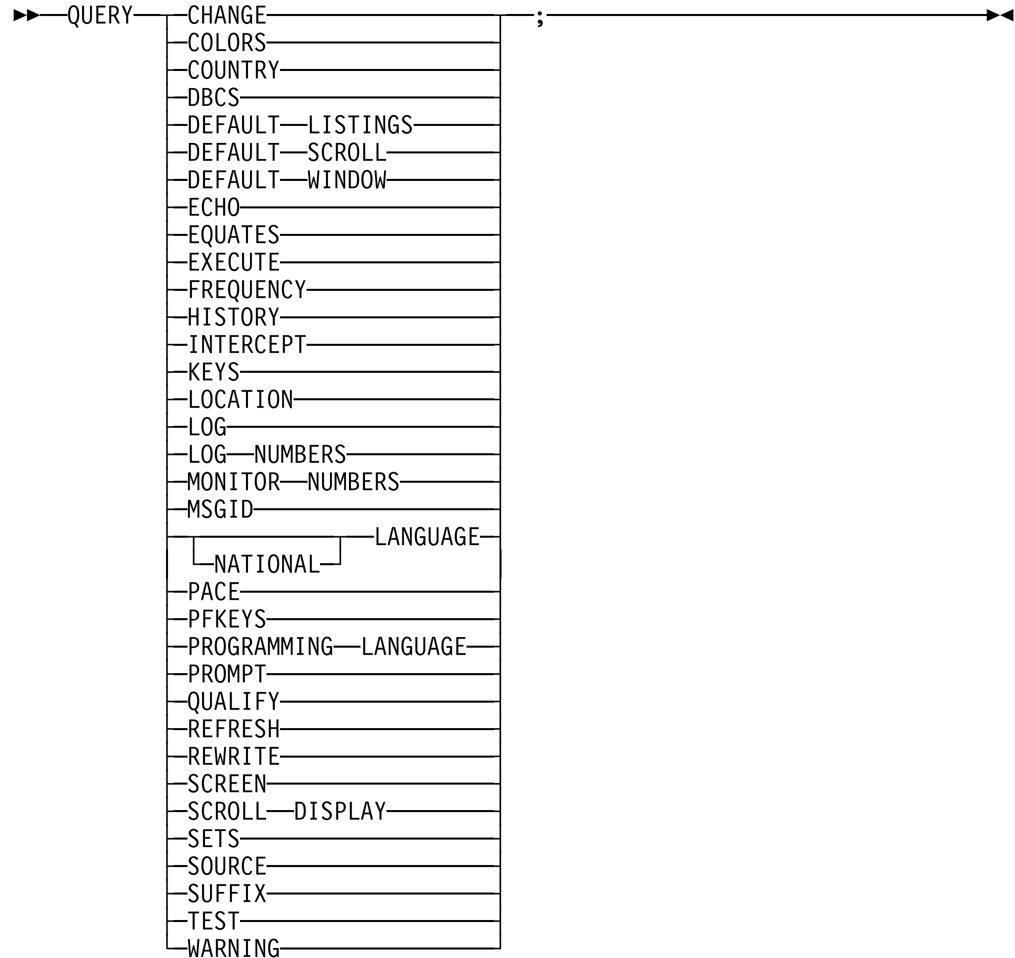
- Define a procedure named setat34 that sets a breakpoint at statement 34. Procedure setat34 contains a nested procedure lister that lists current statement breakpoints. Procedure lister can only be called from within setat34.

```
setat34: PROCEDURE;
  AT 34;
  lister: PROCEDURE;
    LIST AT STATEMENT;
  END;
  CALL lister;
END;
```

QUERY Command

The QUERY command displays the current value of the specified Debug Tool setting, the current setting of all the Debug Tool settings, or the current location in the suspended program.

For an explanation of the Debug Tool settings, see the SET command ("SET Command" on page 302).



CHANGE
Displays the current CHANGE setting.

COLORS (Full-Screen Mode)
Displays the current COLOR setting.

COUNTRY
Displays the current COUNTRY setting.

DBCS
Displays the current DBCS setting.

DEFAULT LISTINGS
Displays the current DEFAULT LISTINGS setting.

DEFAULT SCROLL (Full-Screen Mode)
Displays the current DEFAULT SCROLL setting.

DEFAULT WINDOW (Full-Screen Mode)
Displays the current DEFAULT WINDOW setting.

ECHO
Displays the current ECHO setting.

EQUATES
Displays the current EQUATE definitions.

EXECUTE
Displays the current EXECUTE setting.

FREQUENCY
Displays the current FREQUENCY setting.

HISTORY
Displays the current HISTORY setting and size.

INTERCEPT
Displays the current INTERCEPT setting.

KEYS (Full-Screen Mode)
Displays the current KEYS setting.

LOCATION
Displays the statement identifier where execution is suspended. The current statement identified by QUERY LOCATION has not yet executed. If suspended at a breakpoint, the description of the breakpoint is also displayed.

LOG
Displays the current LOG setting.

LOG NUMBERS (Full-Screen Mode)
Displays the current LOG NUMBERS setting.

MONITOR NUMBERS (Full-Screen Mode)
Displays the current MONITOR NUMBERS setting.

MSGID
Displays the current MSGID setting.

NATIONAL LANGUAGE
Displays the current NATIONAL LANGUAGE setting.

PACE (Full-Screen Mode)
Displays the current PACE setting.

PFKEYS (Full-screen Mode)
Displays the current PFKEY definitions.

PROGRAMMING LANGUAGE
Displays the current PROGRAMMING LANGUAGE setting.

PROMPT (Full-Screen Mode)
Displays the current PROMPT setting.

QUALIFY
Displays the current QUALIFY BLOCK setting.

REFRESH (Full-Screen Mode)
Displays the current REFRESH setting.

REWRITE
Displays the current REWRITE setting. This setting is not supported in batch mode.

SCREEN (Full-Screen Mode)
Displays the current SCREEN setting.

SCROLL DISPLAY (Full-Screen Mode)
Displays the current SCROLL DISPLAY setting.

SETS
Displays all current settings.

SOURCE
Displays the current SOURCE setting.

SUFFIX (Full-Screen Mode)
Displays the current SUFFIX setting.

TEST
Displays the current TEST setting.

WARNING (C)
Displays the current WARNING setting.

Examples:

- Display the current ECHO setting.

```
QUERY ECHO;
```

- Display all current settings.

```
QUERY SETS;
```

QUERY Prefix (Full-Screen Mode)

Queries what statements on a particular line have statement breakpoints when you issue this command via the Source window prefix area.

```
►►──QUERY──────────────────────────────────────────────────────►◄
```

Usage Notes:

- When the QUERY Prefix command is issued, a sequence of characters corresponding to the statements is displayed in the prefix area of the Source window. If the statement contains a breakpoint, "*" is used, or ".", if it does not. If there are more than eight statements or verbs on the line, and one or more past the eighth statement have breakpoints, the eighth character of the map is replaced by a "+".

  For example, a display of "..*." would indicate that four statements or verbs begin on the line and the third one has a breakpoint defined.

- The QUERY Prefix command is not logged.

QUIT Command

The QUIT command ends a Debug Tool session and terminates your application. If an expression is specified, Debug Tool sets the return code. In Full-Screen mode QUIT also invokes a prompt panel (full-screen) that asks if you really want to quit the debug session. In batch mode, the QUIT command ends the session without prompting.

```
►►──QUIT──┬──────────────────┬──;──────────────────────────────►◄
          └─(──expression──)─┘
```

expression
A valid Debug Tool expression in the current programming language; see “Expression” on page 202.

If *expression* is specified, this value is used as the application return code value. The actual return code for the run is determined by the execution environment.

Usage Notes:

- QUIT is always logged in a comment line. This makes it unnecessary for you to "comment out" the QUIT to reuse the log file as a primary commands file.
- If QUIT is issued from a Debug Tool commands file, no prompt is issued. This applies to the Debug Tool preferences file, primary commands file, and USE files.
- For PL/I, the expression will be converted to FIXED BINARY (31,0), if necessary. In addition, if an expression is specified, it is used as if there was an invocation of the PLIRETC built-in subroutine in your program.
- For PL/I, the value of the expression must be nonnegative and less than 1000.

Examples:

- End a Debug Tool session.

```
QUIT;
```

- End a Debug Tool session and use the value in variable x as the application return code.

```
QUIT (x);
```

RETRIEVE Command (Full-Screen Mode)

The RETRIEVE command displays the last command entered on the command line. For long commands this may only be the last line of the command.

```
              ┌─COMMAND─┐
►►──RETRIEVE──┴─────────┴──;──────────────────────────────►◄
```

COMMAND
Retrieves commands. Any command retrieved to the command line can be performed by pressing Enter. The retrieved command can also be modified before it is performed. Successive RETRIEVE commands continue to display up to 12 commands previously entered on the command line. This operand is most useful when assigned to a PF key.

Usage Note: The RETRIEVE command is not logged.

Example:

Retrieve the last line so that it can be reissued or modified.

```
RETRIEVE COMMAND;
```

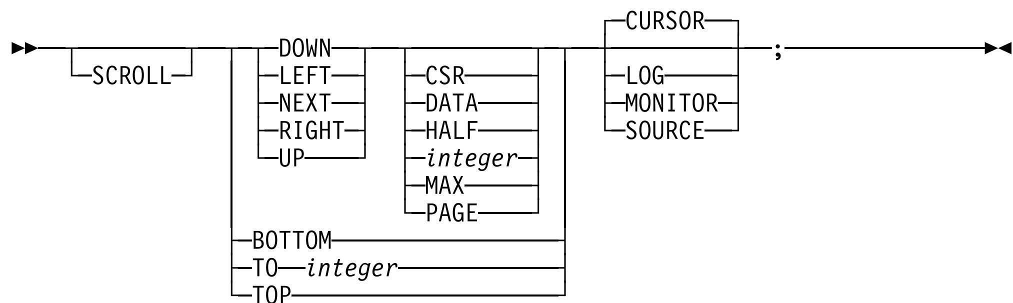
RUN Command

The RUN command is synonymous to the GO command. See "GO Command" on page 265.

SCROLL Command (Full-Screen Mode)

The SCROLL command provides horizontal and vertical scrolling in full-screen mode. Scroll commands can be made immediately effective with the IMMEDIATE command. The SCROLL keyword is optional.

The Log, Monitor, or Source window will not wrap around when scrolled.



DOWN
Scrolls the specified number of lines in a window toward the top margin of that window. DOWN is equivalent to NEXT.

LEFT
Scrolls the specified number of columns in a window toward the right margin of that window.

NEXT
Is equivalent to DOWN.

RIGHT
Scrolls the specified number of columns in a window toward the left margin of that window.

UP
Scrolls the specified number of lines in a window toward the bottom margin of that window.

CSR
Specifies scrolling based on the current position of the cursor in a selected window. The window scrolls up, down, left, or right of the cursor position until the character where the cursor is positioned reaches the edge of the window.

If the cursor is not in a window or if it is already positioned at the edge of a window, a full-page scroll occurs.

DATA
Scrolls by one line less than the window size or by one character less than the window size (if moving left or right).

HALF
Scrolls by half the window size.

integer
Scrolls the specified number of lines (up or down) or the specified number of characters (left or right). Maximum value is 9999.

MAX
Scrolls in the specified direction until the limit of the data is reached. To scroll the maximum amount, you must use the MAX keyword. You cannot scroll the maximum amount by filling in the scroll amount field.

PAGE
Scrolls by the window size.

BOTTOM
Scrolls to the bottom of the data.

TO *integer*
Specifies that the selected window is to scroll to the given line (as indicated in the prefix area of the selected window). This can be in either the UP or DOWN direction (for example, if you are line 30 and issue "TO 20", it will return to line 20). Maximum value is 999999.

TOP
Scrolls to the top of the data.

CURSOR
Selects the window where the cursor is currently positioned.

LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the source listing window.

Usage Notes:

- If you do not specify an operand with the DOWN, LEFT, NEXT, RIGHT, or UP keywords, and the cursor is outside the window areas, the window scrolled is determined by the current default window setting (if the window is open) and the scroll amount is determined by the current default scroll setting, shown in the SCROLL field on the Debug Tool session panel. Default scroll and default window settings are controlled by SET DEFAULT SCROLL and SET DEFAULT WINDOW commands.
- When the SCROLL field on the Debug Tool session panel is overtyped, the equivalent SET DEFAULT SCROLL command is issued just as if you had typed the command in directly from the command line (that is, it is logged and retrievable).

- The SCROLL command is not logged.
- See also "SET DEFAULT SCROLL (Full-Screen Mode)" on page 307.

Examples:

- Scroll one page down in the window containing the cursor.

```
SCROLL DOWN PAGE CURSOR;
```

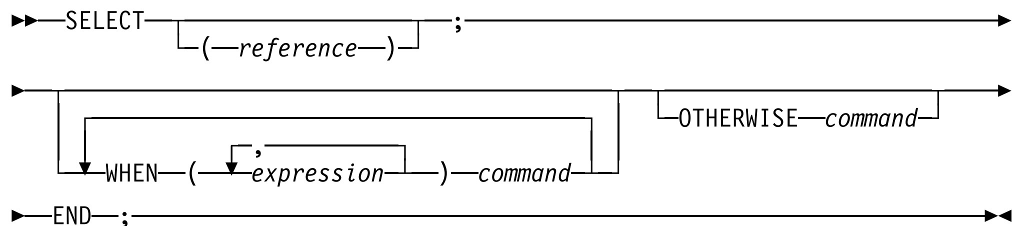
- Scroll the Monitor window 12 columns to the left.

```
SCROLL LEFT 12 MONITOR;
```

SELECT Command (PL/I)

The SELECT command chooses one of a set of alternate commands.

If the reference can be satisfied by more than one of the WHEN clauses, only the first one is performed. If there is no reference, the first WHEN clause containing an expression that is true is executed. If none of the WHEN clauses are satisfied, the command specified on the OTHERWISE clause, if present, is performed. If the OTHERWISE clause should be executed and it is not present, a Debug Tool message is issued.



reference
: A valid Debug Tool PL/I scalar reference. An aggregate (array or structure) cannot be used as a reference.

WHEN
: Specifies that an expression or a group of expressions be evaluated and either compared with the reference immediately following the SELECT keyword, or evaluated to true or false (if *reference* is omitted).

expression
: A valid Debug Tool PL/I expression.

command
: A valid Debug Tool command.

OTHERWISE
: Specifies the command to be executed when every test of the preceding WHEN statements fails.

Example:

When sum is equal to the value of c+ev, display a message. When sum is equal to either fv or 0, display a message. If sum is not equal to the value of either c+ev, fv, or 0, a Debug Tool error message is issued.

```
SELECT (sum);
  WHEN (c + ev) LIST ('Match on when group number 1');
  WHEN (fv, 0) LIST ('Match on when group number 2');
END;
```

SET Command

The SET command sets various switches that affect the operation of Debug Tool. Except where otherwise specified, settings remain in effect for the entire debug session.

The various forms of the SET command are summarized in Table 14.

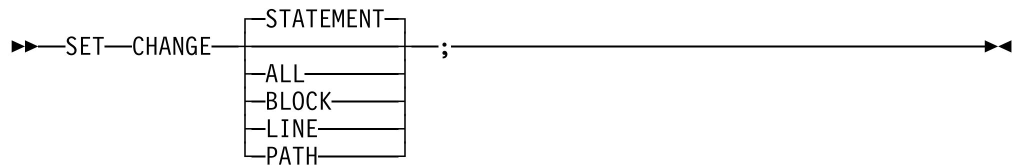
Table 14 (Page 1 of 2). Summary of SET Commands

SET CHANGE	controls the frequency of checking the AT CHANGE breakpoints.
SET COLOR	provides control of the color, highlighting, and intensity attributes.
SET COUNTRY	changes the current national country setting.
SET DBCS	controls whether DBCS shift-in and shift-out codes are recognized.
SET DEFAULT LISTINGS	defines a default sublibrary to be searched for program source listings or source files.
SET DEFAULT SCROLL	sets the default scroll amount.
SET DEFAULT WINDOW	specifies what window is defaulted.
SET ECHO	controls whether GO and STEP commands are recorded in the Log window.
SET EQUATE	equates a symbol to a string of characters.
SET EXECUTE	controls whether commands are performed or just syntax checked.
SET FREQUENCY	controls whether statement executions are counted.
SET HISTORY	specifies whether entries to Debug Tool are recorded in the history table.
SET INTERCEPT (C and COBOL)	intercepts input to and output from specified files, displaying prompts and output in the log.
SET KEYS	controls whether PF key definitions are displayed.
SET LOG	controls the logging of output and assignment to the log file.
SET LOG NUMBERS	controls whether line numbers are shown in the Log window.
SET MONITOR NUMBERS	controls whether line numbers are shown in the Monitor window.
SET MSGID	controls whether message identifiers are shown.

Table 14 (Page 2 of 2). Summary of SET Commands	
SET NATIONAL LANGUAGE	switches your application to a different run-time national language.
SET PACE	specifies the maximum pace of animated execution.
SET PFKEY	associates a Debug Tool command with a PF key.
SET PROGRAMMING LANGUAGE	sets the current programming language.
SET PROMPT	controls the display of the current program location.
SET QUALIFY	simplifies the identification of references and statement numbers by resetting the point of view.
SET REFRESH	controls screen refreshing when the SCREEN setting is ON.
SET REWRITE	forces a periodic screen rewrite.
SET SCREEN	controls how information is displayed on the screen.
SET SCROLL DISPLAY	controls whether the scroll field is displayed.
SET SOURCE	associates a source listing or source file with one or more compile units.
SET SUFFIX	controls the display of the Source window suffix area.
SET TEST	overrides the initial run-time TEST options specified at invocation.
SET WARNING	controls display of the Debug Tool warning messages and whether exceptions are reflected to the application program.

SET CHANGE

Controls the frequency of checking the AT CHANGE breakpoints. The initial setting is STATEMENT/LINE.



STATEMENT
Specifies that the AT CHANGE breakpoints are checked at all statements. STATEMENT is equivalent to LINE.

ALL
Specifies that the AT CHANGE breakpoints are checked at all statements, block entry and exits, and path points.

BLOCK
Specifies that the AT CHANGE breakpoints are checked at all block entry and exits.

LINE
Is equivalent to STATEMENT.

PATH
Specifies that the AT CHANGE breakpoints are checked at all path points.

Examples:

- Specify that AT CHANGE breakpoints are checked at all statements.

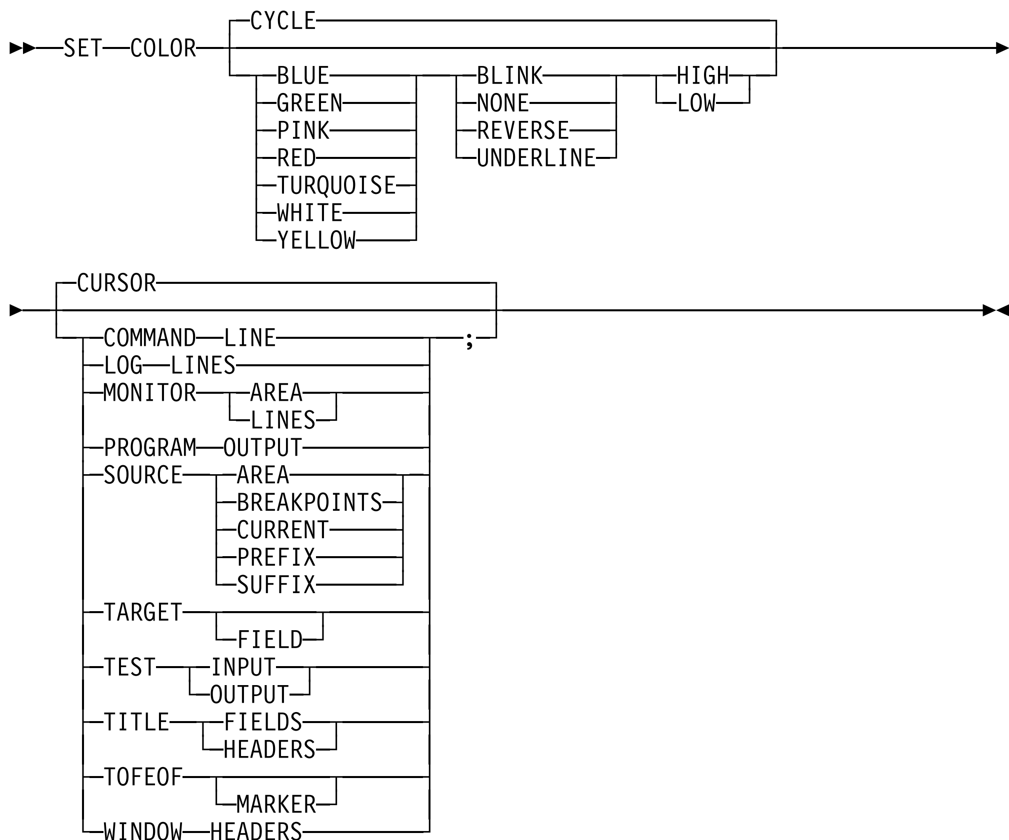
```
SET CHANGE;
```

- Specify that AT CHANGE breakpoints are checked at all path points.

```
SET CHANGE PATH;
```

SET COLOR (Full-Screen Mode)

Provides control of the color, highlighting, and intensity attributes when the SCREEN setting is ON. The color, highlighting, and intensity keywords can be specified in any order.



CYCLE
Causes the color to change to the next one in the sequence of colors. The sequence follows the order shown in the syntax diagram.

BLINK
Causes the characters to blink (if supported by the terminal).

NONE
Causes the characters to appear in normal type.

REVERSE
Transforms the characters to reverse video (if supported by the terminal).

UNDERLINE
Causes the characters to be underlined (if supported by the terminal).

HIGH
Causes screen colors to be high intensity (if supported by the terminal).

LOW
Causes screen colors to be low intensity (if supported by the terminal).

CURSOR
Specifies that cursor pointing is used to select the field. Optionally, you can type in the field name (for example, COMMAND LINE) as shown in the syntax diagram.

COMMAND LINE
Selects the command input line (preceded by ===>).

LOG LINES
Selects the line number portion of the Log window.

MONITOR AREA
Selects the primary area of the Monitor window.

MONITOR LINES
Selects the line number portion of the Monitor window.

PROGRAM OUTPUT
Selects the application program output displayed in the Log window.

SOURCE AREA
Selects the primary area of the Source window.

SOURCE BREAKPOINTS
Selects the source prefix fields next to statements where breakpoints are set.

SOURCE CURRENT
Selects the line containing the source statement that is about to be performed.

SOURCE PREFIX
Selects the statement identifier column at the left of the source window.

SOURCE SUFFIX
Selects the frequency column at the right of the Source window.

TARGET FIELD
Selects the target of a FIND command in full-screen mode, if found.

TEST INPUT
Selects the Debug Tool input displayed in the Log window.

TEST OUTPUT
Selects the Debug Tool output displayed in the Log window.

TITLE FIELDS
Selects the information fields in the top line of the screen, such as current programming language setting or the current location within the program.

TITLE HEADERS
Selects the descriptive headers in the top line of the screen, such as location.

TOFEOF MARKER
Selects the top-of-file and end-of-file lines in the session panel windows.

WINDOW HEADERS
Selects the header lines for the windows in the main session panel.

Examples:

- Set the Source window display area to yellow reverse video.

```
SET COLOR YELLOW REVERSE SOURCE AREA;
```

- Set the Monitor window display area to high intensity green.

```
SET COLOR HIGH GREEN MONITOR AREA;
```

SET COUNTRY

Changes the current national country setting for the application program. It is available only where supported by LE/VSE. The IBM-supplied initial country code is US.

```
►►──SET──COUNTRY──country_code──;──────────────────────────────────────►◄
```

country_code
A valid two-letter set that identifies the country code used. The country code can have one of the following values:

United States: US
Japanese: JP

Country codes cannot be truncated.

Usage Notes:

- This setting affects both your application and Debug Tool.
- At the beginning of an enclave, the settings are those provided by LE/VSE or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.

Example:

Change the current country code to correspond to Japan.

```
SET COUNTRY JP;
```

SET DBCS

Controls whether shift-in and shift-out codes are interpreted on input and supplied on DBCS output. SET DBCS is valid for all programming languages; however, when used with C, it is always considered ON.

```
                  ┌─ON──┐
►►──SET──DBCS─────┼─────┼──;────────────────────────────────────────►◄
                  └─OFF─┘
```

ON
Interprets shift-in and shift-out codes.

OFF
Ignores shift-in and shift-out codes.

Example:

Specify that shift-in and shift-out codes are interpreted.

```
SET DBCS ON;
```

CICS Only

Using SET DBCS ON under CICS may cause ATNI transaction abends. This is because the CICS translator can generate undisplayable characters in the parameter string for the EXEC Interface program. For example, a shift out character is generated in the parameter string for the EXEC CICS RETURN command.

With DBCS set to OFF, Debug Tool will convert any undisplayable character in the source listing to the period (.) character. With DBCS set to ON no such conversion is performed.

End of CICS Only

SET DEFAULT LISTINGS

Defines a default sublibrary to be searched for program source listings or source files. The LISTINGS keyword cannot be abbreviated.

listings_file
Specifies a valid sublibrary name to be searched for program source listings or source files.

Usage Notes:

- The SET SOURCE ON command has a higher precedence than the SET DEFAULT LISTINGS command.

Example:

Indicate that the default listings file is allocated to sublibrary `LISTINGS.LIBRARY`.

```
SET DEFAULT LISTINGS LISTINGS.LIBRARY ;
```

SET DEFAULT SCROLL (Full-Screen Mode)

Sets the default scroll amount that is used when a SCROLL command is issued without the amount specified. The initial setting is PAGE.

CSR
Scrolls in the specified direction until the character where the cursor is positioned reaches the edge of the window.

DATA
Scrolls by one line less than the window size or by one character less than the window size (if moving left or right).

HALF
Scrolls by half the window size.

integer
Scrolls the specified number of lines (up or down) or the specified number of characters (left or right). Maximum value is 9999.

MAX
Scrolls in the specified direction until the limit of the data is reached.

PAGE
Scrolls by the window size.

Example:

Set the default amount to half the size of the window.

```
SET DEFAULT SCROLL HALF;
```

SET DEFAULT WINDOW (Full-Screen Mode)

Specifies what window is selected when a window referencing command (for example, FIND, SCROLL, or WINDOW) is issued without explicit window identification and the cursor is outside the window areas. The initial setting is SOURCE.

LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the source listing window.

Example:

Set the default to the Monitor window for use with scrolling commands.

```
SET DEFAULT WINDOW MONITOR;
```

SET ECHO

Controls whether GO and STEP commands are recorded in the Log window when they are not subcommands. The presence of long sequences of GO and STEP commands clutters the Log window and provides little additional information. SET ECHO makes it possible to suppress the display of these commands. The contents of the log file are unaffected. The initial setting is ON.

```
                        ┌─*───────┐
►►──SET──ECHO──┬─ON──┬──┼─────────┼──;──────────────────────────────►◄
               └─OFF─┘  └─keyword─┘
```

ON
Shows given commands in the Log window.

OFF
Suppresses given commands in the Log window.

keyword
Can be GO (with no operand) or STEP.

*****
Specifies that the command is applied to the GO and STEP commands. This is the default.

Examples:

- Specify that the display of GO and STEP commands is suppressed.

  ```
  SET ECHO OFF;
  ```

- Specify that GO and STEP commands are displayed.

  ```
  SET ECHO ON *;
  ```

SET EQUATE

Equates a symbol to a string of characters. The equated symbol can be used anywhere a keyword, identifier, or punctuation is used in a Debug Tool command. When an equated symbol is found in a Debug Tool command (other than the *identifier* operand in SET EQUATE and CLEAR EQUATE), the equated symbol is replaced by the specified string before parsing continues.

```
►►──SET──EQUATE──identifier──=──string──;──────────────────────────►◄
```

identifier
An identifier that is valid in the current programming language. The maximum length of the identifier is:

- For C, 32 SBCS characters
- For COBOL, 30 SBCS characters
- For PL/I, 31 SBCS characters

The identifier can contain DBCS characters.

string
A string constant in the current programming language. The maximum length of the replacement string is 255 SBCS characters.

Usage Notes:

- Operands of the following commands are for environments other than the standard Debug Tool environment and are not scanned for EQUATEd symbol substitution:

  COMMENT
  INPUT
  SET DEFAULT LISTINGS
  SET INTERCEPT ON/OFF FILE
  SET LOG ON FILE
  SET SOURCE (cu_spec)
  USE

- To remove an EQUATE definition, use the CLEAR EQUATE command.

- To remain accessible when the current programming language setting is changed, symbols that are equated when the current programming

language setting is C must be entered in uppercase and must be valid in the other programming languages.

- If an EQUATE identifier coincides with an existing keyword or keyword abbreviation, EQUATE takes precedence. If the EQUATE identifier is already defined, the new definition replaces the old.
- The equate string is not scanned for, or substituted with, symbols previously set with a SET EQUATE command.

Examples:

- Specify that the symbol `INFO` is equated to "ABC, DEF (H+1)". The current programming language setting is either C or COBOL.

```
SET EQUATE INFO = "ABC, DEF (H+1)";
```

- Specify that the symbol `tstlen` is equated to the equivalent of a `#define` for structure pointing. The current programming language setting is C. Note that this lowercase symbol will not necessarily be accessible if the current programming language changes.

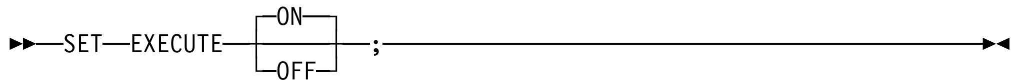
```
SET EQUATE tstlen = "struct1->member.b->c.len";
```

- Specify that the symbol `VARVALUE` is equated to the command LIST `x`.

```
SET EQUATE VARVALUE = "LIST x";
```

SET EXECUTE

Controls whether commands from all input sources are performed or just syntax checked (primarily for checking USE files). The initial setting is ON.



ON
Specifies that commands are accepted and performed.

OFF
Specifies that commands are accepted and parsed; however, only the following commands are performed: END, GO, SET EXECUTE ON, QUIT, and USE.

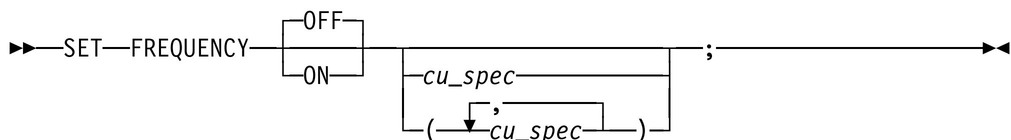
Example:

Specify that all commands are accepted and performed.

```
SET EXECUTE ON;
```

SET FREQUENCY

Controls whether statement executions are counted. The initial setting is OFF.



ON
Specifies that statement executions are counted.

OFF
Specifies that statement executions are not counted.

cu_spec
A valid compile unit specification; see “CU_Spec” on page 201. If omitted, all compile units with statement information are processed.

Note: See also “LIST FREQUENCY” on page 277.

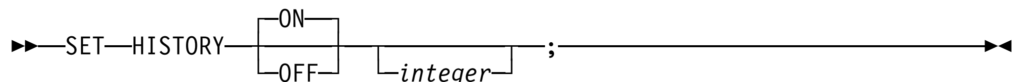
Example:

Specify that statement executions are counted in compile units `main` and `subr1`.

```
SET FREQUENCY ON (main, subr1);
```

SET HISTORY

Specifies whether entries to Debug Tool are recorded in the history table and optionally adjusts the size of the table. The history table contains information about the most recently processed breakpoints and conditions. The initial setting is ON; the initial size is 100.



ON
Maintains the history of invocations.

OFF
Suppresses the history of invocations.

integer
The number of entries kept in the history table.

Note: See also “LIST LAST” on page 278.

Examples:

- Adjust the history table size to 50 lines.

  ```
  SET HISTORY 50;
  ```

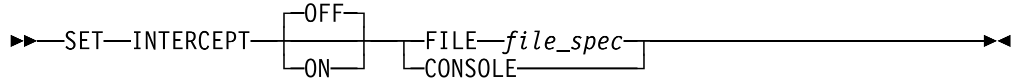
- Turn off history recording.

  ```
  SET HISTORY OFF;
  ```

SET INTERCEPT (C and COBOL)

Intercepts input to and output from specified files. Output and prompts for input are displayed in the log.

Only sequential I/O can be intercepted. I/O intercepts remain in effect for the entire debug session, unless you terminate them by selecting SET INTERCEPT OFF. The initial setting is OFF.



ON

Turns on I/O interception for the specified file. Output appears in the log, preceded by the file specifier for identification. Input causes a prompt entry in the log, with the file specifier identified. You can then enter input for the specified file on the command line by using the INPUT command. See "INPUT Command (C and COBOL)" on page 271.

OFF

Turns off I/O interception for the specified file.

FILE *file_spec* **(C)**

A valid file specification that is interpreted by each supported language. The FILE keyword cannot be abbreviated.

In C, this can be any valid `fopen()` file specifier including stdin, stdout, or stderr.

Note: If the interception is for a sublibrary member, the file specification (DD:) must include the sublibrary name, even if the fopen() is unqualified. that is, a specification of the form `DD:library.sublibry(member.name)` is required.

CONSOLE (COBOL)

Turns on I/O interception for the console.

This consists of:

- Job log output from DISPLAY UPON CONSOLE
- Screen output (and confirming input) from STOP 'literal'
- Terminal input for ACCEPT FROM CONSOLE or ACCEPT FROM SYSIN.

Usage Notes:

- COBOL supports only the CONSOLE command.
- For C, intercepted streams or files cannot be part of any C I/O redirection during the execution of a nested enclave.
- For PL/I, SET INTERCEPT is not supported.
- For CICS, SET INTERCEPT is not supported.

Examples:

- Turn on the I/O interception for the console. The current programming language setting is COBOL.

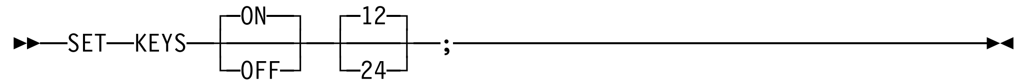
  ```
  SET INTERCEPT CONSOLE;
  ```

- Turn on the I/O interception for the `fopen()` file specifier `dd:mydd`. The current programming language setting is C.

  ```
  SET INTERCEPT ON FILE dd:mydd;
  ```

SET KEYS (Full-Screen Mode)

Controls whether PF key definitions are displayed when the SCREEN setting is ON. The initial setting is ON.



ON
Displays PF key definitions.

OFF
Suppresses the display of the PF key definitions.

12 Shows PF1-PF12 on the screen bottom.

24 Shows PF13-PF24 on the screen bottom.

See also “SET PFKEY” on page 316.

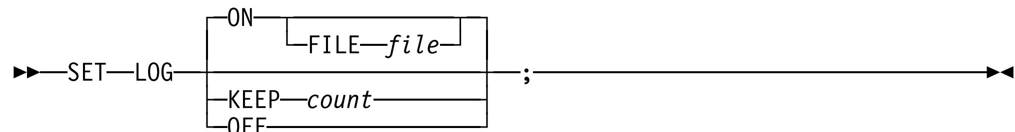
Example:

Specify that the display of the PF key definitions is suppressed.

```
SET KEYS OFF;
```

SET LOG

Controls whether each performed command and the resulting output is written to the log file and defines (or redefines) the file that is used. The initial setting is ON FILE SYSLST.



ON
Specifies that commands and output are written to the log file.

FILE *file*
Identifies the log file used. The FILE keyword cannot be abbreviated.

Details on the log file can be found in “The Log File” on page 30.

KEEP *count*
Specifies the number of lines of log output retained for display. The initial setting is 1000; *count* cannot equal zero (0).

OFF
Specifies that commands and output are not written to a log file.

Usage Notes:

- The log output lines retained for display are always the last (that is, the most recent) lines.
- Setting LOG OFF does not suppress the log display.
- If the same file name already exists, the output log is appended to the existing file.

Examples:

- Specify that commands and output are written to the sublibrary member `mainprog.log`.

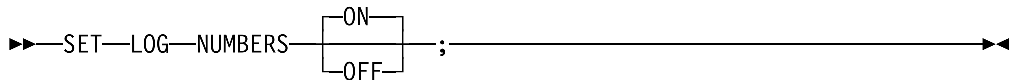
  ```
  SET LOG ON FILE (mainprog.log);
  ```

- Indicate that 500 lines of log output are retained for display.

  ```
  SET LOG KEEP 500;
  ```

SET LOG NUMBERS (Full-Screen Mode)

Controls whether line numbers are shown in the Log window. The initial setting is ON.



ON
Shows line numbers in the Log window.

OFF
Suppresses line numbers in the Log window.

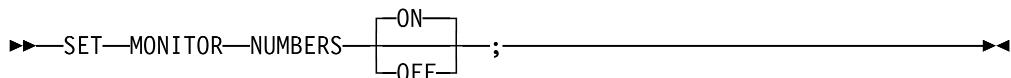
Example:

Specify that log line numbers are not shown.

```
SET LOG NUMBERS OFF;
```

SET MONITOR NUMBERS (Full-Screen Mode)

Controls whether line numbers are shown in the Monitor window. The initial setting is ON.



ON
Shows line numbers in the Monitor window.

OFF
Suppresses line numbers in the Monitor window.

Example:

Specify that monitor line numbers are not shown.

```
SET MONITOR NUMBERS OFF;
```

SET MSGID

Controls whether the Debug Tool messages are displayed with the message prefix identifiers. The initial setting is OFF.

```
             ┌─OFF─┐
►►──SET──MSGID──┼─────┼──;─────────────────────────────────────►◄
             └─ON──┘
```

ON
Displays message identifiers. The first 7 characters of the message contain the EQAnnnn message prefix identifier, then a blank, then the original message text, such as: 'EQA2222 Program does not exist.'

OFF
Displays only the message text.

Example:

Specify that message identifiers are suppressed.

```
SET MSGID OFF;
```

SET NATIONAL LANGUAGE

Switches your application to a different run-time national language that determines what translation is used when a message is displayed. The switch is effective for the entire run-time environment; it is not restricted to Debug Tool activity only. The initial setting is supplied by LE/VSE, according to the setting in the current enclave.

```
        ┌─NATIONAL─┐
►►──SET──┴──────────┴──LANGUAGE──language_code──;──────────────►◄
```

language_code
A valid three-letter set that identifies the language used. The language code can have one of the following values:

United States English: ENU
United States English (Uppercase): UEN
Japanese: JPN

Usage Notes:

- This setting affects both your application and Debug Tool.
- At the beginning of an enclave, the settings are those provided by LE/VSE or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.

Examples:

- Set the current national language to Japanese.

  `SET NATIONAL LANGUAGE JPN;`

- Set the current national language to United States English.

  `SET LANGUAGE ENU;`

SET PACE

Specifies the maximum pace of animated execution, in steps per second. The initial setting is two steps per second. This setting is not supported in batch mode and it has no effect under CICS.

```
►►──SET──PACE──number──;──────────────────────────────────────────►◄
```

number
A decimal number between 0 and 9999; it must be a multiple of 0.5.

Usage Notes:

- Associated with the SET PACE command is the STEP command. Animated execution is achieved by defining a PACE and then issuing a STEP *n* command where *n* is the number of steps to be seen in animated mode. `STEP *` can be used to see all steps to the next breakpoint in animated mode.
- When PACE is set to 0, no animation occurs.

Example:

Set the animated execution pace to 1.5 steps per second.

`SET PACE 1.5;`

SET PFKEY

Associates a Debug Tool command with a Program Function key (PF key). This setting is not supported in batch mode.

```
►►──SET──PFn──┬────────┬──=──command──;───────────────────────────►◄
              └─string─┘
```

PFn
A valid program function key specification (PF1 - PF24).

string
The label shown in the PF key display (if the KEYS setting is ON). It is entered as a string constant and is truncated if longer than eight characters. If the string is omitted, the first eight characters of the command are displayed.

command
A valid Debug Tool command or partial command.

Usage Notes:

- In Debug Tool, if there is any text on the command line at the time the PF key is pressed, that text is appended to the PF key string, with an intervening blank, for execution.
- In Debug Tool, the following initial PF key settings exist:

PF1 '?'	= ?
PF2 'STEP'	= STEP
PF3 'QUIT'	= QUIT
PF4 'LIST'	= LIST
PF5 'FIND'	= IMMEDIATE FIND
PF6 'AT/CLEAR'	= AT TOGGLE
PF7 'UP'	= IMMEDIATE UP
PF8 'DOWN'	= IMMEDIATE DOWN
PF9 'GO'	= GO
PF10 'ZOOM'	= IMMEDIATE ZOOM
PF11 'ZOOM LOG'	= IMMEDIATE ZOOM LOG
PF12 'RETRIEVE'	= IMMEDIATE RETRIEVE

PF keys 13-24 are equivalent to PF keys 1-12, respectively.

Example:

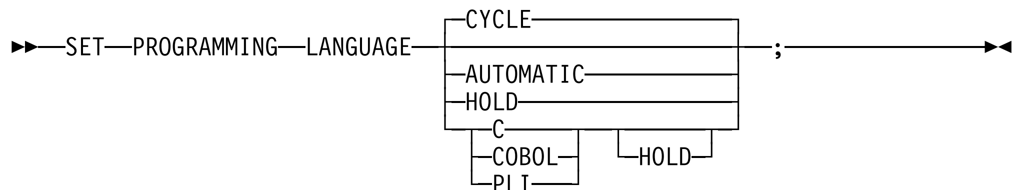
Define PF key 5 to scroll the cursor-selected screen forward. The current programming language setting is COBOL.

```
SET PF5 'Down' = IMMEDIATE SCROLL DOWN;
```

SET PROGRAMMING LANGUAGE

Sets the current programming language. You can only set the current programming language to the selection of languages of the programs currently loaded. For example, if the current phase contains both C and COBOL compile units, but not PL/I, you can set the language only to C or COBOL. However, if you later STEP or GO into another phase that contains C, COBOL, and PL/I compile units, you can set the language to any of the three.

The programming language setting affects the parsing of incoming Debug Tool commands. The execution of a command is always consistent with the current programming language setting that was in effect when the command was parsed. The programming language setting at execution time is ignored.



CYCLE
Specifies that the programming language is set to the next language in the alphabetic sequence of supported languages.

AUTOMATIC
Cancels a HOLD by specifying that the programming language is set according to the current qualification and thereafter changed automatically each time the qualification changes or STEP or GO is issued.

HOLD
Specifies that the given language (or the current language, if no language is specified) remains in effect regardless of qualification changes. The language remains in effect until SET PROGRAMMING LANGUAGE changes the language or releases the hold.

C Sets the current programming language to C.

COBOL
Sets the current programming language to COBOL.

PLI
Sets the current programming language to PL/I.

Usage Notes:

- If CYCLE or one of the explicit programming language names is specified, the current programming language setting is changed regardless of the currently suspended program or the current qualification.
- The current programming language setting affects how commands are parsed, not how they are performed. Commands are always performed according to the programming language setting where they were parsed. For example, it is not possible for a Debug Tool procedure to contain a mixture of C and COBOL commands; there is no way for the programming language setting to be changed while the procedure is being parsed. Also, it is not possible for a command parsed with one programming language setting to reference variables, types, or labels in another programming language.
- If SET PROGRAMMING LANGUAGE AUTOMATIC is in effect, changing the qualification automatically sets the current programming language to the specified block or compile unit.
- SET PROGRAMMING LANGUAGE can be used to set the programming language to any supported language in the current or parent enclaves.

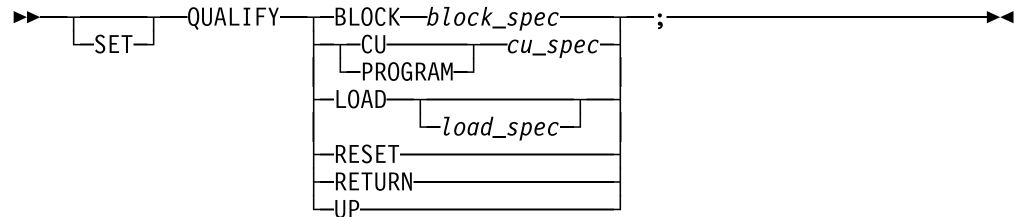
Example:

Specify that C is the current programming language.

```
SET PROGRAMMING LANGUAGE C;
```

SET QUALIFY

Simplifies the identification of references and statement numbers by resetting the point of view to a new block, compile unit, or phase. In full-screen mode this affects the contents of the Source window. If you are currently viewing one compile unit in your Source window and you want to view another, issue the SET QUALIFY command to change the qualification. The SET keyword is optional.



BLOCK
Sets the current point of view to the specified block.

block_spec
A valid block specification; see "Block_Spec" on page 200.

CU
Sets the current point of view to the specified compile unit. CU is equivalent to PROGRAM.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201.

PROGRAM
Is equivalent to CU.

LOAD
Sets the current point of view to the specified phase.

load_spec
A valid phase specification; see "Load_Spec" on page 203. If omitted, the initial (primary) phase qualification is used.

RESET
Resets qualification to the block of the suspended program and (if the SCREEN setting is ON) scrolls the Source window to display the current statement line.

RETURN
Switches qualification to the next higher calling program.

UP
Switches qualification up one lexical level to the statically containing block.

Usage Notes:

- If SET PROGRAMMING LANGUAGE AUTOMATIC is in effect (that is, HOLD is not in effect), changing the qualification automatically sets the current programming language to the specified block or compile unit.
- If you are debugging a program that has multiple enclaves, SET QUALIFY can be issued only for phases, compile units, and blocks which are known in the current enclave.
- The SET QUALIFY command does not imply a change in flow of control when the program is resumed with the GO command.
- The SET QUALIFY command cannot modify the point of view to a Debug Tool or library block.
- SET QUALIFY LOAD will not change the results of the QUERY QUALIFY command.

Examples:

- Indicate to Debug Tool that the phase `statphs` should be used when no phase is specified.

```
SET QUALIFY LOAD statphs;
```

- Set the qualification back to the point of the suspended program.

```
SET QUALIFY RESET;
```

- Set the block qualification to `blockx`. As a result, the phase qualification and compile unit qualification will be updated to the phase and compile unit which contain the block `blockx`.

```
SET QUALIFY BLOCK blockx;
```

SET REFRESH (Full-Screen Mode)

Controls screen refreshing. This command is only valid when in full screen mode, that is the SET SCREEN setting is ON. The initial setting for REFRESH is OFF.

```
                   ┌─OFF─┐
►►──SET──REFRESH───┼─────┼──;────────────────────────────────────►◄
                   └─ON──┘
```

ON
Clears the screen before each rewrite.

OFF
Rewrites without clear.

Note: SET REFRESH ON is needed for applications that also make use of the screen.

Example:

Specify that rewrites only occur on those portions of the screen that have changed. The screen is not cleared before being rewritten.

```
SET REFRESH OFF;
```

SET REWRITE

Forces a periodic screen rewrite during long sequences of output. This setting is not supported in batch mode.

```
►►──SET──REWRITE──┬─────────┬─number──;──────────────────────────►◄
                  └─EVERY───┘
```

number
Specifies how many lines of intercepted output are written by the application program before Debug Tool refreshes the screen. The initial setting is 50.

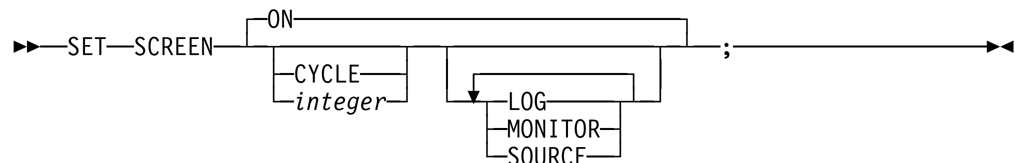
Example:

Force screen rewrite after each 100 lines of screen output.

```
SET REWRITE EVERY 100;
```

SET SCREEN (Full-Screen Mode)

Controls how information is displayed on the screen. The initial setting for a supported full-screen terminal is ON.



CYCLE
Switches to the next window configuration in sequence.

integer
An integer in the range 1 to 6, selecting the window configuration. The initial setting is 1.

LOG or MONITOR or SOURCE
Specifies the sequence of window assignments within the selected configuration (left to right, top to bottom). There must be no more than three objects specified and they must all be different.

See “Changing Session Panel Window Layout” on page 95 for more information.

ON
Activates the Debug Tool full-screen services.

Usage Note: If neither CYCLE nor *integer* is specified, there is no change in the choice of configuration. If no windows are specified, there is no change in the assignment of windows to the configuration.

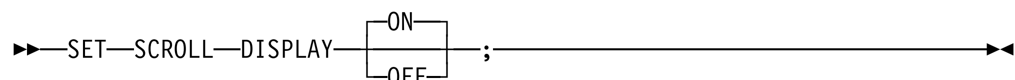
Example:

Indicate that the Log window is positioned above the Source window on the left hand side of the screen and the Monitor window is to occupy the upper right side portion of the screen. For more information, see “Customizing Your Session” on page 94.

```
SET SCREEN 2 LOG MONITOR;
```

SET SCROLL DISPLAY (Full-Screen Mode)

Controls whether the scroll field is displayed when operating in full-screen mode. The initial setting is ON.



ON
Displays scroll field.

OFF
Suppresses scroll field.

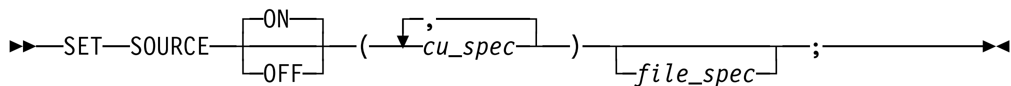
Example:

Specify that the scroll field is suppressed.

```
SET SCROLL DISPLAY OFF;
```

SET SOURCE

Associates a source file (for C) or source listing (for COBOL or PL/I) with one or more compile units.



ON
Displays the compile unit source file when active.

OFF
Specifies that the file is not displayed.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201. Multiple compile units can be associated with the same source listing or source file.

file_spec
Identifies the compile unit source file used. It is used in place of the default sublibrary member or file-id for the compile unit. It can be a sublibrary member, a filename, or a file-id.

A valid name is required unless it is already known to Debug Tool (via a previous SET SOURCE) or the default file name is valid.

Usage Notes:

- When SET SOURCE is issued against a compile unit that is the current compile unit, it checks for the existence of the file. However, if the compile unit is *not* the current compile unit, this check is not done. The file associated with the source may not exist and the error (for nonexistent file) does not appear until a function which requires this file is attempted.
- The SET SOURCE ON command has a higher precedence than the SET DEFAULT LISTINGS command.
- For COBOL, if the cu_spec includes any names that are case sensitive include the name in single or double quotes.
- For PL/I, you may need to use the SET SOURCE command to specify the location of your listing file if the CU or program name is not the same as the listing file name. For example, for program name `AVER` Debug Tool looks for the sublibrary member `AVER.LIST`. If the Debug Tool window comes up empty, use the following command:

```
SET SOURCE ON (PGMNAME) listings.library(cu_name.LIST) ;
```

or

```
SET SOURCE ON (PGMNAME) (cu_name.LIST) ;
```

This specifies the actual location of the listing file, in this example a sublibrary member with the program name differing from the CU name.

- For C, if your program has been compiled from SYSIPT (that is, Debug Tool shows the CU name as DD:SYSIPT) you should use the following command to point to the correct location for the source file:

```
SET SOURCE ON (%CU) (cu_name.c) ;
```

Examples:

- Indicate that the source file associated with compile unit OEFUN is found in a member OEFUN.C in a sublibrary in the SOURCE search chain.

```
SET SOURCE ON (oefun) (oefun.c) ;
```

- Indicate that the source file associated with a compile unit compiled from SYSIPT is found as member `oefun2.c` in a sublibrary in the SOURCE search chain.

```
SET SOURCE ON (%CU) (oefun2.c) ;
```

SET SUFFIX (Full-Screen Mode)

Controls the display of frequency counts at the right edge of the Source window when in full-screen mode. The initial setting is ON.

```
               ┌─ON──┐
►►──SET──SUFFIX──┼─────┼──;──────────────────────────────────────►◄
               └─OFF─┘
```

ON
Displays the suffix column.

OFF
Suppresses the suffix column.

Example:

Specify that the suffix column is displayed.

```
SET SUFFIX ON;
```

SET TEST

Overrides the initial run-time TEST options specified at invocation. The initial setting is ALL.

```
              ┌─test_level───────────┐
►►──SET──TEST──┴─(──test_level──)─────┴──;──────────────────────►◄
```

test_level
Specifies what exception conditions cause Debug Tool to gain control, even though no breakpoint exists. The parentheses are optional.

Test_level can include the following:

ALL
Specifies that the occurrence of an attention interrupt, termination of your program (either normally or through an ABEND), or any program or LE/VSE condition of Severity 1 and above causes Debug Tool to gain control, regardless of whether a breakpoint is defined for that type of condition. If a condition occurs and a breakpoint exists for the condition, the commands specified in the breakpoint are executed. If a condition occurs and a breakpoint does not exist for that condition, or if an attention interrupt occurs, Debug Tool will:

- In interactive mode, read commands from the commands file (if it exists) or prompt you for commands, or
- In batch mode, read commands from the commands file.

For more information about attention interrupts, see "Requesting an Attention Interrupt During Interactive Sessions" on page 133.

ERROR
Specifies that only the following conditions cause Debug Tool to gain control without a user-defined breakpoint.

- An attention interrupt
- Program termination
- A predefined LE/VSE condition of Severity 2 or above
- Any C condition other than SIGUSR1, SIGUSR2, SIGINT or SIGTERM.

Note: LE/VSE conditions are described in *LE/VSE Debugging Guide and Run-Time Messages*.

If a breakpoint exists for one of the above conditions, any commands specified in the breakpoint are executed. If no commands are specified, Debug Tool reads commands from a commands file or prompts you for commands in interactive mode.

NONE
Specifies that Debug Tool gains control only at an attention interrupt, or at a condition if a breakpoint is defined for that condition. If a breakpoint does exist for the condition, the commands specified in the breakpoint are executed.

Examples:

- Indicate that only an attention interrupt or exception causes Debug Tool to gain control when no breakpoint exists.

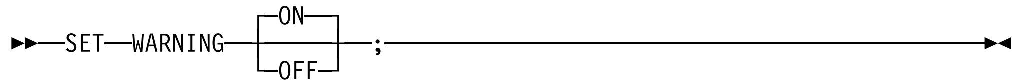
```
SET TEST ERROR;
```

- Indicate that no condition causes Debug Tool to gain control unless a breakpoint exists for that condition.

```
SET TEST NONE;
```

SET WARNING (C and PL/I)

Controls display of the Debug Tool warning messages and whether exceptions are reflected to the application program. The initial setting is ON.



ON
Displays the Debug Tool warning messages, and conditions such as a divide check result in a diagnostic message.

OFF
Suppresses the Debug Tool warning messages, and conditions raise an exception in the application program.

Exceptions that occur due to interaction with you are likely to be due to typing errors and are probably not intended to be passed to the application program. However, you might want to raise a real exception in the program, for example, to test some error recovery code. (TRIGGER is not always appropriate for this because it does not set up the exception information.)

Usage Notes:

- Debug Tool detects C conditions such as the following:
  - Division by zero
  - Array subscript out of bounds for defined arrays
  - 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.

  See “C Expressions” on page 145 for more information about which conditions will be reported when WARNING is ON.
- Debug Tool detects the following PL/I computational conditions:
  - Invalid decimal data
  - CHARACTER to BIT conversion errors
  - Division by zero
  - Invalid length in varying strings

  See “Using SET WARNING Command with Built-Ins” on page 187 for more information about which conditions will be reported when WARNING is ON.

Example:

Specify that conditions result in a diagnostic message.

```
SET WARNING ON;
```

SET Command (COBOL)

The SET command assigns a value to a COBOL reference. The SET keyword cannot be abbreviated.

```
►►──SET──reference──TO──┬─reference─┬──;──────────────────────────►◄
                        └─literal───┘
```

reference
A valid Debug Tool COBOL reference.

literal
A valid COBOL numeric literal constant.

Usage Notes:

- If Debug Tool was invoked because of a computational condition or an attention interrupt, using an assignment to set a variable might not give expected results. This is due to the uncertainty of variable values within statements as opposed to their values at statement boundaries.
- SET assigns a value only to a single receiver; unlike COBOL, multiple receiver variables are not supported.
- Only formats 1 and 5 of the COBOL SET command are supported.
- Index-names can only be program variables (since OCCURS is not supported for the Debug Tool session variables).
- COBOL ADDRESS OF identifier is supported only for identifiers that are LINKAGE SECTION variables. In addition, COBOL ADDRESS OF as a receiver must be level 1 or 77, and COBOL ADDRESS OF as a sender can be any level except 66 or 88.
- Only the sender/receiver combinations listed in “Allowable Moves for the Debug Tool SET Command” on page 352 are supported.

Examples:

- Set index name `table-index` to 5.

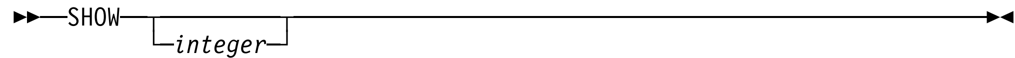
  ```
  SET table-index TO 5;
  ```

- Assign to variable `h-address` the address of variable `m-name`.

  ```
  SET h-address TO ADDRESS OF m-name;
  ```

SHOW Prefix Command (Full-Screen Mode)

The SHOW Prefix command specifies what relative statement (for C) or relative verb (for COBOL) within the line is to have its frequency count temporarily shown in the suffix area.



integer
Selects a relative statement (for C) or a relative verb (for COBOL) within the line. The default value is 1.

Usage Notes:

- If SET SUFFIX is currently OFF, SHOW Prefix forces it ON.
- The suffix display returns to normal on the next interaction.
- The SHOW Prefix command is not logged.

Example:

Display the frequency count of the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
SHOW 3
```

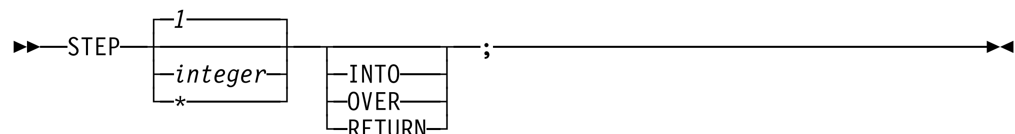
No space is needed as a delimiter between the keyword and the integer; hence, SHOW 3 is equivalent to SHOW3.

STEP Command

The STEP command causes Debug Tool to dynamically step through a program, executing one or more program statements. In full-screen mode, it provides animated execution.

STEP ends if one or more of the following conditions is reached:

- Attention interrupt
- A breakpoint is encountered
- Normal or unusual termination of the program



integer
Indicates the number of statements performed. The default value is 1. If *integer* is greater than 1, the statement is performed as if it were that many repetitions of STEP with the same keyword and a count of one. The speed of execution, or the *pace* of stepping, is set by either the SET PACE command, or with the **Pace of visual trace** field on the Profile panels.

*****
Specifies that the program should run until interrupted. STEP * is equivalent to GO.

INTO
Steps into any called procedures or functions. This means that stepping continues within called procedures or functions. This is the default except when the called procedure or function is a library or operating system routine.

OVER
Steps over any procedure call or function invocations. This operand provides full-speed execution (with no animation) while in called procedures and functions, resuming STEP mode on return. This is the default when the called procedure or function is a library or operating system routine.

RETURN
Steps to the return point the specified number of levels back, halting at the statement following the corresponding procedure call or function invocation. This operand provides full-speed execution (with no animation) for the remainder of the current procedure or function, and for any called procedures or functions, resuming STEP mode on return.

Usage Notes:

- If STEP is specified in a command list (for example, as the subject of an IF command or WHEN clause), all subsequent commands in the list are ignored.
- If STEP is specified within the body of a loop, it causes the execution of the loop to end.
- To suppress the logging of STEP commands, use the SET ECHO OFF command.
- If two operands are given, they can be specified in either order.
- The animation execution timing is set by the SET PACE command.
- The source panel provides a means of suppressing the display of selected listings or files. This gives some control of "debugging scope," since animated execution does not occur within a phase where the source listing or source file is not displayed.

Examples:

- Step through the next 25 statements and if an application subroutine or function is invoked, continue stepping into that subroutine or function.

```
STEP 25 INTO;
```

- Step through the next 25 statements, but if any application subroutines or functions are invoked, switch to full-speed execution without animation until the subroutine or function returns.

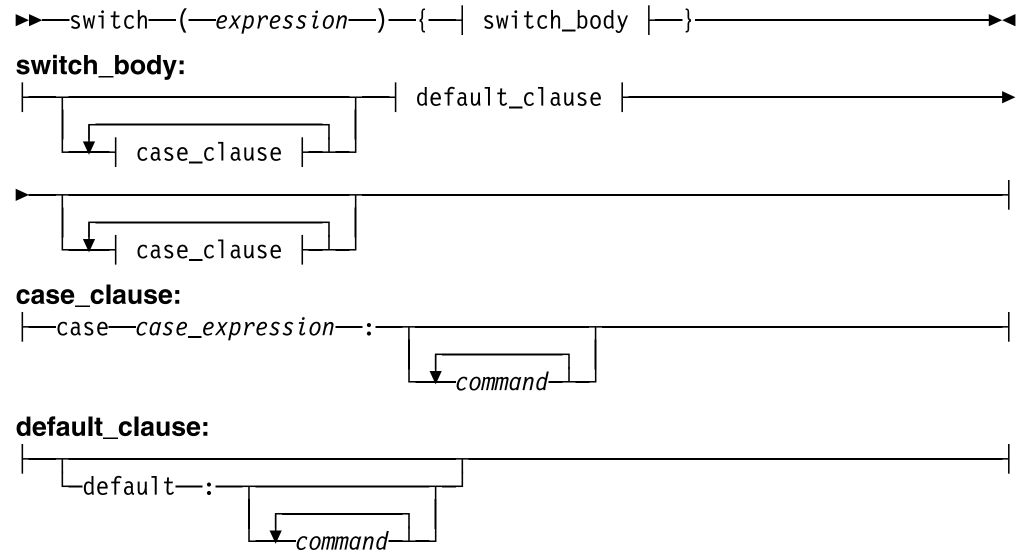
```
STEP 25 OVER;
```

- Return at full speed through three levels of calls.

```
STEP 3 RETURN;
```

switch Command (C)

The `switch` command enables you to transfer control to different commands within the `switch` body, depending on the value of the `switch` expression. The `switch`, `case`, and `default` keywords must be lowercase and cannot be abbreviated.



expression
A valid Debug Tool C expression.

case_expression
A valid character or optionally signed integer constant.

command
A valid Debug Tool command.

The value of the switch expression is compared with the value of the expression in each case clause. If a matching value is found, control is passed to the command in the case clause that contains the matching value. If a matching value is not found and a default clause appears anywhere in the switch body, control is passed to the command in the default clause. Otherwise, control is passed to the command following the switch body.

If control passes to a command in the switch body, control does not pass from the switch body until a break command is encountered or the last command in the switch body is performed.

Usage Notes:

- Declarations are not allowed within a switch command.
- The switch command does not end with a semicolon. A semicolon after the closing brace is treated as a Null command.
- Although this command is similar to the switch statement in C, it is subject to Debug Tool restrictions on expressions.
- Duplicate *case_expression* values are not supported.

Examples:

- The following `switch` command contains several `case` clauses and one `default` clause. Each clause contains a function call and a `break` command. The `break` commands prevent control from passing down through subsequent commands in the `switch` body.

  If `key` has the value `'/'`, the `switch` command calls the function `divide`. On return, control passes to the command following the `switch` body.

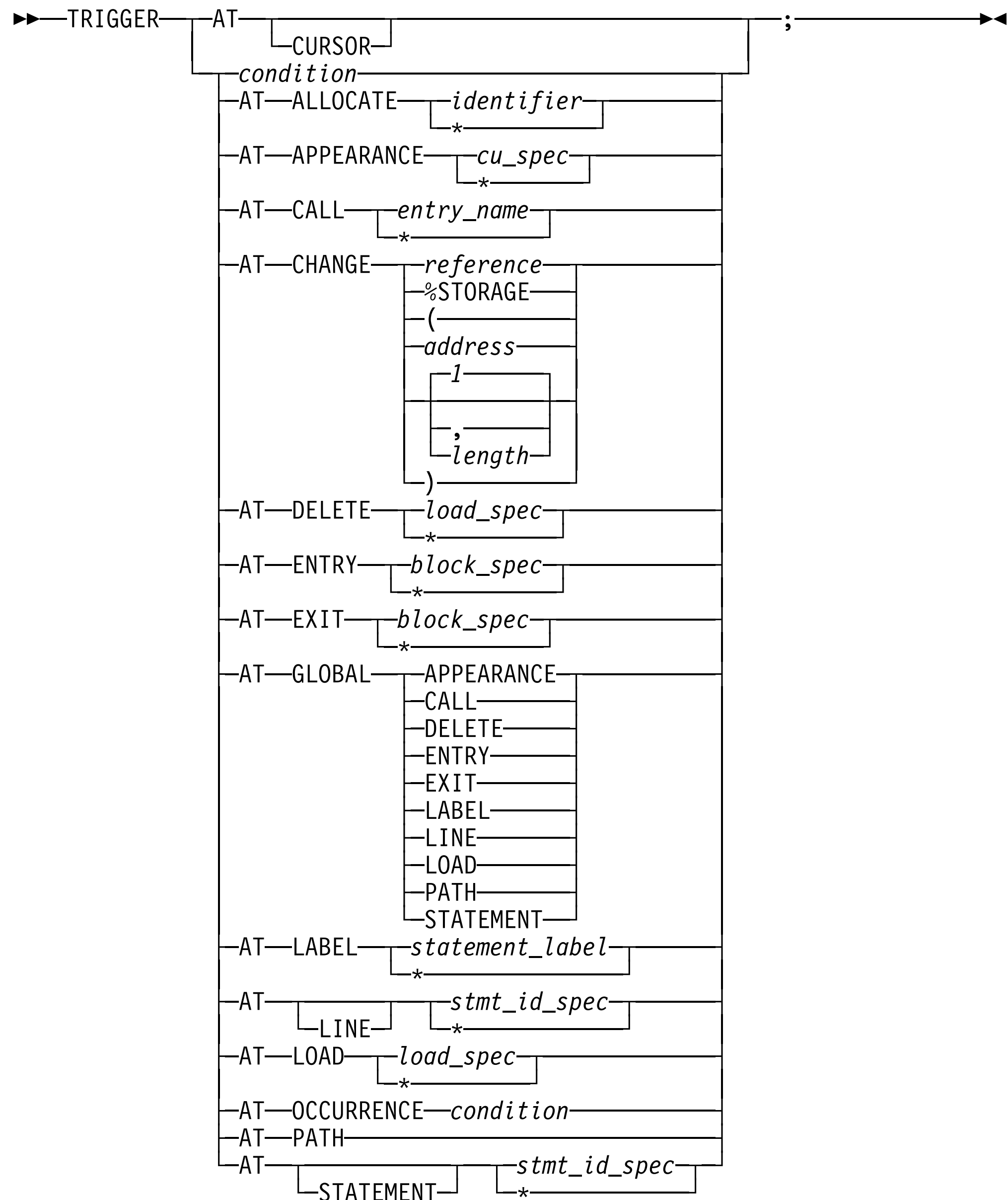
  ```
  char key;

  printf("Enter an arithmetic operator\n");
  scanf("%c",&key);

  switch (key)
  {
     case '+':
        add();
        LIST (key);
        break;
     case '-':
        subtract();
        LIST (key);
        break;
     case '*':
        multiply();
        LIST (key);
        break;
     case '/':
        divide();
        LIST (key);
        break;
     default:
        printf("Invalid key\n");
        break;
  }
  ```

- In the following example, `break` commands are not present. If the value of `c` is equal to 'A', all 3 counters are incremented. If the value of `c` is equal to 'a', `lettera` and `total` are increased. Only `total` is increased if `c` is not equal to 'A' or 'a'.

  ```
  char text[100];
  int  capa, i, lettera, total;

  for (i=0; i < sizeof(text); i++) {

      switch (text[i]) {
         case 'A':
           capa++;
         case 'a':
           lettera++;
         default:
           total++;
      }
  }
  ```

TRIGGER Command

The TRIGGER command raises the specified AT-condition in Debug Tool, or it raises the specified programming language condition in your program.



condition
: A valid condition or exception. This can be either an LE/VSE symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.

If no active condition handler exists for the specified condition, the default condition handler can cause the program to end prematurely.

Following are the C condition constants; they must be uppercase and not abbreviated. See also Appendix C, “Using C Reference Information with Debug Tool” on page 346 for a list of C conditions and their LE/VSE equivalents.

SIGABND SIGABRT SIGFPE SIGILL SIGINT SIGIOERR SIGSEGV SIGTERM SIGUSR1 SIGUSR2

There are no COBOL condition constants. Instead, an LE/VSE symbolic feedback code must be used, for example, CEE347. See *LE/VSE Programming Guide* for more details about language condition handling interactions.

PL/I condition constants may be used, for syntax and acceptable abbreviations see "ON Command (PL/I)" on page 287.

cu_spec
A valid compile unit specification; see "CU_Spec" on page 201.

entry_name
A valid external entry point name constant or zero (`0`); however, `0` can only be specified if the current programming language setting is C or PL/I.

reference
A valid Debug Tool reference in the current programming language; see "References" on page 203.

%STORAGE
A built-in function that provides an alternative way to select an AT CHANGE subject.

address
The starting address of storage to be watched for changes. This must be a hex constant: *0x* in C, *H* in COBOL (using either double (") or single (') quotes), or a *PX* constant in PL/I.

length
The number of bytes of storage being watched for changes. This must be a positive integer constant. The default value is 1.

load_spec
A valid phase specification; see "Load_Spec" on page 203.

block_spec
A valid block specification; see "Block_Spec" on page 200.

statement_label
A valid source label constant; see "Statement_Label" on page 205.

stmt_id_spec
A valid statement id specification; see "Statement_Id_Range and Stmt_Id_Spec" on page 204.

Usage Note:

- AT TERMINATION cannot be raised by TRIGGER.

Examples:

In the following examples, note the difference between triggering a breakpoint, which performs Debug Tool commands associated with the breakpoint, and triggering a condition, which actually raises the condition and causes a corresponding *system* action.

- Perform the commands in the AT OCCURRENCE CEE347 breakpoint (the CEE347 condition is not raised). The current programming language setting is COBOL.

```
AT OCCURRENCE CEE347 PERFORM
  SET ix TO 5;
END-PERFORM;

TRIGGER AT OCCURRENCE CEE347;   /* SET ix TO 5 is executed */
```

- Raise the SIGTERM condition in your program. The current programming language setting is C.

```
TRIGGER SIGTERM;
```

- A previously defined STATEMENT breakpoint (for line 13) is triggered.

```
AT 13 LIST "at 13";
TRIGGER AT 13;
/* "at 13" will be the echoed output here */
```

- Assume the following breakpoints exist in a program:

```
AT CHANGE x LIST TITLED (x); AT STATEMENT 10;
```

  If Debug Tool is invoked for the STATEMENT breakpoint and you want to trigger the commands associated with the AT CHANGE breakpoint, enter:

```
TRIGGER AT CHANGE x;
```

  Debug Tool displays the value of x.

USE Command

The USE command causes the Debug Tool commands in the specified file to be either performed or syntax checked. This file can be a log file from a previous session. The specified file can itself contain another USE command. The maximum number of USE files open at any time is limited to eight. The USE keyword cannot be abbreviated.

```
►►──USE───file───;──────────────────────────────────────────►◄
```

file A valid file identifier. containing the Debug Tool commands to be performed. It can be a sublibrary member, a filename, or a file-id.

Usage Notes:

- To check the syntax of the commands in a USE file, set the EXECUTE setting to OFF and then issue a USE command for the file.
- Commands read from a USE file are logged as comments.
- The log file can serve as a USE file in a subsequent Debug Tool session.
- Recursive calls are not allowed; that is, a commands file cannot be USEd if it is already active. This includes the primary commands and preferences files. If another invocation of Debug Tool occurs during the execution of a USE file (for example, if a condition is raised while executing a command from a USE file), the USE file is not used for command input until control returns from the condition.
- The USE file is closed when the end of the file is reached.
- If a "nonreturning" command (such as GO) is performed from a USE file, the action taken (as far as closing the USE file) depends on:
  1. If the USE file was invoked directly or indirectly from the primary commands file or preferences file, then it has the same characteristics as the primary commands file. That is, it "keeps its place" and the next time Debug Tool requests a command, it reads from the USE file where it left off.
  2. If the USE file was not invoked directly or indirectly from the primary commands file or preferences file, the rest of the USE file and the file that invoked the USE file is skipped.
- If the end of the USE file is reached without encountering a QUIT command, Debug Tool returns to the command source where the USE command was issued. This can be the terminal, a command string, or another commands file.
- A USE file takes on the aspects of whatever command source issued the USE command, relative to its behavior when a GO, GOTO, or STEP is executed. When invoked from the primary commands file, it continues with its next sequential command at the next breakpoint. If it is invoked from any other command sequence, the GO, GOTO, or STEP causes any remaining commands in the USE file to be discarded.

Examples:

- Perform the Debug Tool commands in the file pointed to by the filename DUSE300 in the following JCL statement.

  ```
  // DLBL DUSE300,'userid.DTCMDS',0

  USE duse300;
  ```

- For CICS, perform Debug Tool commands in the sequential file with the file-id TS64081.USE.FILE.

  ```
  USE TS64081.USE.FILE;
  ```

  In addition to using sequential files, you can perform Debug Tool commands using sublibrary members.

  ```
  USE library.sublibr(member.type);
  ```

while Command (C)

The `while` command enables you to repeatedly perform the body of a loop until the specified condition is no longer met or evaluates to false. The `while` keyword must be lowercase and cannot be abbreviated.

```
▶▶──while──(──expression──)──command──────────────────────────────▶◀
```

expression
A valid Debug Tool C expression.

command
A valid Debug Tool command.

The expression is evaluated to determine whether the body of the loop should be performed. If the expression evaluates to false, the body of the loop never executes. Otherwise, the body does execute. After the body has been performed, control is given once again to the evaluation of the expression. Further execution of the action depends on the value of the condition.

A `break` command can cause the execution of a `while` command to end, even when the condition does not evaluate to false.

Examples:

- List the values of `x` starting at 3 and ending at 9, in increments of 2.

```
x = 1;
while (x +=2, x < 10)
  LIST x;
```

- While `--index` is greater than or equal to zero (`0`), triple the value of the expression `item[index]`.

```
while (--index >= 0) {
  item[index] *= 3;
  printf("item[%d] = %d\n", index, item[index]);
}
```

WINDOW Command (Full-Screen Mode)

The WINDOW command provides window manipulation functions. WINDOW commands can be made immediately effective with the IMMEDIATE command. The cursor-sensitive form is most useful when assigned to a PF key. The WINDOW keyword is optional.

The various forms of the WINDOW command are summarized in Table 15.

Table 15 (Page 1 of 2). Summary of WINDOW Commands

Command	Description
WINDOW CLOSE	closes the specified window in the Debug Tool full-screen session panel.
WINDOW OPEN	opens a previously-closed window in the Debug Tool full-screen session panel.
WINDOW SIZE	controls the relative size of currently visible windows in the Debug Tool full-screen session panel.

Table 15 (Page 2 of 2). Summary of WINDOW Commands	
WINDOW ZOOM	expands the indicated window to fill the entire screen.

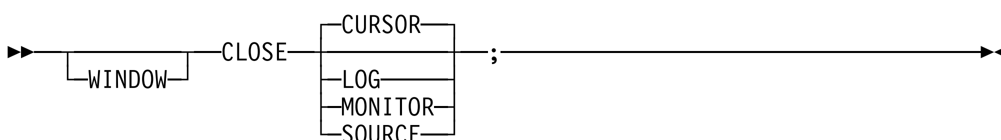
Usage Notes:

- If no operand is specified and the cursor is on the command line, then the default window id set by SET DEFAULT WINDOW is used (if it is open, otherwise the precedence is SOURCE, LOG, MONITOR).
- The WINDOW command is not logged.

WINDOW CLOSE

Closes the specified window in the Debug Tool full-screen session panel. The remaining open windows expand to fill the remainder of the screen. Closing a window does not effect the contents of that window. For example, closing the Monitor window does not stop the monitoring of variable values assigned by the LIST MONITOR command.

If there is only one window visible, WINDOW CLOSE is invalid.



CURSOR
Selects the window where the cursor is currently positioned unless on the command line.

LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the source listing window.

Example:

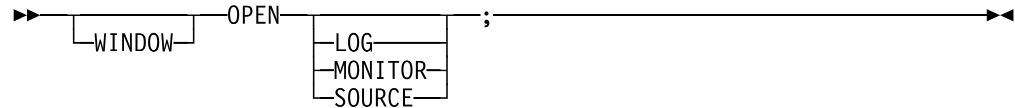
Close the window containing the cursor.

```
WINDOW CLOSE CURSOR;
```

WINDOW OPEN

Opens a previously-closed window in the Debug Tool full-screen session panel. Any existing windows are resized according to the configuration selected with the PANEL LAYOUT command.

If the OPEN command is issued without an operand, Debug Tool opens the last closed window.



LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the source listing window.

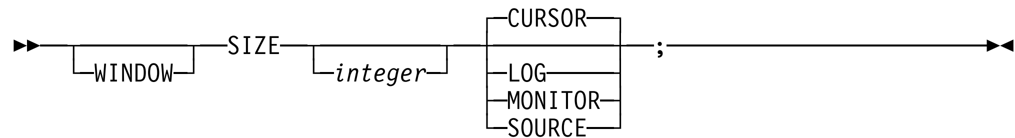
Example:

Open the Monitor window.

```
WINDOW OPEN MONITOR;
```

WINDOW SIZE

Controls the relative size of currently visible windows in the Debug Tool full-screen session panel.



integer
Specifies the number of rows or columns, as appropriate for the selected window and the current window configuration.

CURSOR
Selects the window where the cursor is currently positioned unless on the command line. The cursor form of WINDOW SIZE applies to that window if *integer* is specified. Otherwise, it redraws the configuration of windows so that the intersection of the windows is at the cursor, or if the configuration does not have a common intersection, so that the nearest border is at the cursor.

LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the Source listing window.

Usage Notes:

- You cannot use WINDOW SIZE if a window is ZOOMed or if there is only one window open.
- Each window in any configuration has only one adjustable dimension:
  - If one or more windows are as wide as the screen:
    - The number of rows is adjustable for each window as wide as the screen
    - The number of columns is adjustable for the remaining windows
  - If one or more windows are as high as the screen:
    - The number of columns is adjustable for each window as high as the screen
    - The number of rows is adjustable for the remaining windows

Examples:

- Adjust the size of the Source window to 15 rows.

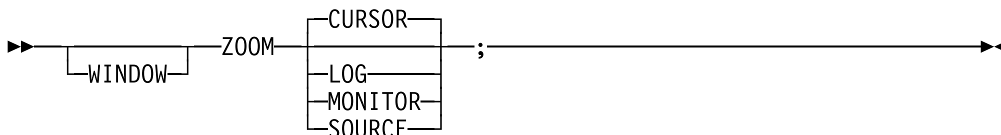
  ```
  WINDOW SIZE 15 SOURCE;
  ```

- Adjust the size of the window where the cursor is currently positioned to 20 rows.

  ```
  SIZE 20 CURSOR;
  ```

WINDOW ZOOM

Expands the indicated window to fill the entire screen or restores the screen to the currently defined window configuration.



CURSOR
Selects the window where the cursor is currently positioned unless on the command line.

LOG
Selects the session Log window.

MONITOR
Selects the Monitor window.

SOURCE
Selects the source listing window.

If the selected window is currently ZOOMed then the zoom mode is toggled. That is, the currently defined window configuration is restored.

Example:

Expand the Log window.

```
WINDOW ZOOM LOG;
```

Part 4. Appendixes

© Copyright IBM Corp. 1995, 1996

Appendix A. Coexistence

This appendix discusses Debug Tool's level of coexistence with other HLL debug tools, and the amount of debugging support you can expect for previous versions of debuggable languages.

Coexistence with Other Debug Tools

Coexistence of Debug Tool with other HLL debug tools cannot be guaranteed.

C, COBOL, and PL/I are dependent upon LE/VSE to provide debugging information.

Another debug tool might provide limited services for a HLL not yet supported by Debug Tool, but conditions such as exceptions cause LE/VSE to pass control to an installed LE/VSE debug tool.

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.

COBOL programs compiled with VS COBOL II should work successfully with Debug Tool, especially if they are link-edited with the LE/VSE run-time environment. Refer to VS COBOL II Application Programming Guide for VSE for further details on compiling with VS COBOL II.

© Copyright IBM Corp. 1995, 1996

Appendix B. Using Debug Tool in a Production Mode

This appendix 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.

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 hooks**

  One option for increasing the performance of your program is to compile with a minimum of hooks or with no hooks. Compiling with the option TEST(NOLINE, BLOCK, NOPATH) for C programs and TEST(BLOCK) for COBOL programs causes the compiler to insert a minimum number of hooks while still allowing you to perform tasks at block boundaries.

  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.

  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, COBOL, 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.

Removing Hooks, Statement Tables, and Symbol Tables

Debug Tool can also gain control at program initialization via the PROMPT suboption of the run-time TEST option. Even if you decide to remove 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 run-time TEST option, or when a call to `__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

© Copyright IBM Corp. 1995, 1996

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 enclosing block:

```
     list (%ADDRESS, %EPA);  (where %EPA 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 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;
```

- Continue your program processing:

```
     GO;
```

- End your program processing:

```
     QUIT;
```

If your program does not contain a statement or symbol table, you can use temporary variables to make the task of examining values of variables easier.

Even in this limited environment, HLL library routines are still available.

Using Debug Tool on Optimized Programs

If you want to debug your application program with Debug Tool after compiling with the compile-time OPTIMIZE option (where applicable), you must keep in mind that optimization decreases the reliability of Debug Tool functions.

In the case of variable values, Debug Tool displays the contents of the storage where the variable has been assigned. However, in an optimized program, the variable might actually be residing in a register. As an example, consider the following assignments:

```
a = 5
b = a + 3
```

In an optimized program, the value of "5" associated with the variable `a` might never be placed into storage. Instead, it might be pulled from a machine register. If Debug Tool is requested to `LIST TITLED a;`, however, it looks in the storage assigned to `a` and displays that value, no matter what it is.

LIST STATEMENT NUMBERS shows the statements that can be used in AT and GOTO commands. Optimization has a similar effect when trying to determine the source statement associated with a specific storage location. Normally, the statement table supplies this information to Debug Tool, but if you request optimization, the statement table might be incorrect. Code associated with one statement can move to another storage location, and can appear (according to the statement table) to be part of a completely different statement. Therefore, the statement number Debug Tool displays as associated with a particular breakpoint might be incorrect.

Also, if you have requested that your application be optimized, Debug Tool cannot guarantee that a breakpoint set at a particular statement indeed occurs at the beginning of the code generated for that statement.

Finally, optimization usually causes the code generated for a statement to be dependent on register values loaded by the code for preceding statements. Thus, if you request Debug Tool to change the path of flow in your program, you run the risk of depriving statements of necessary input.

Appendix C. Using C Reference Information with Debug Tool

This appendix contains reference information for use when debugging C programs with Debug Tool.

Debug Tool Interpretive Subset of C Commands

Table 16 lists the Debug Tool interpretive subset of C commands. This subset is a list of commands recognized by Debug Tool that either closely resemble or duplicate the syntax and action of the C command. This subset of commands is valid only when the current programming language is set to C.

Table 16. Debug Tool Interpretive Subset of C Commands

Command	Description
block ({})	Composite command grouping
break	Termination of loops or `switch` commands
Declaration	Declaration of session variables
do/while	Iterative looping
Expression	Any C expression except the conditional (?) operator
for/while	Iterative looping
if	Conditional execution
switch	Conditional execution

C Reserved Keywords

Table 17 lists all keywords reserved by the C language. These keywords cannot be abbreviated, used as variable names, or used as any other type of identifiers.

Table 17. C Reserved Keywords

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Operators and Operands

Table 18 on page 347 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 18 on page 347 lists the C operators and their orders of precedence.

© Copyright IBM Corp. 1995, 1996

Table 18. Operator Precedence and Associativity

Precedence Level	Associativity	Operators
Primary	left to right	() [ ] . ->
Unary	right to left	++ -- - + ! ˜ & * (*typename*) sizeof
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise Logical AND	left to right	&
Bitwise Exclusive OR	left to right	∧ or ¬
Bitwise Inclusive OR	left to right	¦
Logical AND	left to right	&&
Logical OR	left to right	¦¦
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^= ¦=
Comma	left to right	,

LE/VSE Conditions and Their C Equivalents

LE/VSE condition names (the symbolic feedback codes CEExxx) can be used interchangeably with the equivalent C conditions listed in Table 19. 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 19 (Page 1 of 2). LE/VSE Conditions and Their C Equivalents

LE/VSE Condition	Description	Equivalent C condition
CEE341	Operation exception	SIGILL
CEE342	Privileged operation exception	SIGILL
CEE343	Execute exception	SIGILL
CEE344	Protection exception	SIGSEGV
CEE345	Addressing exception	SIGSEGV
CEE346	Specification exception	SIGILL
CEE347	Data exception	SIGFPE
CEE348	Fixed point overflow exception	SIGFPE
CEE349	Fixed point divide exception	SIGFPE
CEE34A	Decimal overflow exception	SIGFPE
CEE34B	Decimal divide exception	SIGFPE
CEE34C	Exponent overflow exception	SIGFPE
CEE34D	Exponent underflow exception	SIGFPE

Table 19 (Page 2 of 2). LE/VSE Conditions and Their C Equivalents

LE/VSE Condition	Description	Equivalent C condition
CEE34E	Significance exception	SIGFPE
CEE34F	Floating-point divide exception	SIGFPE

Appendix D. Using COBOL Reference Information with Debug Tool

This appendix contains reference information for use when debugging COBOL programs with Debug Tool.

Debug Tool Interpretive Subset of COBOL Commands

Table 20 lists the Debug Tool interpretive subset of COBOL language commands. This subset is a list of commands recognized by Debug Tool that either closely resemble or duplicate the syntax and action of the appropriate COBOL command. This subset of commands is valid only when the current programming language is COBOL.

Table 20. Debug Tool Interpretive Subset of COBOL Commands

Command	Description
CALL	Subroutine call
COMPUTE	Computational assignment (including expressions)
Declaration	Declaration of session variables
EVALUATE	Multiway switch
IF	Conditional execution
MOVE	Noncomputational assignment
PERFORM	Iterative looping
SET	INDEX and POINTER assignment

COBOL Reserved Keywords

In addition to the subset of COBOL commands you can use while in Debug Tool, there is a list of reserved keywords used and recognized by COBOL that cannot be abbreviated, used as a variable name, or used as any other type of identifier. You can find this list in the various COBOL language references.

Allowable Comparisons for the Debug Tool IF Command

Table 21 shows the allowable comparisons for the Debug Tool IF command. A description of the codes follows the table.

Table 21 (Page 1 of 2). Allowable Comparisons for the Debug Tool IF Command

OPERAND	GR	AL	AN	ED	BI	NE	ANE	ID	IN	IDI	PTR	@	IF	EF	D1
GROUP (GR)	NN	NN	NN	NN	NN	NN	NN	NN		NN			NN	NN	
ALPHABETIC (AL)	NN	NN													
ALPHANUMERIC (AN)	NN		NN												
EXTERNAL DECIMAL (ED)	NN			NU											

© Copyright IBM Corp. 1995, 1996

Table 21 (Page 2 of 2). Allowable Comparisons for the Debug Tool IF Command

OPERAND	GR	AL	AN	ED	BI	NE	ANE	ID	IN	IDI	PTR	@	IF	EF	D1
BINARY	NN				NU				NU[4]						
NUMERIC EDITED (NE)	NN					NN									
ALPHANUMERIC EDITED (ANE)	NN						NN								
FIGCON ZERO[7]	NN			NU	NU			NU					NU	NU	
FIGCON[1,7]	NN	NN	NN				NN								
NUMERIC LITERAL[7]	NN			NU	NU			NU	NU[4]				NU	NU	
NONNUMERIC LITERAL[2,7]	NN	NN[3]	NN			NN	NN								
INTERNAL DECIMAL (ID)	NN							NU							
INDEX NAME (IN)	NN				NU[4]				IO[4]	NU					
INDEX DATA ITEM (IDI)	NN								NU	IV					
POINTER DATA ITEM (PTR)											NU[5]	NU[5]			
ADDRESS OF (@)											NU[5]	NU[5]			
FLOATING POINT LITERAL[7]	X												NU	NU	
INTERNAL FLOATING POINT (IF)	NN												NU	NU	
EXTERNAL FLOATING POINT (EF)	NN												NU	NU	
DBCS DATA ITEM (D1)															NN
DBCS LITERAL[7]															NN
HEX LITERAL[6]											NU[5]				

Notes:

1 FIGCON includes all figurative constants except ZERO and ALL.
2 A nonnumeric literal must be enclosed in quotation marks, and the quotation marks are not valid characters in the literal.
3 Must contain only alphabetic characters.
4 Index name converted to subscript value before compare.
5 Only comparison for equal and not equal can be made.
6 Must be hexadecimal characters only, delimited by either double (") or single (') quotation marks and preceded by *H*.
7 Constants and literals can also be compared against constants and literals of the same type.

Allowable comparisons are comparisons as described in *IBM OS Full American National Standard COBOL* for the following:

NN Nonnumeric operands

NU Numeric operands

IO Two index names

IV Index data items

X High potential for user error

Allowable Moves for the Debug Tool MOVE Command

Table 22 shows the allowable moves for the Debug Tool MOVE command.

Table 22. Allowable Moves for the Debug Tool MOVE Command

Receiving Field / **Source Field**	**GR**	**AL**	**AN**	**ED**	**BI**	**NE**	**ANE**	**ID**	**IF**	**EF**	**D1**
GROUP (GR)	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	
ALPHABETIC (AL)	Y	Y									
ALPHANUMERIC (AN)	Y		Y								
EXTERNAL DECIMAL (ED)	Y[1]			Y							
BINARY (BI)	Y[1]				Y						
NUMERIC EDITED (NE)	Y										
ALPHANUMERIC EDITED (ANE)	Y						Y				
FIGCON ZERO	Y		Y	Y[2]	Y[2]		Y	Y[2]	Y	Y	
SPACES (AL)	Y	Y	Y				Y				
HIGH-VALUE, LOW-VALUE, QUOTES	Y		Y				Y				
NUMERIC LITERAL	Y[1]			Y	Y			Y	Y	Y	
NONNUMERIC LITERAL	Y	Y	Y			Y[1]	Y				
INTERNAL DECIMAL (ID)	Y[1]							Y			
FLOATING POINT LITERAL	Y[1]								Y	Y	
INTERNAL FLOATING POINT (IF)	Y[1]								Y	Y	
EXTERNAL FLOATING POINT (EF)	Y[1]								Y	Y[3]	
DBCS DATA ITEM (D1)											Y
DBCS LITERAL											Y

Notes:

1 Move without conversion (like AN to AN)
2 Numeric move
3 Decimal-aligned and truncated, if necessary

For more information, see “MOVE Command (COBOL)” on page 286.

Allowable Moves for the Debug Tool SET Command

Table 23 shows the allowable moves for the Debug Tool SET command.

Table 23. Allowable Moves for the Debug Tool SET Command

Receiving Field / **Source Field**	**IN**	**IDI**	**PTR**	**ED**	**BI**	**ID**	**OR**
INDEX NAME (IN)	Y	Y		Y	Y	Y	
INDEX DATA ITEM (IDI)	Y	Y					
POINTER DATA ITEM (PTR)			Y				
HEX LITERAL[1]			Y				
NULL (NUL)			Y				
INTEGER LITERAL	Y[2]						
EXTERNAL DECIMAL (ED)	Y						
BINARY (BI)	Y						
INTERNAL DECIMAL (ID)	Y						
OBJECT REFERENCE (OR)							Y

Notes:

1 Must be hexadecimal characters only, delimited by either double (") or single (') quotation marks and preceded by *H*.
2 Index name is converted to index value.

Appendix E. Using PL/I Reference Information with Debug Tool

This appendix contains reference information for use when debugging PL/I programs with Debug Tool.

Debug Tool Interpretive Subset of PL/I Commands

Table 24 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 24. Debug Tool Subset of PL/I Commands

Command	Description
Assignment	Scalar and vector assignment
BEGIN	Composite command grouping
CALL	Debug Tool procedure call
DECLARE or **DCL**	Declaration of session variables
DO	Iterative looping and composite command grouping
IF	Conditional execution
ON	Define an exception handler
SELECT	Conditional execution

These PL/I language-oriented commands are only a subset of all the commands that are supported by Debug Tool.

PL/I Reserved Keywords

In addition to the subset of PL/I commands you can use while in Debug Tool, there is a list of reserved keywords used and recognized by PL/I that cannot be abbreviated, used as a variable name, or used as any other type of identifier. You can find this list in *IBM PL/I for VSE/ESA Language Reference*.

Conditions and Condition Handling

All PL/I conditions are recognized by Debug Tool. They are used with the AT OCCURRENCE and ON commands. See “AT OCCURRENCE” on page 225 and “ON Command (PL/I)” on page 287.

When an OCCURRENCE breakpoint is triggered, the Debug Tool %CONDITION variable holds the following values:

Table 25 (Page 1 of 2). PL/I Conditions and %CONDITION Values

Triggered Condition	%CONDITION Value
AREA	AREA

© Copyright IBM Corp. 1995, 1996

Table 25 (Page 2 of 2). PL/I Conditions and %CONDITION Values

Triggered Condition	%CONDITION Value
COND	CONDITION
CONVERSION	CONVERSION
ENDFILE (MF)	ENDFILE
ENDPAGE (MF)	ENDPAGE
ERROR	ERROR
FINISH	CEE066
FOFL	CEE348
KEY (MF)	KEY
NAME (MF)	NAME
OVERFLOW	CEE34C
RECORD (MF)	RECORD
SIZE	SIZE
STRG	STRINGRANGE
STRINGSIZE	STRINGSIZE
SUBRG	SUBSCRIPTRANGE
TRANSMIT (MF)	TRANSMIT
UNDEFINEDFILE (MF)	UNDEFINEDFILE
UNDERFLOW	CEE34D
ZERODIVIDE	CEE349

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)

Appendix F. Debug Tool Messages

All messages in this appendix are shown in ENGLISH format. The UENGLISH format message text is the same, but is in uppercase letters.

Each message has a number of the form EQAnnnnx, where EQA indicates that the message is an Debug Tool message, nnnn is the number of the message, and x indicates the severity level of each message. The value of x is I, W, E, S, or U, as described below:

I An ***informational*** message calls attention to some aspect of a command response that might assist the programmer.

W A ***warning*** message calls attention to a situation that might not be what is expected or to a situation that Debug Tool attempted to fix.

E An ***error*** message describes an error that Debug Tool detected or cannot fix.

S A ***severe*** error message describes an error that indicates a command referring to bad data, control blocks, program structure, or something similar.

U An ***unrecoverable*** error message describes an error that prevents Debug Tool from continuing.

Many of the Debug Tool messages contain information that is inserted by the system when the message is issued. In this publication, such inserted information is indicated by highlighted symbols, as shown by *breakpoint-id* in the following example:

EQA1046I The *breakpoint-id* breakpoint is replaced.

EQA1001I The window configuration is *configuration* ; the sequence of window is *sequence*

Explanation: Used to display SCREEN as part of QUERY SCREEN.

EQA1002I One window must be open at all times.

Explanation: Only one window was open when a CLOSE command was issued. At least one window must be open at all times, so the CLOSE command is ignored.

EQA1003I Target window is closed; FIND not performed.

Explanation: The window specified in the FIND command is closed.

EQA1004I Target window is closed; SIZE not performed.

Explanation: The window specified in the SIZE command is closed.

EQA1005I Target window is closed; SCROLL not performed.

Explanation: The window specified in the SCROLL command is closed.

EQA1006I Command

Explanation: It is the character string 'Command' in the main panel command line.

EQA1007I Step

Explanation: It is the character string 'Step' in the main panel command line while stepping.

EQA1008I Scroll

Explanation: It is the character string 'Scroll' in the main panel command line.

© Copyright IBM Corp. 1995, 1996

EQA1009I DBCS characters are not allowed.

Explanation: The user entered DBCS characters in scroll, window object id, qualify, prefix, or panel input areas.

EQA1010I More...

Explanation: It is the character string 'More' in the main panel command line.

EQA1011I Do you really want to terminate this session?

Explanation: This is for the END pop-up window.

EQA1012I Enter Y for YES and N for NO

Explanation: This is for the END pop-up window. Y, YES, N, and NO should NOT be translated.

EQA1013I Current command is incomplete, pending more input

Explanation: This informational message is displayed while entering a block of commands, until the command block is closed by an END statement.

EQA1030I PENDING:

Explanation: Debug Tool needs more input in order to completely parse a command. This can occur in COBOL, for example, because `PERFORM`; was entered on the last line.

Programmer Response: Complete the command.

EQA1031I The partially parsed command is:

Explanation: The explanation of a command was requested or a command was determined to be in error.

Programmer Response: Determine the cause of the error and reenter the command.

EQA1032I The next word can be one of:

Explanation: This title line will be followed by message 1015.

EQA1033I *list items*

Explanation: This message is used to list all the items that can follow a partially parsed command.

Programmer Response: Reenter the acceptable part of the command and suffix it with one of the items in this list.

EQA1046I The *breakpoint-id* **breakpoint is replaced.**

Explanation: This alerts the user to the fact that a previous breakpoint action existed and was replaced.

Programmer Response: Verify that this was intended.

EQA1047I The *breakpoint-id* **breakpoint is replaced.**

Explanation: This alerts the user to the fact that a previous breakpoint action existed and was replaced.

Programmer Response: Verify that this was intended.

EQA1048I Another generation of *variable name* **is allocated.**

Explanation: An ALLOCATE occurred for a variable where an AT ALLOCATE breakpoint was established.

EQA1049I The *breakpoint-id* **breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is no every_clause. Enabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1050I The *breakpoint-id* **breakpoint has an EVERY value of** *number***, a FROM value of** *number***, and a TO value of** *number***. The breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is an every_clause. Enabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1051I The (deferred) *breakpoint-id* **breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is no every_clause. Deferred and enabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1052I The (deferred) *breakpoint-id* **breakpoint has an EVERY value of** *number***, a FROM value of** *number***, and a TO value of** *number***. The breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is an every_clause. Deferred and enabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1053I The (disabled) *breakpoint-id* **breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is not an every_clause. For disabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1054I The (disabled) *breakpoint-id* **breakpoint has an EVERY value of** *number***, a FROM value of** *number***, and a TO value of** *number***. The breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is an every_clause. For disabled breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1055I The (disabled and deferred) *breakpoint-id* **breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is not an every_clause. For disabled and deferred breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1056I The (disabled and deferred) *breakpoint-id* **breakpoint has an EVERY value of** *number***, a FROM value of** *number***, and a TO value of** *number***. The breakpoint action is:**

Explanation: Used to display a command after LIST AT when there is an every_clause. For disabled and deferred breakpoints only. This message is followed by a message of one or more lines showing the commands performed each time the breakpoint is hit.

EQA1057I AT *stmt-number* **can be risky because the code for that statement may have been merged with that of another statement.**

Explanation: You are trying to issue an AT STATEMENT command against a statement but the code for that statement was either optimized away or combined with other statements.

EQA1076I *Direction* **an unknown program.**

Explanation: The program can be written in assembler language or in an unsupported language. The message is issued as a result of the LIST CALLS command.

EQA1077I *Direction* **address** *Address* **in a** *PLANG* **NOTEST block.**

Explanation: The compile unit was compiled without the TEST option. The message is issued as a result of the LIST CALLS command.

EQA1078I *Direction Place* **in** *PLANG CU***.**

Explanation: CU name of the call chain. The message is issued as a result of the LIST CALLS command.

EQA1086I The previous declaration of *variable name* **will be removed.**

Explanation: You declared a variable whose name is the same as a previously declared variable. This declaration overrides the previous one.

EQA1090I The compile-time data for program *cu_name* **is**

Explanation: This is the title line for the DESCRIBE PROGRAM command.

EQA1091I The program was compiled with the following options:

Explanation: This is the first of a group of DESCRIBE PROGRAM messages.

EQA1092I *compile option*

Explanation: Used to display a compile option without parameters, for example, NOTEST.

EQA1093I *compile option* **(***compile suboption***)**

Explanation: Used to display a compile option with one parameter, for example, OPT.

EQA1094I *compile option* **(***compile suboption***,** *compile suboption***)**

Explanation: Used to display a compile option with two parameters, for example, TEST.

EQA1095I This program has no subblocks.

Explanation: A DESCRIBE PROGRAM command refers to a program that is totally contained in one block.

EQA1096I The subblocks in this program are nested as follows:

Explanation: The names of the blocks contained by the program are displayed under this title line.

EQA1097I *space characters block name*

Explanation: The first insert controls the indentation while the second is the block name without qualification.

EQA1098I The statement table has the short format.

Explanation: The statement table is abbreviated such that no relationship between storage locations and statement identifications can be determined.

Programmer Response: If statement identifications are required, the program must be recompiled with different compile-time parameters.

EQA1099I The statement table has the NUMBER format.

Explanation: The program named in the DESCRIBE PROGRAM command was compiled with GONUMBER assumed.

EQA1100I The statement table has the STMT format.

Explanation: The program named in the DESCRIBE PROGRAM command was compiled with GOSTMT assumed.

EQA1101I *file name*

Explanation: This message is used in listing items returned from the back end in response to the DESCRIBE ENVIRONMENT command.

EQA1102I ATTRIBUTES for *variable name*

Explanation: Text of a DESCRIBE ATTRIBUTES message.

EQA1103I Its address is *address*

Explanation: Text of a DESCRIBE ATTRIBUTES message.

EQA1104I Compiler: *Compiler version*

Explanation: Indicate compiler version for DESCRIBE CU.

EQA1105I Its length is *length*

Explanation: Text of a DESCRIBE ATTRIBUTES message.

EQA1106I Programming language COBOL does not return information for DESCRIBE ENVIRONMENT

Explanation: COBOL run-time library does not return information to support this command.

EQA1107I There are no open files.

Explanation: This is issued in response to DESCRIBE ENVIRONMENT if no open files are detected.

EQA1108I The following conditions are enabled:

Explanation: This is the header message issued in response to DESCRIBE ENVIRONMENT before issuing the list of enabled conditions.

EQA1109I The following conditions are disabled:

Explanation: This is the header message issued in response to DESCRIBE ENVIRONMENT before issuing the list of disabled conditions.

EQA1110I This program has no Statement Table.

Explanation: This message is used for the DESCRIBE CU command. If a CU was compiled with NOTEST, no statement table was generated.

EQA1111I Attributes for names in block *block name*

Explanation: This is a title line that is the result of a DESCRIBE ATTRIBUTES *;. It precedes the names of all variables contained within a single block.

EQA1112I *variable name and/or attributes*

Explanation: The first insert controls the indentation while the second is the qualified variable name followed by attribute string. (for C, only the attributes are given.)

EQA1114I Currently open files are:

Explanation: This is the title line for the list of files that are known to be open. This is in response to the DESCRIBE ENVIRONMENT command.

EQA1115I The program has insufficient compilation information for the DESCRIBE CU command.

Explanation: This program has insufficient information. It may be compiled without the test option.

EQA1116I Common LE/VSE math library is being used

Explanation: This is the response for the DESCRIBE ENVIRONMENT command when the LE/VSE math library is being used.

EQA1117I PL/I Math library is being used

Explanation: This is the response for the DESCRIBE ENVIRONMENT command when the PL/I math library is being used.

EQA1140I *character*

Explanation: This message is used to produce output for LIST (...).

EQA1141I *expression name* **=** *expression value*

Explanation: This message is used to produce output for LIST TITLED (...) when an expression is a scalar.

EQA1142I *expression element*

Explanation: This insert is used for naming the expression for expression element.

EQA1143I >>> EXPRESSION ANALYSIS <<<

Explanation: First line of output from the ANALYZE EXPRESSION command

EQA1144I *alignment spaces* **It is a bit field with offset** *bit offset***.**

Explanation: Text of a DESCRIBE ATTRIBUTES message.

EQA1145I Its Offset is *offset***.**

Explanation: Text of a DESCRIBE ATTRIBUTES message.

EQA1146I *column elements*

Explanation: This message is used to produce a columned list. For example, it is used to format the response to LIST STATEMENT NUMBERS.

EQA1147I *name*

Explanation: The name of a variable that satisfies a LIST NAMES request is displayed.

EQA1148I *name* **structure**

Explanation: The name of a variable that satisfies a LIST NAMES request is displayed. It is contained within an aggregate but is a parent name and not an elemental data item.

EQA1149I *name* **in** *parent name*

Explanation: The name of a variable that satisfies a LIST NAMES request is displayed. It is contained within an aggregate and is an elemental data item.

EQA1150I *name* **structure in** *parent name*

Explanation: The name of a variable that satisfies a LIST NAMES request is displayed. It is an aggregate contained within another aggregate.

EQA1151I The following names are known in block *block name*

Explanation: This is a title line that is the result of a LIST NAMES command. It precedes the names of all variables contained within a single block.

EQA1152I The following session names are known

Explanation: This is a title line that is the result of a LIST NAMES command. It precedes the names of all session variables contained within a single block.

EQA1153I The following names with pattern *pattern* **are known in** *block name*

Explanation: This title line precedes the list of variable names that satisfy the pattern in a LIST NAMES command.

EQA1154I The following session names with pattern *pattern* **are known**

Explanation: This title line precedes the list of session names that satisfy the pattern in a LIST NAMES command.

EQA1155I The following CUs are known in *Phase name***:**

Explanation: This title line precedes a list of compile unit names for noninitial phases in a LIST NAMES CUS command.

EQA1156I The following CUs with pattern *pattern* **are known in** *Phase name*

Explanation: This title line precedes a list of compile unit names for noninitial phases that satisfy the pattern in a LIST NAMES CUS command.

EQA1157I There are no CUs with pattern *pattern* **in** *Phase name***.**

Explanation: This line appears when no compile unit satisfied the pattern in a LIST NAMES CUS command for noninitial phases.

EQA1158I The following CUs have pattern *pattern*

Explanation: This title line precedes a list of compile unit names for an initial phase in a LIST NAMES CUS command.

EQA1159I There are no CUs with pattern *pattern***.**

Explanation: This line appears when no compile unit satisfied the pattern in a LIST NAMES CUS command for an initial phase.

EQA1160I There are no Procedures with pattern *pattern***.**

Explanation: This line appears when no Procedures satisfied the pattern in a LIST NAMES PROCEDURES command.

EQA1161I The following Procedures have pattern *pattern***:**

Explanation: This title line precedes a list of Procedure names for a LIST NAMES PROCEDURES command.

EQA1162I There are no names in block *block name*

Explanation: The LIST NAMES command found no variables in the specified block.

EQA1163I There are no session names.

Explanation: The LIST NAMES command found no variables that had been declared in the session for the current programming language.

EQA1164I There are no names with pattern *pattern* **in** *block name***.**

Explanation: The LIST NAMES command found named variables in the named block but none of the names satisfied the pattern.

EQA1165I There are no session names with pattern *pattern***.**

Explanation: The LIST NAMES command found named variables that had been declared in the session but none of the names satisfied the pattern.

EQA1166I There are no known session procedures.

Explanation: A LIST NAMES PROCEDURES was issued but no session procedures exist.

EQA1167I *register name* **=** *register value*

Explanation: Used when listing registers.

EQA1168I No LIST STORAGE data is available for the requested reference or address.

Explanation: The given reference or address is invalid.

EQA1169I No frequency data is available

Explanation: This message is issued upon failure to find frequency information.

EQA1170I Frequency of verb executions in *cu_name*

Explanation: This is the header produced by the LIST FREQUENCY command.

EQA1171I *character string* **=** *count*

Explanation: This is the frequency count produced by the LIST FREQUENCY command.

EQA1172I TOTAL VERBS= *total statements***, TOTAL VERBS EXECUTED=** *total statements executed***, PERCENT EXECUTED=** *percent executed*

Explanation: This is the trailer produced by the LIST FREQUENCY command.

EQA1173I (*history number***) ENTRY hook for** *cu_name*

Explanation: This is a LIST HISTORY message.

EQA1174I (*history number***) ENTRY hook for block** *block name* **in** *cu_name*

Explanation: This is a LIST HISTORY message.

EQA1175I (*history number***) EXIT hook for** *cu_name*

Explanation: This is a LIST HISTORY message.

EQA1176I (*history number***) EXIT hook for block** *block name* **in** *cu_name*

Explanation: This is a LIST HISTORY message.

EQA1177I (*history number***) STATEMENT hook at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1178I (*history number***) PATH hook at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1179I (*history number***) Before CALL hook at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1180I (*history number***) CALL CEETEST at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1181I (*history number***) Waiting for program input from** *filename*

Explanation: This is a LIST HISTORY message.

EQA1182I (*history number***) LOAD occurred at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1183I (*history number***) DELETE occurred at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1184I (*history number***)** *condition name* **raised at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1185I (*history number***) LABEL hook at statement** *cu_name* **:>** *statement_id*

Explanation: This is a LIST HISTORY message.

EQA1186I Unable to display value of *variable name***. Use LIST (***variable name***) for further details**

Explanation: This is used to inform the user that for some reason one of the variables cannot be displayed for LIST TITLED.

EQA1187I There are no data members in the requested object.

Explanation: The requested object does not contain any data members. It contains only methods.

EQA1226I The EQUATE named *EQUATE name* **is replaced.**

Explanation: This alerts the user to the fact that a previous EQUATE existed and was replaced.

Programmer Response: Verify that this was intended.

EQA1227I The following EQUATE definitions are in effect:

Explanation: This is the header for the QUERY EQUATES command.

EQA1228I *EQUATE identifier* **= "** *EQUATE string***"**

Explanation: Used to display EQUATE identifiers and their associated strings. The string is enclosed in quotation marks so that any leading or trailing blanks are noticeable.

EQA1229I The program is currently exiting block *block name***.**

Explanation: Shows the bearings in an interrupted program.

EQA1230I The program is currently executing prolog code for *block name***.**

Explanation: Shows the bearings in an interrupted program.

EQA1231I You are executing commands within a __ctest function.

Explanation: Shows the bearings in an interrupted program.

EQA1232I You are executing commands within a CEETEST function.

Explanation: Shows the bearings in an interrupted program.

EQA1233I The established MONITOR commands are:

Explanation: This is the header produced by LIST MONITOR.

EQA1234I MONITOR *monitor number monitor type*

Explanation: This is the line produced by LIST MONITOR before each command is displayed

EQA1235I The command for MONITOR *monitor number monitor type* **is:**

Explanation: This is the header produced by LIST MONITOR *monitor number*.

EQA1236I The MONITOR *monitor number* **command is replaced.**

Explanation: This is a safety message: the user is reminded that a MONITOR command is replacing an old one.

EQA1237I The current qualification is *block name***.**

Explanation: Shows the current point of view.

EQA1238I The current location is *cu_name* **:>** *statement id***.**

Explanation: Shows the place where the program was interrupted.

EQA1239I The program is currently entering block *block name***.**

Explanation: Shows the bearings in an interrupted program.

EQA1240I You are executing commands within a CALL PLITEST statement.

Explanation: Shows the bearings in an interrupted program.

EQA1241I You are executing commands from the run-time command-list.

Explanation: Shows the bearings in an interrupted program.

EQA1242I You are executing commands in the *breakpoint-id* **breakpoint.**

Explanation: Shows the bearings in an interrupted program.

EQA1243I The setting of *SET-command object* **is** *status*

Explanation: The status of the object of a SET command is displayed when QUERYed individually.

EQA1244I *SET-command object status*

Explanation: The status of the object of a SET command is displayed when issued as part of QUERY SET.

EQA1245I The current settings are:

Explanation: This is the header for QUERY SET.

EQA1246I *PFKEY string command*

Explanation: Used to display PFKEYS as part of QUERY PKFEYS.

EQA1247I *colored area color hilight intensity*

Explanation: Used to display SCREEN as part of QUERY SCREEN.

EQA1248I You were prompted because STEP ended.

Explanation: Shows the bearings in an interrupted program.

EQA1249I *character string* **- The QUERY source setting file name is not available.**

Explanation: The source listing file is not available. The source listing was not required or set prior to this command.

EQA1250I SET INTERCEPT is already set *on or off* **for FILE** *filename***.**

Explanation: You tried to issue the SET INTERCEPT ON/OFF for a file that is already set to ON/OFF. This is just an informational message to notify you that you are trying to duplicate the current setting. The command is ignored.

EQA1276I TEST:

Explanation: Debug Tool is ready to accept a command from the terminal.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information is displayed identifying the commands you are allowed to enter.

EQA1277I The USE file is empty.

Explanation: Debug Tool tried to read commands from an empty USE file. If unintentional, this could be because of an incorrect file specification.

Programmer Response: Correct the file specification and retry.

EQA1278I *alignment spaces command part*

Explanation: This is part of a command that is being displayed in the log or in response to a LIST AT. Since a group of commands can be involved, their appearance is improved by indenting the subgroups. Therefore, the first insert is used for indentation, and the second to contain the command. This is the command as it is understood by Debug Tool.

- Truncated keywords are no longer truncated.
- Lowercase to uppercase conversion was done where appropriate.
- Only a single command is contained in a record. If multiple commands are involved, additional records are prepared using this format.

EQA1286I (Application program has terminated)

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in full-screen mode when an initial prompt occurs at the termination of the application program.

EQA1287I Unknown

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in full-screen mode when an initial prompt occurs and the location is unknown.

EQA1288I initialization

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in full-screen mode when an initial prompt occurs after Debug Tool initialization and before any program hooks are reached.

EQA1289I *filename***:** *program output*

Explanation: Displays program output with the filename (DLBL) preceding the output.

EQA1290I The program is waiting for input from *filename*

Explanation: Debug Tool has gained control because the program is waiting for input.

EQA1291I Use the INPUT command to enter *recsize* **characters for the intercepted fixed-format file.**

Explanation: Prompts you for intercepted input of fixed-format file.

EQA1292I Use the INPUT command to enter up to a maximum of *recsize* **characters for the intercepted variable-format file.**

Explanation: Prompt user for intercepted input of variable-formatted file.

EQA1306I You were prompted because the *CONDITION name* **condition was raised in your program.**

Explanation: The program has stopped running due to the occurrence of the named condition.

EQA1307I You were prompted because an attention interrupt occurred.

Explanation: The attention request from the terminal was recognized and the Debug Tool was given control.

EQA1308I You were prompted because a condition was raised in your program.

Explanation: The program stopped running due to the occurrence of a condition whose name is unknown.

EQA1309I *CONDITION name* **is a severity or class** *SEVERITY code* **condition.**

Explanation: The condition named is described by its severity level or class code. See *LE/VSE Programming Guide*.

EQA1316I Block *block name* **contains the following statements:**

Explanation: This message precedes the message that identifies all statement numbers in the block.

EQA1317I *block level space characters block name*

Explanation: This message is used instead of EQA1097I when the number of block levels is greater than the indentation allowed.

EQA1326I *character string*

Explanation: This message is used during product development and service.

EQA1327I *character string character string*

Explanation: This message is used during product development and service.

EQA1329I The procedure named *procedure name* **has the form:**

Explanation: This is the information that is produced when a LIST PROCEDURE command is processed. This message is followed by a message of one or more lines showing the commands that form the procedure.

EQA1330I You are not currently within a procedure.

Explanation: The LIST PROCEDURE command was issued without naming a session procedure and the current command context is outside of a session procedure.

Programmer Response: Verify the request. Reenter the command and name a specific procedure if necessary.

EQA1331I The RETRIEVE queue is empty.

Explanation: There are no entries in the retrieve queue.

EQA1332I FIND has continued from top of area.

Explanation: FIND searched the file to the end of the string without finding it and continues the search from the top, back to the starting point of the search.

EQA1333I The string was found.

Explanation: FIND was successful in locating the target string.

EQA1334I The operating system has generated the following message:

Explanation: The Operating System can issue its own messages. These are relayed to the user.

EQA1335I *OS message*

Explanation: The operating system can issue its own messages. These are relayed to the user.

EQA1336I Debug Tool for VSE/ESA Version 1 Release 1 Mod 0 *time stamp* **(C) Copyright IBM Corp. 1992, 1996**

Explanation: This message is used to place the Debug Tool logo, a timestamp, and copyright at the beginning of the log.

EQA1337I - Its address is *address* **and its length is** *length*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1338I - Its offset is *offset* **and its length is** *length*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1339I - Its length is *length*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1340I - Its address is *address*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1341I - Its Offset is *offset*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1342I ATTRIBUTES for *variable name variable type*

Explanation: Text of a DESCRIBE ATTRIBUTES message for PL/I.

EQA1343I Presently not in accessible storage

Explanation: The requested variable cannot be accessed.

EQA1344I The OTHERWISE statement would have been executed but was *not present*

Explanation: The was no OTHERWISE clause present in the SELECT statement and none of the WHEN clauses were selected. This message is simply indicating that the OTHERWISE clause would have been executed if it had been present.

EQA1400E The value entered is invalid.

Explanation: The user entered an invalid value.

EQA1401E The command entered is not a valid panel sub-command.

Explanation: The user entered a command not recognized by panel processor.

EQA1402E Each window must have unique letters of L, M, and S.

Explanation: The user entered either duplicated letters or just one letter.

EQA1403E Invalid prefix command was entered.

Explanation: The user entered an invalid prefix command.

EQA1404E Search target not found.

Explanation: The target for the search command was not found.

EQA1405E No previous search arguments exist; find not performed.

Explanation: A FIND command was issued without an argument. Since the FIND command had not been issued previously, Debug Tool had nothing to search for.

EQA1406E Invalid window id

Explanation: The window header field contains an invalid window ID. Valid window IDs are SOURCE, MONITOR, and LOG.

EQA1407E Invalid scroll amount entered.

Explanation: Scroll field contains an invalid scroll amount.

EQA1408E Duplicate window ID

Explanation: More than one window header field contains the same window id.

EQA1430W The EQUATE named *EQUATE name* was has not been established.

Explanation: CLEAR EQUATE <name> was attempted for an EQUATE name that has not been established.

Programmer Response: For a list of the current EQUATES definitions, issue QUERY EQUATES.

EQA1431W There are no EQUATE definitions in effect.

Explanation: CLEAR EQUATE or QUERY EQUATES was issued but there are no EQUATE definitions.

EQA1432E *function* is not supported.

Explanation: Language/Country is not supported.

Programmer Response: Set National Language and Country.

EQA1433E Switching to the programming language *language-name* is invalid because there are no *language-name* compilation units in the initial phase.

Explanation: A SET PROGRAMMING LANGUAGE command was issued, but the initial phase contains no compilation units compiled in the language specified (or implied).

EQA1434E Error in setting debug *name* to *?????????*.

Programmer Response: Refer to the maximum number of CUs allowed for debugging.

EQA1435E Error in setting *name*.

Explanation: This is a generic message for SET command errors.

EQA1436W SET EXECUTE is OFF -- command will not be executed.

Explanation: The command was parsed but not executed.

EQA1450E Unable to display the result from expression evaluation

Explanation: The entire result from the expression evaluation cannot be displayed; for example, the array is too large.

EQA1451E *operand* contains incompatible data type.

Explanation: Comparison or assignment involves incompatible data types.

EQA1452E *argument name* is not a valid argument.

Explanation: The specified argument is not valid.

EQA1453E The number of arguments is not correct.

Explanation: There are either too many or too few arguments specified.

EQA1454E *operand name* is not a valid operand.

Explanation: The specified operand is undefined or is an invalid literal.

EQA1455E An unsupported operator/operand is specified.

Explanation: An operator or an operand was not understood, and therefore was not processed.

EQA1456S The variable *variable name* is undefined or is incorrectly qualified.

Explanation: The named variable could not be located or undefined.

Programmer Response: You need to qualify to a different block in order to locate the variable.

EQA1457E The exponent *exponent* contains a decimal point. This feature is not supported.

Explanation: No decimal point is allowed in exponent specification.

EQA1458E The address of *data item* has been determined to be invalid.

Explanation: This can happen for items within a data record where the file is not active or the record area is not available; for items in a structure following Occurs, depending on the item where the ODO variable was not initialized; or before program initialization.

EQA1459E *literal string* is not a valid literal.

Explanation: The combination of characters specified for the literal is not a valid literal.

EQA1460E Operand *operand name* should be numeric.

Explanation: A non-numeric operand was found where a numeric operand was expected.

EQA1461E Invalid data for *data item* is found.

Explanation: The memory location for a data item contains data that is inconsistent with the data type of the item. The item may not have been initialized.

EQA1462E Invalid sign for ***data item*** **is found.**

Explanation: The sign position of a signed data item contains an invalid sign. The item may not have been initialized.

EQA1463E A divisor of 0 is detected in a divide operation.

Explanation: The expression contains a divide operation where the divisor was determined to be zero.

EQA1464E ***data item*** **is used as a receiver but it is not a data name.**

Explanation: The target of an assignment is not valid.

EQA1465E The TGT for a program is not available.

Explanation: The program may have been deleted or canceled.

EQA1466E ***data item*** **is not a valid subscript or index.**

Explanation: The subscript or index may be out of range or an ODO variable may not be initialized.

EQA1467E No subscript or index is allowed for ***data item***

Explanation: One or more subscripts or indexes were specified for a data item that was not defined as a table. The reference to the data item is not allowed.

EQA1468E Missing subscripts or indexes for ***data item***

Explanation: A data item defined as a table was referenced without specifying any subscripts or indexes. The reference is not allowed.

EQA1469E Incorrect number of subscripts or indexes for ***data item***

Explanation: A data item defined as a table was referenced with incorrect number of subscripts or indexes. The reference is not allowed.

EQA1470E Incorrect length specification for ***data item***

Explanation: The length of a data item is incorrect for the definition, usually due to a faulty ODO object.

EQA1471E Incorrect value for ODO variable ***data item***

Explanation: The ODO variable may not have been initialized, or the current value is out of range.

EQA1472E Invalid specification of reference modification.

Explanation: The specification of the reference modification is not consonant with the length field.

EQA1473E Invalid zero value for ***data item***

Explanation: The value of a data item is zero. A zero is invalid in the current context.

EQA1474E ***procedure name*** **was found where a data name was expected.**

Explanation: Invalid name is specified for a data item.

EQA1475E ***data item*** **is an invalid qualifier in a qualified reference.**

Explanation: A qualified reference is invalid. One or more qualifiers might be undefined or not in the same structure as the desired data item.

EQA1476E Too many qualifiers in a qualified reference.

Explanation: The qualified reference contains more than the legal number of qualifiers.

EQA1477E DATA DIVISION does not contain any entries.

Explanation: There is no data to display for a LIST * request because the DATA DIVISION does not contain any entries.

EQA1478E No status available for sort file ***sort file***

Explanation: Status was requested for a sort file. There is never a status available for a sort file.

EQA1479E Unable to locate any TGT.

Explanation: An attempt to locate any TGT failed. No COBOL program exists in TEST mode.

EQA1480E ***operand name*** **is an invalid operand for SET command.**

Explanation: The operands for a SET command are incorrect. At least one of the operands must be index name.

EQA1481E Too many digits for the exponent of floating point literal ***data item***

Explanation: The exponent specified for a floating-point literal contains too many digits.

EQA1482E *command name* **command is terminated due to an error in processing.**

Explanation: The command is terminated unsuccessfully because an error occurred during processing.

EQA1483E *reference* **could not be formatted for display.**

Explanation: The requested data item could not be displayed due to an error in locating or formatting the data item.

EQA1484E Resources (for example, heap storage) are not available for processing and the command is terminated unsuccessfully.

Explanation: The command could not be completed due to inadequate resources.

Programmer Response: Increase the partition size and restart Debug Tool.

EQA1485E The command is not supported because the CU is compiled with incorrect compile-time options.

Explanation: For COBOL, the CUs must be compiled with VS COBOL II Version 1 Release 4 and the compile-time TEST or FDUMP option, or IBM COBOL for VSE/ESA and the compile-time TEST option.

EQA1486E *variable name* **is presently not in accessible storage.**

Explanation: The variable may be CONTROLLED or AUTOMATIC and does not yet exist.

EQA1487S The number of dimensions for *variable name* **is** *number* **-- but** *number* **have been specified.**

Explanation: The wrong number of subscripts were specified with the variable reference.

EQA1488E The indices in *variable name* **are invalid. Use the DESCRIBE ATTRIBUTES command (without any indices specified) to see the valid indices.**

Explanation: The subscripts with the variable reference do not properly relate to the variable's characteristics.

EQA1489S *variable name* **is not a based variable but a locator has been supplied for it.**

Explanation: A pointer cannot be used unless the variable is BASED.

Programmer Response: Use additional qualification to get to the desired variable.

EQA1490S *variable name* **cannot be used as a locator variable.**

Explanation: Only variables whose data type is POINTER or OFFSET can be used to locator with other variables.

EQA1491S There is no variable named *character string***, and if it is meant to be a built-in function, the maximum number of arguments to the** *character string* **built-in function is** *number***, but** *number* **were specified.**

Explanation: A subscripted variable could not be found. Its name, however, is also that of a PL/I built-in function. If the built-in function was intended, the wrong number of arguments were present.

EQA1492S There is no variable named *character string***, and if it is meant to be a built-in function, the minimum number of arguments to the** *character string* **built-in function is** *number***, but** *number* **were specified.**

Explanation: A subscripted variable could not be found. Its name, however, is also that of a PL/I built-in function. If the built-in function was intended, more arguments must be present.

EQA1493E There is no variable named *character string***, and if it is meant to be a built-in function, remember built-in functions are allowed only in expressions.**

Explanation: A variable could not be found. Its name, however, is also that of a PL/I built-in function. If the built-in function was intended, it is not in the correct context. Note that in Debug Tool, pseudo-variables cannot be the target of assignments.

EQA1494S *variable name* **is an aggregate. It cannot be used as a locator reference.**

Explanation: The variable that is being as a locator is not the correct data type.

EQA1495S The name *variable name* is ambiguous and cannot be resolved.

Explanation: Names of structure elements can be ambiguous if not fully qualified. For example in DCL 1 A, 2 B, 3 Z POINTER, 2 C, 3 Z POINTER, the names Z and A.Z are ambiguous.

Programmer Response: Retry the command with enough qualification so that the name is unambiguous.

EQA1496S The name *variable name* refers to a structure, but structures are not supported within this context.

Explanation: Given DCL 1 A, 2 B FIXED, 2 C FLOAT, the name A refers to a structure.

Programmer Response: Break the command into commands for each of the basic elements of the structure, or use the DECLARE command with a BASED variable to define a variable overlaying the structure.

EQA1497S An aggregate cannot be used as an index into an array.

Explanation: Given DCL A(2) FIXED BIN(15) and DCL B(2) FIXED BIN(15), references to A(B), A(B+2), and so on are invalid.

Programmer Response: Use a scalar as the index.

EQA1498S Generation and recursion numbers must be positive.

Explanation: In %GENERATION(x,y) and %RECURSION(x,y), y must be positive.

EQA1499S Generation and recursion expressions cannot be aggregate expressions.

Explanation: In %GENERATION(x,y) and %RECURSION(x,y), y must be a scalar.

EQA1500S %RECURSION can be applied only to parameters and automatic variables.

Explanation: In %RECURSION(x,y), x must be a parameter or an automatic variable

EQA1501S %RECURSION *number* of *procedure name* does not exist. The present number of recursions of the block *block name* is *number*.

Explanation: In %RECURSION(x,y), y must be no greater than the number of recursions of the block where x is declared.

EQA1502S %Generation can be applied only to controlled variables.

Explanation: In %GENERATION(x,y), x must be controlled.

EQA1503S %Generation *number* of *variable name* does not exist. The present number of allocations of *variable name* is *number*.

Explanation: In %GENERATION(x,y), y must be no greater than the number of allocations of the variable x.

EQA1504S %Generation *number* of %RECURSION (*procedure name*, *number*) does not exist. The present number of allocations of %RECURSION (*procedure name*, *number*) is *number*.

Explanation: In %GENERATION(x,y), y must be no greater than the number of allocations of the variable x.

EQA1505S The variable *variable name* belongs to a FETCHed procedure and is a CONTROLLED variable that is not a parameter. This violates the rules of PL/I.

Explanation: PL/I does not allow FETCHed procedures to contain CONTROLLED variable types.

Programmer Response: Correct the program.

EQA1506S The variable *character string* cannot be used.

Explanation: The variable belongs to the class of variables, such as members of structures with REFER statements, which Debug Tool does not support.

EQA1507E The expression in the QUIT command must be a scalar that can be converted to an integer value.

Explanation: The expression in the QUIT command cannot be an array, a structure or other data aggregate, and if it is a scalar, it must have a type that can be converted to integer.

EQA1508E An internal error occurred in C run time during expression processing.

Explanation: This message applies to C. An internal error occurred in the C run time and the command is terminated.

EQA1509E The unary operator *operator name* requires a scalar operand.

Explanation: This message applies to the C unary operator ! (logical negation).

EQA1510E The unary operator ***operator name*** **requires a modifiable lvalue for its operand.**

Explanation: This message applies to the C unary operators ++ and --.

EQA1511E The unary operator ***operator name*** **requires an integer operand.**

Explanation: This message applies to the C unary operator (bitwise complement).

EQA1512E The unary operator ***operator*** **requires an operand that is either arithmetic or a pointer to a type with defined size.**

Explanation: This message applies to the C unary operators + and -. These operators cannot be applied to pointers to void-function designators, or pointers to functions.

EQA1513E The unary operator ***operator*** **requires an arithmetic operand.**

Explanation: This message applies to the C unary operator + and -.

EQA1514E Too many arguments specified in function call.

Explanation: This message applies to C function calls.

EQA1515E Too few arguments specified in function call.

Explanation: This message applies to C function calls.

EQA1516E The logical operator ***operator*** **requires a scalar operand.**

Explanation: This message applies to the C binary operators && (logical and) and || (logical or).

EQA1517E The operand of the type cast operator must be scalar.

Explanation: This message applies to the C type casts.

EQA1518E The named type of the type cast operator must not be an expression.

Explanation: This message applies to the C type casts.

EQA1519E A real type cannot be cast to a pointer type.

Explanation: This message applies to C type casts. In the example 'float f;', the type cast '(float *) f' is invalid.

EQA1520E A pointer type cannot be cast to a real type.

Explanation: Invalid operand for the type cast operator.

EQA1521E The operand in a typecast must be scalar.

Explanation: This message applies to C type casts.

EQA1522E Argument ***argument*** **in function call** ***function*** **has an invalid type.**

Explanation: This message applies to C function calls.

EQA1523E Invalid type for function call.

Explanation: This message applies to C function calls.

EQA1524E The first operand of the subscript operator must be a pointer to a type with defined size.

Explanation: This message applies to the C subscript operator. The subscript operator cannot be applied to pointers to void, function designators or pointers to functions.

EQA1525E Subscripts must have integer type.

Explanation: This message applies to the C subscript operator.

EQA1526E The first operand of the sizeof operator must not be a function designator, a typedef, a bitfield or a void type.

Explanation: This message applies to the C unary operator sizeof.

EQA1527E The second operand of the ***operator*** **operator must be a member of the structure or union specified by the first operand.**

Explanation: This message applies to the C operators (select member) and -> (point at member).

EQA1528E The first operand of the *operator* operator must have type pointer to struct or pointer to union.

Explanation: This message applies to the C operator -> (point at member).

EQA1529E The operand of the *operator* operator must be an array, a function designator, or a pointer to a type other than void.

Explanation: This message applies to the C indirection operator.

EQA1530E The first operand of the *operator* operator must have type struct or union.

Explanation: This message applies to the C subscript operator (select member).

EQA1531E The relational operator *operator* requires comparable data types.

Explanation: This message applies to the C relational operators. For example, <, >, <=, >=, and ==.

EQA1532E The subtraction operator requires that both operands have arithmetic type or that the left operand is a pointer to a type with defined size and the right operand has the same pointer type or an integral type.

Explanation: This message applies to the C binary operator -. The difference between two pointers to void or two pointers to functions is undefined because sizeof is not defined for void types and function designators.

EQA1533E Assignment contains incompatible types.

Explanation: This message applies to C assignments, for example, +=, -=, and *=.

EQA1534E The TEST expression in the switch operator must have integer type.

Explanation: This applies to the test expression in a C switch command.

EQA1535E The addition operator requires that both operands have arithmetic or that one operand has integer type and the other operand is a pointer to a type with defined size.

Explanation: This message applies to the C binary operator +.

EQA1536E The operand of the address operator must be a function designator or an lvalue that is not a bitfield.

Explanation: This message applies to the C unary operator & (address).

EQA1537E Invalid constant for the C language.

Explanation: This message applies to C constants.

EQA1538E Argument *argument* in function call *function* is incompatible with the function definition. Since Warning is on, the function call is not made.

Explanation: This message applies to C function calls. The argument must have a type that would be valid in an assignment to the parameter.

EQA1539E The binary operator *operator* requires integer operands.

Explanation: This message applies to the C binary operator % (remainder), << (bitwise left shift), >> (bitwise right shift), & (bitwise and), ??¬' (bitwise exclusive or), |(bitwise inclusive or), and the corresponding assignment operators (for example, %=, and <<=).

EQA1540E The binary operator *operator* requires a modifiable lvalue for its first operand.

Explanation: This message applies to the C binary assignment operators.

EQA1541E The binary operator *operator* requires arithmetic operands.

Explanation: This message applies to the C binary operators * and /.

EQA1542E Source in assignment to an enum is not a member of the enum. Since Warning is on, the operation is not performed.

Explanation: This message applies to C. You attempted to assign a value to enum, but the value is not legitimate for that enum.

EQA1543E Invalid value for the shift operator *operator*. Since Warning is on, the operation will not be performed.

Explanation: This message applies to the C binary operators << (bitwise left shift) and >> (bitwise right shift). Shift values must be nonnegative and less than 33. These tests are made only when WARNING is on.

EQA1544E Array subscript is negative. Since Warning is on, the operation is not performed.

Explanation: This message applies to the C subscripts.

EQA1545E Array subscript exceeds maximum declared value. Since Warning is on, the operation is not performed.

Explanation: This message applies to the C subscripts.

EQA1546E ZeroDivide would have occurred in performing a division operator. Since Warning is on, the operation is not performed.

Explanation: Divide by zero is detected by C run time.

EQA1547E *variable* **is presently not in accessible storage.**

Explanation: This message applies to C. Use the LIST NAMES command to list all known variables.

EQA1548E There is no variable named *variable*

Explanation: This message applies to C. Use the LIST NAMES command to list all known variables.

EQA1549E The function call *function* **is not performed because the function linkages do not match.**

Explanation: This message applies to C function calls and can occur,for example, when a function's linkage is specified as CEE, but the function was compiled with linkage OS.

EQA1550E There is no typedef identifier named *name*

Explanation: This message applies to C. The message is issued, for example, in response to the command 'DESCRIBE ATTRIBUTE typedef x', if x is not a typedef identifier.

EQA1551E *name* **is the name of a member of an enum type.**

Explanation: This message applies to C.

EQA1552E The name *name* **is invalid.**

Explanation: This message applies to C declarations.

EQA1553E Linkage type for function call *function* **is unknown.**

Explanation: This message applies to C function calls.

EQA1554E Function call *function* **has linkage type PL/I, which is not supported.**

Explanation: This message applies to C function calls.

EQA1555E Function call *function* **has linkage type FORTRAN which is not supported.**

Explanation: This message applies to C function calls.

EQA1556E *name* **is a tag name. This cannot be listed since it has no storage associated with it.**

Explanation: This message applies to C tag names.

EQA1557E *name* **is not an lvalue. This cannot be listed since it has no storage associated with it.**

Explanation: This message applies to C names.

EQA1558E *name* **has storage class void, not permitted on the LIST command.**

Explanation: This message applies to C. In the example 'void' funcname (...)', the command 'LIST TITLED (funcname())' is invalid.

EQA1559E The second operand of the %RECURSION operator must be arithmetic.

Explanation: This message applies to C. In %RECURSION(x,y), the second expression, y, must have arithmetic type.

EQA1560E The second operand of the %RECURSION operator must be positive.

Explanation: This message applies to C. In %RECURSION(x,y), the second expression, y, must be positive.

EQA1561E The first operand of the %RECURSION operator must be a parameter or an automatic variable.

Explanation: This message applies to C. In %RECURSION(x,y), the first expression, x, must be a parameter or an automatic variable.

EQA1562E The first operand of the %INSTANCE operator must be a parameter or an automatic variable.

Explanation: This message applies to C. In %INSTANCE(x,y), the first expression, x, must be a parameter or an automatic variable.

EQA1563E Generation specified for %RECURSION is too large.

Explanation: This message applies to C. In %RECURSION(x,y), the recursion number, y, exceeds the number of generations of x that are currently active.

EQA1564E The identifier *identifier* has been replaced.

Explanation: This message applies to C declarations.

EQA1565E The declaration is too large

Explanation: This message applies to C declarations.

EQA1566E An attempt to modify a constant was made. Since Warning is on, the operation is not performed.

Explanation: This message applies to C.

EQA1567E An attempt to take the address of a variable with register storage was made. Since Warning is on, the operation is not performed.

Explanation: This message applies to C.

EQA1568E Type of expression to %DUMP must be a literal string.

Explanation: This message applies to CALL %DUMP for C.

EQA1569E Octal constant is too long.

Explanation: This message applies to C constants.

EQA1570E Octal constant is too big.

Explanation: This message applies to C constants.

EQA1571E Hex constant is too long.

Explanation: This message applies to C constants.

EQA1572E Decimal constant is too long.

Explanation: This message applies to C constants.

EQA1573E Decimal constant is too big.

Explanation: This message applies to C constants.

EQA1574E Float constant is too long.

Explanation: This message applies to C constants.

EQA1575E Float constant is too big.

Explanation: This message applies to C constants.

EQA1576E The environment is not yet fully initialized.

Explanation: You can STEP and try the command again.

EQA1577E Size of the aggregate is too large

Explanation: This message applies to PL/I constants.

EQA1578E Only "=" and "¬=" are allowed as operators in comparisons involving program control data.

Explanation: Other relationships between program control data are not defined.

Programmer Response: Check to see if a variable was misspelled.

EQA1579E Program control data may be compared only with program control data of the same type.

Explanation: ENTRY vs ENTRY, LABEL vs LABEL, etc. are okay. LABEL vs ENTRY is not.

EQA1580E Area variables cannot be compared.

Explanation: Equivalency between AREA variables is not defined.

EQA1581E Aggregates are not allowed in conditional expressions such as the expressions in IF ... THEN, WHILE (...), UNTIL (...), and WHEN (...) clauses.

Explanation: This is not supported.

Programmer Response: Check to see if the variable name was misspelled. If this was not the problem, you must find other logic to perform the task.

EQA1582E Only "=" and "¬=" are allowed as operators in comparisons involving complex numbers.

Explanation: Equal and not equal are defined for complex variables, but you have attempted to relate them in some other way.

EQA1583E Strings with the GRAPHIC attribute may be concatenated only with other strings with the GRAPHIC attribute.

Explanation: You are not allowed to concatenate GRAPHIC (DBCS) strings to anything other than other GRAPHIC (DBCS) strings.

EQA1584E Strings with the GRAPHIC attribute may be compared only with other strings with the GRAPHIC attribute.

Explanation: Equivalency between the GRAPHIC data type and other data types has not been defined.

EQA1585E Only numeric data, character strings, and bit strings may be the source for conversion to character data.

Explanation: You are trying to convert something to a character format when such a relationship has not been defined.

EQA1586E Only numeric data, character strings, and bit strings may be the source for conversion to bit data.

Explanation: You are trying to convert something to a bit format when such a relationship has not been defined.

EQA1587E Only numeric data, character strings, bit strings, and pointers may be the source for conversion to numeric data.

Explanation: You are trying to convert something to a numeric format when such a relationship has not been defined.

EQA1588E Aggregates are not allowed in control expressions.

Explanation: This message applies to PL/I constants.

EQA1589W CONVERSION would have occurred in performing a CHARACTER to BIT conversion, but since WARNING is on, the conversion will not be performed.

Explanation: The specified conversion probably contained characters that were something other than '0' or '1'. Since the conversion to BIT could therefore not be done, this message is displayed rather than raising the CONVERSION condition.

EQA1590W Varying string ***variable name*** **has a length that is greater than its declared maximum. It will not be used in expressions until it is fixed.**

Explanation: The variable named has been declared as VARYING with length n, but its current length is greater than n. The variable may be uninitialized or may have been written over.

EQA1591W Varying string ***variable name*** **has a negative string length. It will not be used in expressions until it is fixed.**

Explanation: The variable named has been declared as VARYING with length n, but its current length is less than 0. The variable may be uninitialized or it may have been written over.

EQA1592W Fixed decimal variable ***variable name*** **contains bad data. Since WARNING is on, the operation will not be performed.**

Explanation: A variable contains bad decimal data if its usage would cause a data exception to occur (that is, its numeric digits are not 0-9 or its sign indicator is invalid), or it has even precision but its leftmost digit is nonzero. LIST STORAGE can be used to find the contents of the variable, and an assignment statement can be used to correct them.

EQA1593W The size of AREA variable ***variable name*** **is less than zero. Since WARNING is on, the operation will not be performed.**

Explanation: Negative sizes are not understood and, therefore, are not processed.

EQA1594W The size of AREA variable ***variable name*** **exceeds its declared maximum size. Since WARNING is on, the operation will not be performed.**

Explanation: Performing the operation would alter storage that is outside of the AREA. Such an operation is not within PL/I, so will be avoided.

EQA1595W Fixed binary variable ***variable name*** **contains more significant digits than its precision allows. Since WARNING is on, the operation will not be performed.**

Explanation: For example, a FIXED BIN(5,0) variable can have only 5 significant digits thus limiting its valid range of values to -32 through 31 inclusive.

EQA1596E The subscripted variable *variable name* **was not found. The name matches a built-in function, but the parameters are wrong.**

Explanation: This message applies to PL/I constants.

EQA1597E AREA condition would have been raised

Explanation: This message applies to PL/I constants.

EQA1598E The bounds and dimensions of all arrays in an expression must be identical.

Explanation: Array elements of an expression (such as A + B or A = B) must all have the same number of dimensions and the same lower and upper bounds for each dimension.

EQA1599E You cannot assign an array to a scalar.

Explanation: The PL/I language does not define how a scalar would represent an array; the assignment is rejected as an error.

EQA1600E Aggregate used in wrong context.

Explanation: This message applies to PLI constants.

EQA1601E The second expression in the *built-in function name* **built-in function must be greater than or equal to 1 and less than or equal to the number of dimensions of the first expression.**

Explanation: The second expression of the named built-in function is dependent upon the dimensions of the array (the first built-in function argument).

Programmer Response: Correct the relationship between the first and second arguments.

EQA1602E The second expression in the *built-in function name* **built-in function must not be an aggregate.**

Explanation: Debug Tool does not support aggregates in this context.

EQA1603E The first argument in the *built-in function name* **built-in function must be an array expression.**

Explanation: The named built-in function expects an array to be the first argument.

EQA1604E Argument number *number* **in the** *built-in function name* **built-in function must be a variable.**

Explanation: You used something other than a variable name (for example, a constant) in your invocation of the named built-in function.

EQA1605E STRING(*variable name* **) is invalid because the STRING built-in function may be used only with bit, character and picture variables.**

Explanation: You must use a variable of the correct data type with the STRING built-in function.

EQA1606E POINTER(*variable name* **,...) is invalid because the first argument to the POINTER built-in function must be an offset variable.**

Explanation: The first argument to POINTER was determined to be something other than an OFFSET data type.

EQA1607E POINTER(..., *variable name* **) is invalid because the second argument to the POINTER built-in function must be an area variable.**

Explanation: The second argument to POINTER was determined to be something other than an AREA data type.

EQA1608E OFFSET(*variable name* **,...) is invalid because the first argument to the OFFSET built-in function must be a pointer variable.**

Explanation: The first argument to OFFSET was determined to be something other than a POINTER data type.

EQA1609E OFFSET(..., *variable name* **) is invalid because the second argument to the OFFSET built-in function must be an area variable.**

Explanation: The second argument to OFFSET was determined to be something other than an AREA data type.

EQA1610E *built-in function name* **(** *variable name* **) is invalid because the argument to the** *built-in function name* **built-in function must be a file reference.**

Explanation: The name built-in function requires the name of a FILE to operate. Some other data type was used as the argument.

EQA1611E COUNT(*variable name* **) must refer to an open STREAM file.**

Explanation: You must name an open STREAM file in the COUNT built-in function.

EQA1612E LINENO(*variable name* **) must refer to an open PRINT file.**

Explanation: You must name an open PRINT file in the LINENO built-in function.

EQA1613E SAMEKEY(*variable name* **) must refer to a RECORD file.**

Explanation: You must name a RECORD file in the SAMEKEY built-in function. This requirement is tested for all file constants, but is tested for file variables only if the file variable is associated with an open file.

EQA1614E The argument in the *built-in function name* **built-in function must be a variable.**

Explanation: The built-in function is expecting a variable but a constant or some other invalid item appeared as one of the arguments.

EQA1615E Argument to POINTER is an aggregate when pointer is being used as a locator.

Explanation: This message applies to PL/I constants.

EQA1616E The result of invoking the GRAPHIC built-in function must not require more than 16383 DBCS characters.

Explanation: GRAPHIC(x,y) is illegal if y > 16383, and GRAPHIC(x) is illegal if length(CHAR(X)) > 16383.

EQA1617W The first argument to the *built-in function name* **built-in function is negative, but since WARNING is on, the evaluation will not be performed.**

Explanation: The specified built-in function would fail if a negative argument was passed. Use of the built-in function will be avoided.

EQA1618W The second argument to the *built-in function name* **built-in function is negative, but since WARNING is on, the evaluation will not be performed.**

Explanation: The specified built-in function would fail if a negative argument was passed. Use of the built-in function will be avoided.

EQA1619W The third argument to the *built-in function name* **built-in function is negative, but since WARNING is on, the evaluation will not be performed.**

Explanation: The specified built-in function would fail if a negative argument was passed. Use of the built-in function will be avoided.

EQA1620E If the CHAR built-in function is invoked with only one argument, that argument must not have the GRAPHIC attribute with length 16383.

Explanation: CHAR(x) is illegal if x is GRAPHIC with length 16383 since the resultant string would require 32768 characters.

EQA1621E *built-in function* **(** *variable name* **) is not defined since** *variable name* **is not connected.**

Explanation: This applies to the PL/I CURRENTSTORAGE and STORAGE built-in functions.

EQA1622E *built-in function* **(** *variable name* **) is not defined since** *variable name* **is an unaligned fixed-length bit string.**

Explanation: This applies to the PL/I CURRENTSTORAGE and STORAGE built-in functions.

EQA1623E *built-in function* **(x) is undefined if ABS(x) > 1.**

Explanation: This applies to the PL/I ASIN and ACOS built-in functions.

EQA1624E ATANH(z) is undefined if z is COMPLEX and z = +1 or z = -1.

Explanation: This applies to the PL/I ATANH built-in function.

EQA1625E ATAN(z) is undefined if z is COMPLEX and z = +1i or z = -1i.

Explanation: This applies to the PL/I ATAN built-in function.

EQA1626E Built-in function not defined since the argument is real and less than or equal to zero

Explanation: This message applies to PL/I constants.

EQA1627E SQRT(x) is undefined if x is REAL and x < 0.

Explanation: This applies to the PL/I SQRT built-in function.

EQA1628E *built-in function* (x,y) is undefined if x or y is COMPLEX.

Explanation: This applies to the PL/I ATAN and ATAND built-in functions.

EQA1629E Built-in function(X,Y) is undefined if X=0 and Y=0

Explanation: This applies to PL/I constants.

EQA1630E The argument in *built-in function* is too large.

Explanation: This applies to the PL/I trigonometric built-in functions.

For short floating-point arguments, the limits are:

COS and SIN	ABS(X) <= (2**18)*pi
TAN	ABS(X) <= (2**18)*pi if x is real and ABS(REAL(X)) <= (2**17)*pi if x is complex
TANH	ABS(IMAG(X)) <= (2**17)*pi if x is complex
COSH, EXP and SINH	ABS(IMAG(X)) <= (2**18)*pi if x is complex
COSD, SIND and TAND	ABS(X) <= (2**18)*180

For long floating-point arguments, the limits are:

COS and SIN	ABS(X) <= (2**50)*pi
TAN	ABS(X) <= (2**50)*pi if x is real and ABS(REAL(X)) <= (2**49)*pi if x is complex
TANH	ABS(IMAG(X)) <= (2**49)*pi if x is complex
COSH, EXP and SINH	ABS(IMAG(X)) <= (2**50)*pi if x is complex
COSD, SIND and TAND	ABS(X) <= (2**50)*180

For extended floating-point arguments, the limits are:

COS and SIN	ABS(X) <= (2**106)*pi
TAN	ABS(X) <= (2**106)*pi if x is real and ABS(REAL(X)) <= (2**105)*pi if x is complex
TANH	ABS(IMAG(X)) <= (2**105)*pi if x is complex
COSH, EXP and SINH	ABS(IMAG(X)) <= (2**106)*pi if x is complex
COSD, SIND and TAND	ABS(X) <= (2**106)*180

EQA1631E The subject of the SUBSTR pseudovariable (*character string*) is not a string.

Explanation: You are trying to get a substring from something other than a string.

EQA1632E Argument to pseudovariable must be complex numeric

Explanation: This message applies to PL/I constants.

EQA1633E The first argument to a pseudovariable must refer to a variable, not an expression or a pseudovariable.

Explanation: The arguments that accompany a pseudovariable are incorrect.

EQA1634E The length of the bit string that would be returned by UNSPEC is greater than the maximum for a bit variable. Processing of the expression will stop.

Explanation: This will occur in UNSPEC(A) where A is CHARACTER(n) and n > 4095, where A is CHARACTER(n) VARYING and n > 4093, where A is AREA(n) and n > 4080, etc.

EQA1635E Maximum number of arguments to PLIDUMP subroutine is two

Explanation: This message applies to PL/I constants.

EQA1636E Invalid argument in CALL %DUMP

Explanation: This message applies to PL/I constants.

EQA1637E PL/I cannot process the expression *expression name*.

Explanation: This applies to PL/I constants.

EQA1638E Argument *argument number* to the MPSTR built-in function must not have the GRAPHIC attribute.

Explanation: GRAPHIC (DBCS) strings are prohibited as arguments to the MPSTR built-in function.

EQA1639E ALLOCATION(*variable name*) is invalid because the ALLOCATION built-in function can be used only with controlled variables.

Explanation: You must name a variable that is ALLOCATEable.

Programmer Response: The variable by that name cannot be a controlled variable within the current context. If the variable exists somewhere else (and is a

controlled variable), you should use qualification with the variable name.

EQA1640E ***variable name*** **is an aggregate and hence is invalid as an argument to the POINTER built-in function when that built-in function is used as a locator.**

Explanation: The argument to the POINTER built-in function is invalid. The argument to the POINTER built-in function should be an OFFSET data type for the first argument, or an AREA data type for the second argument.

EQA1641E Structures are not supported within this context.

Explanation: Given DCL 1 A, 2 B FIXED, 2 C FLOAT, the name A refers to a structure.

Programmer Response: Break the command into commands for each of the basic elements of the structure, or use the DECLARE command with a BASED variable to define a variable overlaying the structure.

EQA1642E The first argument to the ***built-in function name*** **built-in function must have POINTER type.**

Explanation: This applies to the POINTERADD built-in function. The first argument must have pointer type, and it must be possible to convert the other to FIXED BIN(31,0).

EQA1643E The argument in the ***built-in function name*** **built-in function must have data type:** ***data type*****.**

Explanation: This message applies to various built-in functions. By built-in function, the datatypes required are:

ENTRYADDR	ENTRY
BINARYVALUE	POINTER
BINVALUE	POINTER

EQA1644W STRINGRANGE is disabled and the SUBSTR arguments are such that STRINGRANGE ought to be raised. Debug Tool will revise the SUBSTR reference as if STRINGRANGE were enabled.

Explanation: See the Language Reference Manual (LRM) built-in function chapter for the description of when STRINGRANGE is raised. See the LRM condition chapter for the values of the revised SUBSTR reference.

EQA1645E The subject of the ***pseudovariable name*** **pseudovariable must have data type:** ***data type*****.**

Explanation: This message applies to various pseudovariables. By pseudovariable, the datatypes required are:

ENTRYADDR	ENTRY VARIABLE

EQA1646E ***built-in function*** **(z) is undefined if z is COMPLEX.**

Explanation: This applies to the PL/I ACOS, ASIN, ATAND, COSD, ERF, ERFC, LOG2, LOG10, SIND and TAND built-in functions.

Explanation: This applies to PL/I constants.

EQA1649E Error: see Command Log.

Explanation: An error has occurred during expression evaluation. See the Debug Tool Command Log for more detailed information.

EQA1650E The range of statements ***statement_id*** **-** ***statement_id*** **is invalid because the two statements belong to different blocks.**

Explanation: AT stmt1-stmt2 is valid only if stmt1 and stmt2 are in the same block.

EQA1651W The ***breakpoint-id*** **breakpoint has not been established.**

Explanation: You just issued a CLEAR/LIST command against a breakpoint that does not exist.

Programmer Response: Verify that you referred to the breakpoint using the same syntax that was used to establish it. Perhaps a CLEAR command occurred since the command that established the breakpoint.

EQA1652E Since the program for the statement ***statement-number*** **does not have hooks at statements, AT commands are rejected for all statements in the program.**

Explanation: A compile unit must have been compiled with TEST(STMT) or TEST(ALL) for hooks to be present at statements.

EQA1653E A file name is invalid in this context.

Explanation: A command (for example, AT ENTRY) specified a C file name where a function or compound statement was expected.

EQA1654E Since the cu *cu_name* does not have hooks at block entries and exits, all AT ENTRY and AT EXIT commands will be rejected for the cu.

Explanation: A compile unit must have been compiled with TEST(BLOCK), TEST(PATH) or TEST(ALL) for hooks to be present at block exits and block entries.

EQA1655E Since the program for the label *label-name* does not have hooks at labels, AT commands are rejected for all labels in the program.

Explanation: A compilation unit must have been compiled with TEST(PATH) or TEST(ALL) for hooks to be present at labels.

EQA1656E *statement_id* contains a value that is invalid in this context.

Explanation: %STATEMENT and %LINE are invalid in AT commands at block entry and block exit, and in AT and LIST STATEMENT commands at locations that are outside of the program.

EQA1657W There are no *breakpoint-class* breakpoints set.

Explanation: The command CLEAR/LIST AT was entered but there are no AT breakpoints presently set, or the command CLEAR/LIST AT class was entered but there are no AT breakpoints presently set in that class.

EQA1658W There are no enabled *breakpoint-class* breakpoints set.

Explanation: The command CLEAR/LIST AT was entered but there are no enabled AT breakpoints presently set in the requested class of breakpoints.

EQA1659W There are no disabled *breakpoint-class* breakpoints set.

Explanation: The command CLEAR/LIST AT was entered but there are no disabled AT breakpoints presently set in the requested class of breakpoints.

EQA1660W The *breakpoint-id* breakpoint is not enabled.

Explanation: You issued a specific LIST AT ENABLED command against a breakpoint that is not enabled.

EQA1661W The *breakpoint-id* breakpoint is not disabled.

Explanation: You issued a specific LIST AT DISABLED command against a breakpoint that is not disabled.

EQA1662W The *breakpoint-id* breakpoint cannot be triggered because it is disabled.

Explanation: You cannot TRIGGER a disabled breakpoint.

EQA1663W There are no breakpoints set.

Explanation: No breakpoints are currently set.

EQA1664W There are no disabled breakpoints set.

Explanation: No disabled breakpoints are currently set.

EQA1665W There are no enabled breakpoints set.

Explanation: No enabled breakpoints are currently set.

EQA1666W The *breakpoint-id* breakpoint is already enabled.

Explanation: You cannot ENABLE an enabled breakpoint.

EQA1667W The *breakpoint-id* breakpoint is already disabled.

Explanation: You cannot DISABLE a disabled breakpoint.

EQA1668W The attempt to set this breakpoint has failed.

Explanation: For some reason, when Debug Tool tried to set this breakpoint, an error occurred. This breakpoint cannot be set.

EQA1669W The FROM or EVERY value in a breakpoint command must not be greater than the specified TO value.

Explanation: In an every_clause specified with a breakpoint command, if the TO value was specified, the FROM or EVERY value must be less than or equal to the TO value.

EQA1670W GO/RUN BYPASS is ignored. It is valid only when entered for an AT CALL, AT GLOBAL CALL, or AT OCCURRENCE.

Explanation: GO/RUN BYPASS is valid only when Debug Tool is entered for an AT CALL, AT GLOBAL CALL, or AT OCCURRENCE breakpoint.

EQA1671W AT OCCURRENCE breakpoint or TRIGGER of condition *condition-name* cannot have a reference specified. This command not processed.

Explanation: The following AT OCCURRENCE conditions must have a qualifying reference: CONDITION, ENDFILE, KEY, NAME, PENDING, RECORD, TRANSMIT and UNDEFINEDFILE. This would also apply to the corresponding TRIGGER commands.

EQA1672W AT OCCURRENCE breakpoint or TRIGGER of condition *condition-name* must have a valid reference specified. This command not processed.

Explanation: The following AT OCCURRENCE conditions must have a valid qualifying reference: CONDITION, ENDFILE, KEY, NAME, PENDING, RECORD, TRANSMIT and UNDEFINEDFILE. This would also apply to the corresponding TRIGGER commands.

EQA1673W An attempt to automatically restore an AT *breakpoint type* breakpoint failed.

Explanation: Debug Tool was attempting to restore a breakpoint that had been set in the previous process and has failed in that attempt. There are two reasons this could have happened. If the Compile Unit (CU) has been changed (that is, modified and recompiled/linked) between one process and the next and a breakpoint had been established for a statement or variable that no longer exists due to the change, when Debug Tool attempts to reestablish that breakpoint, it will fail with this message.

EQA1674W An attempt to automatically disable an AT *breakpoint type* breakpoint failed.

Explanation: Debug Tool was attempting to disable a breakpoint for a CU that has been deleted from storage (or deactivated), and failed in that attempt.

EQA1675E *variable name* is not a LABEL variable or constant. No GOTO commands can be issued against it.

Explanation: You are trying to GOTO a variable name that cannot be associated with a label in the program.

EQA1676S *label name* is a label variable that is uninitialized or that has been zeroed out. It cannot be displayed and should not be used except as the target of an assignment.

Explanation: You are trying to make use of a LABEL variable, but the control block representing that variable contains improper information (for example, an address that is zero).

EQA1677S *file name* is a file variable that is uninitialized or that has been zeroed out. It cannot be displayed and should not be used except as the target of an assignment.

Explanation: You are trying to make use of a FILE variable, but the control block representing that variable contains improper information (for example, an address that is zero).

EQA1678E The program *CU-name* has a short statement number table, and therefore no statement numbers in the program can be located.

Explanation: A command requires determining which statement was associated with a particular storage address. A statement table could not be located to relate storage to statement identifications.

Programmer Response: Check to see if the program had been compiled using *release name*. If so, was the statement table suppressed?

EQA1679E *variable name* is not a controlled variable. An ALLOCATE breakpoint cannot be established for it.

Explanation: You cannot establish an AT ALLOCATE breakpoint for a variable that cannot be allocated.

EQA1680E *variable name* is a controlled parameter. An ALLOCATE breakpoint can be established for it only when the block in which it is declared is active.

Explanation: Debug Tool cannot, at this time, correlate a block to the named variable. As a result, a breakpoint cannot be established.

Programmer Response: Establish the breakpoint via an AT ENTRY ... AT ALLOCATE ...

EQA1681E *variable name* is not a FILE variable or constant.

Explanation: ON/SIGNAL file-condition (variable) is invalid because the variable is not a PL/I FILE variable.

EQA1682E *variable name* **is not a CONDITION variable.**

Explanation: ON/SIGNAL CONDITION (variable) is invalid because the variable is not a PL/I CONDITION variable.

EQA1700E The session procedure, *procedure name*, **is either undefined or is hidden within a larger, containing procedure.**

Explanation: This is issued in response to a CALL, CLEAR, or QUERY command when the target session procedure cannot be located. It cannot be located for one of two reasons: it was not defined, or it is imbedded with another session procedure.

EQA1701E The maximum number of arguments to the %DUMP built-in subroutine is 2, but *number* **were specified.**

Explanation: %DUMP does not accept more than two parameters.

EQA1702E Invalid argument in CALL %DUMP.

Explanation: In PL/I, the %DUMP arguments must be scalar data that can be converted to character. In C, the %DUMP arguments must be pointers to character or arrays of character.

EQA1703E No arguments can be passed to a session procedure.

Explanation: Parameters are not supported with the CALL *procedure* command.

EQA1704E Invalid or incompatible dump options or suboptions

Explanation: This message is from the feedback code of LE/VSE CEE5DMP call.

EQA1705E Dump argument exceeds the maximum length allowed.

Explanation: The dump option allows a maximum of 255 characters. The dump title allows a maximum of 80 characters.

EQA1706E *pgmname* **must be loaded before calling the program.**

Explanation: The CALL command was terminated unsuccessfully.

EQA1720E There is no declaration for *variable name*.

Explanation: A command (for example, CLEAR VARIABLES) requires the use of a variable, but the specified variable was not declared (or was previously cleared).

Programmer Response: For a list of session variables that can be referenced in the current programming language, use the LIST NAMES TEST command.

EQA1721E The size of the variable is too large.

Explanation: A variable can require no more than 2**24 - 1 bytes in a non-XA machine and no more than 2**31 - 1 bytes in an XA machine.

EQA1722E Error in declaration; invalid attribute *variable name*.

Explanation: A session variable is declared with invalid or unsupported attribute.

EQA1723E There are no session variables defined.

Explanation: The CLEAR VARIABLES command is entered but there is no declaration for session variables.

EQA1724E There is no *tag type* **tag named** *tag name*.

Explanation: This message applies to C. It is issued, for example, after DESCRIBE ATTRIBUTES enum x if x is not an enum tag.

EQA1725E *tag type tag name* **is already defined.**

Explanation: This message applies to C.

A tagged enum, struct, or union type cannot be redefined, unless all variables and type definitions referring to that type and then the type itself are first cleared. For example, given

```
enum colors {red,yellow,blue} primary, * ptrPrimary;
```

enum colors cannot be redefined unless primary, ptrPrimary, and then enum colors are first cleared.

EQA1726E *tag type tag name* **cannot be cleared while one or more declarations refer to that type.**

Explanation: This message applies to C.

A CLEAR DECLARE of a tagged enum, struct, or union type is invalid while one or more declarations refer to that type. For example, given

```
enum colors {red,yellow,blue} primary, * ptrPrimary;
```

CLEAR DECLARE enum colors is invalid until CLEAR DECLARE (primary, ptrPrimary) is issued.

EQA1727E ***enum member name*** **is the name of a declared variable. It cannot be used as the name of a member of an enum type.**

Explanation: This message applies to C.

For example, given

```
int blue;
```

The use of the name blue in the following declaration is invalid:

```
enum teamColors {blue,gold};
```

EQA1728E The ***tag type tag name*** **is recursive: it contains itself as a member.**

Explanation: This message applies to C.

A struct or union type must not contain itself as a member. For example, the following declaration is invalid:

```
struct record {
int member;
struct record next;
}
```

EQA1729E An error occurred during declaration processing.

Explanation: Unable to process the declaration. The command is terminated unsuccessfully.

EQA1750E An error occurred during expression evaluation.

Explanation: Unable to evaluate the expression. The command is terminated unsuccessfully.

EQA1751E Program ***pgmname*** **not found.**

Explanation: A bad program name is specified in a CALL command and processing is terminated unsuccessfully.

EQA1752S Comparison in ***command-name*** **command was invalid. The command was ignored.**

Explanation: This message applies to COBOL.

The operands to be compared are of incompatible types.

EQA1753S The nesting of "switch" command exceeded the maximum.

Explanation: This message applies to C.

There are too many nested levels of switch commands.

EQA1754S An error occurred in "switch" command processing. The command is terminated.

Explanation: This message applies to C.

The switch command is terminated because an error occurred during processing.

EQA1755S Comparison with the ***keyword-name*** **keyword in** ***command-name*** **command was invalid. The command was ignored.**

Explanation: This message applies to COBOL.

The operands to be compared are incompatible. For example, the following comparison is invalid:

```
EVALUATE TRUE
When 6     List ('invalid');
when other List ('other');
END-EVALUATE
```

EQA1766E The target of the GOTO command is in an inactive block.

Explanation: You are trying to GOTO a block that is not active. If it is inactive it doesn't have a register save area, base registers, and so on—all of the mechanics established that would allow the procedure to run.

EQA1767S No offset was found for label "*label*".

Explanation: No offset associated with the label was found; the code associated with the label might have been removed by optimization.

EQA1768S The label "*label*" is not known.

Explanation: The label is not known.

EQA1769S The label "*label*" is ambiguous—multiple labels of this name exist.

Explanation: The label is ambiguous—multiple labels of this name exist.

EQA1770S The GOTO is not permitted, perhaps because of optimization.

Explanation: The GOTO command is not recommended. For COBOL, this might be due to optimization, or because register contents other than the code base cannot be guaranteed for the target.

EQA1771S The GOTO is not permitted due to language rules.

Explanation: The GOTO command is not recommended. For COBOL, this might be due to optimization, or because register contents other than the code base cannot be guaranteed for the target.

EQA1772S The GOTO was not successful.

Explanation: There are various reasons why a GOTO command can be unsuccessful; this message covers all the other situations not covered by the other message in the GOTO LABEL messages group.

EQA1773E GOTO is invalid when the target statement number is in a C function.

Explanation: The target statement number in a GOTO command must belong to an active procedure.

EQA1786W There are no entries in the HISTORY table.

Explanation: Debug Tool has not yet encountered any of the situations that cause entries to be put into the HISTORY table; so it is empty.

EQA1787W There are no STATEMENT entries in the HISTORY table.

Explanation: LIST STATEMENTS or LIST LAST n STATEMENTS was entered, but there are no STATEMENT entries in the HISTORY table. Debug Tool was not invoked for any STATEMENT hooks.

EQA1788W There are no PATH entries in the HISTORY table.

Explanation: LIST PATH or LIST LAST n PATH was entered, but there are no PATH entries in the HISTORY table. Debug Tool was not invoked for any PATH hooks.

EQA1789W Requested register(s) not available.

Explanation: You are trying to work with a register but none exist in this context (for example, during environment initialization).

EQA1790W There are no active blocks.

Explanation: The LIST CALLS command was issued prior to any STEP or GO.

EQA1791E The pattern *pattern* is invalid.

Explanation: A pattern is invalid if it is longer than 128 bytes or has more than 16 parts. (Each asterisk and each name fragment forms a part.)

EQA1792S Only the ADDR and POINTER built-in functions may be used to specify an address in the LIST STORAGE command.

Explanation: LIST STORAGE(built-in function(...)) is invalid if the built-in function is not the ADDR or POINTER built-in function.

EQA1793S ENTRY, FILE, LABEL, AREA, EVENT or TASK variables are not valid in a LIST command.

Explanation: The contents of these program control variables can be displayed by using the HEX or UNSPEC built-in functions or by using the LIST STORAGE command.

EQA1806E The command element *character* is invalid.

Explanation: The command entered could not be parsed because the specified element is invalid.

EQA1807E The command element *character* is ambiguous.

Explanation: The command entered could not be parsed because the specified element is ambiguous.

EQA1808E The hyphen cannot appear as the last character in an identifier.

Explanation: COBOL identifiers cannot end in a hyphen.

EQA1809E Incomplete command specified.

Explanation: The command, as it was entered, requires additional command elements (for example, keywords, variable names).

Programmer Response: Refer to the definition of the command and verify that all required elements of the command are present.

EQA1810E End-of-source has been encountered after an unmatched comment marker.

Explanation: A /* ... was entered but an */ was not present to close the comment. The command is discarded.

Programmer Response: You must either add an */ to the end of the comment or explicitly indicate continuation with an SBCS hyphen.

EQA1811E End-of-source has been encountered after an unmatched quotation mark.

Explanation: The start of a constant was entered (a quotation mark started the constant) but another quotation mark was not found to terminate the constant before the end of the command was reached.

Programmer Response: There could be several solutions for this, among them:

1. You must either add a quotation mark to the end of the constant or explicitly indicate continuation (with an SBCS hyphen).
2. If DBCS is ON you should also verify that you didn't try to start a constant with an SBCS quotation mark and terminate it with a DBCS quotation mark (or vice versa).
3. You might have entered a character constant that contained a quotation mark -- and you didn't double it.

EQA1812E A decimal exponent is required.

Explanation: In COBOL, an E in a float constant must be followed by at least one decimal digit (optionally preceded by a sign). In C, if a + or - sign is specified after an E in a float constant, it must followed by at least one decimal digit.

EQA1813E Error reading DBCS character codes.

Explanation: An unmatched or nested shift code was found.

EQA1814E Identifier is too long.

Explanation: All identifiers must be contained in 255 bytes or less. COBOL identifiers must be contained in 30 bytes or less.

EQA1815E Invalid character code within DBCS name, literal or DBCS portion of mixed literal.

Explanation: A character code point was encountered that was not within the defined code values for the first or second byte of a DBCS character.

EQA1816E An error was found at line *line-number* in the current input file.

Explanation: An error was detected while parsing a command within a USE file, or within a file specified on the run-time TEST option. It occurred at the record number that was displayed.

EQA1817E Invalid hexadecimal integer constant specified.

Explanation: A hexadecimal digit must follow 0x.

EQA1818E Invalid octal integer constant specified.

Explanation: Only an octal digit can follow a digit-0.

EQA1819E A COBOL DBCS name must contain at least one nonalphanumeric double byte character.

Explanation: All COBOL DBCS names must have at least one double byte character not defined as double byte alphanumeric. For EBCDIC, these are characters with `X'42'` in the leading byte, with the trailing byte in the range `X'41'` to `X'FE'`.

EQA1820E Invalid double byte alphanumeric character found in a COBOL DBCS name. Valid COBOL double byte alphanumeric characters are: A-Z, a-z, 0-9.

Explanation: Alphanumeric double-byte characters have a leading byte of `X'42'` in EBCDIC and `X'82'` in ASCII. The trailing byte is an alphanumeric character. The valid COBOL subset of these is A-Z, a-z, 0-9.

EQA1821E The DBCS representation of the hyphen was the first or last character in a DBCS name.

Explanation: COBOL DBCS names cannot have a leading or trailing DBCS hyphen.

EQA1822E A DBCS Name, DBCS literal or mixed SBCS/DBCS literal may not be continued.

Explanation: Continuation rules do not apply to DBCS names, DBCS literals or mixed SBCS/DBCS literals. These items must appear on a single line.

EQA1823E An end of line was encountered before the end of a DBCS name or DBCS literal.

Explanation: An end of line was encountered before finding a closing shift-in control code.

EQA1824E A DBCS literal or DBCS name contains no DBCS characters.

Explanation: A shift-out shift-in pair of control characters were found with no intervening DBCS characters.

EQA1825E End-of-source was encountered while processing a DBCS name or DBCS literal.

Explanation: No closing Shift-In control code was found before end of file.

EQA1826E A DBCS literal was not delimited by a trailing quote or apostrophe.

Explanation: No closing quotation mark

EQA1827E Invalid separator character found following a DBCS name.

EQA1828E Fixed binary constants are limited to 31 digits.

Explanation: A fixed binary constant must be between -2**31 and 2**31 exclusive.

EQA1829E Fixed decimal constants are limited to 15 digits.

Explanation: A fixed decimal constant must be between -10**15 and 10**15 exclusive.

EQA1830E Float binary constants are limited to 109 digits.

Explanation: This limit applies to all PL/I FLOAT BINARY constants.

EQA1831E Float decimal constants are limited to 33 digits.

Explanation: This limit applies to all PL/I FLOAT DECIMAL constants.

EQA1832E Floating-point exponents are limited to 3 digits.

Explanation: This limit applies to all C float constants and to all PL/I FLOAT BINARY constants.

EQA1833E Float decimal exponents are limited to 2 digits.

Explanation: This limit applies to all PL/I FLOAT DECIMAL constants.

EQA1834E Float binary constants must be less than 1E+252B.

Explanation: This limit applies to all PL/I FLOAT BINARY constants.

EQA1835E Float decimal constants must be less than 7.2370055773322622139731865630429 8E+75.

Explanation: This limit applies to all PL/I FLOAT DECIMAL constants.

EQA1836E Float constants are limited to 35 digits.

Explanation: This limit applies to all C float constants.

EQA1837E Float constants must be bigger than 5.397605346934027890866469914250249 6E-79 and less than 7.237005577332262213973186563042992 9E+75.

Explanation: This is the range of values allowed by C.

EQA1872E An error occurred while opening file: ***file name*****.**

Explanation: An error occurred during the initial processing (OPEN) of the file.

EQA1873E An error occurred during an input or output operation.

Explanation: An error occurred performing an input or output operation.

EQA1874I The command ***command name*** **can be used only in full screen mode.**

Explanation: This command is one of a collection that is allowed only when your terminal is operating in full screen mode. The function is not supported in batch mode.

EQA1875I Insufficient storage available.

Explanation: This message is issued when not enough storage is available to process the last command issued or to handle the last invocation.

EQA1876E Not enough storage to display results.

Explanation: Increase size of virtual storage.

EQA1877E An error occurred in writing messages to the dump file.

Explanation: This could be caused by a bad file name specified with the call dump FNAME option.

EQA1878E The cursor is not positioned at a variable name.

Explanation: A command, such as LIST, LIST TITLED, LIST STORAGE, or DESCRIBE ATTRIBUTES, which takes input from the Source window was entered with the cursor in the Source window, but the cursor was not positioned at a variable name.

Programmer Response: Reposition the cursor and reenter.

EQA1879E The listing file name given is too long.

Explanation: Filenames are limited to 7 characters and file-ids are limited to 44 characters. If a sublibrary member is referenced the name and type can each be a maximum of 8 characters (enclosed in parentheses).

EQA1880E You may not resume execution when the program is waiting for input.

Explanation: The user attempted to issue a GO/RUN or STEP request when the program was waiting for input. The input must be entered to resume execution.

EQA1881E The INPUT command is only valid when the program is waiting for input.

Explanation: The user attempted to enter the INPUT command when the program was not waiting for any input.

EQA1882E The logical record length for *filename* is out of bounds. It will be set to the default.

Explanation: The logical record length is less than 32 bytes or greater than 256 bytes.

EQA1883E Error closing previous log file; Return code = *rc*

Explanation: The user attempted to open a new log file and the old one could not be closed; the new log file is used, however.

EQA1884E An error occurred when processing the source listing. Check return code *return code* in the Using the Debug Tool manual for more detail.

Explanation: An error occurred during processing of the list lines command. Possible return codes:

2 - The listing file could not be found or allocated.

5 - The CU was not compiled with the correct compile option.

7 - Failed due to inadequate resources.

EQA1902W The command has been terminated because of the attention request.

Explanation: The previously-executing command was terminated because of an attention request. Normal debugging can continue.

EQA1904E The STEP and GO/RUN commands are not allowed at termination.

Explanation: The STEP and GO/RUN commands are not allowed after the application program ends.

EQA1905W You cannot trigger a condition in your program at this time.

Explanation: The environment is in a position that it would not be meaningful to trigger a condition. For example, you have control during environment initialization.

EQA1906S The condition named *CONDITION name* is unknown.

Explanation: A condition name was expected, but the name entered is not the name of a known condition.

EQA1907W The attempt to trigger this condition has failed.

Explanation: For some reason, when Debug Tool tried to trigger the specified condition, it failed and the condition was not signaled.

EQA1918S The block name *block-qualification* :> *block_name* is ambiguous.

Explanation: There is another block that has the same name as this block.

Programmer Response: Provide further block name qualification—by phase name, by compile unit name, or by additional block names if a nested block.

EQA1919E The present block is not nested. You cannot QUALIFY UP.

Explanation: While you can QUALIFY to any block, you cannot QUALIFY UP (for example, change the qualification to the block's parent) unless there really is a parent of that block. In this case, there is no parent of the currently-qualified block.

Programmer Response: You have either misinterpreted your current execution environment or you have to qualify to some block explicitly.

EQA1920E The present block has no dynamic parent. You cannot QUALIFY RETURN.

Explanation: While you can QUALIFY to any block you cannot QUALIFY RETURN (for example, change the qualification to the block's invoker) unless there really is an invoker of that block. In this case, there is no invoker of the currently-qualified block.

Programmer Response: You have either misinterpreted your current run-time environment or you have to qualify to some block explicitly.

EQA1921S There is no block named *block_name*.

Explanation: The block that you named could not be located by Debug Tool.

Programmer Response: Provide further block name qualification—by phase name, by compile unit name, or by additional block name(s) if a nested block.

EQA1922S There is no block named *block_name* within block *block-qualification*.

Explanation: The qualification you are using (or the spelling of the block names) prevented Debug Tool from locating the target block.

Programmer Response: Verify that the named block should be within the current qualification.

EQA1923S There is no compilation unit named *cu_name*.

Explanation: The compilation unit (program) that you named could not be located by Debug Tool.

EQA1924S Statement *statement_id* is not valid.

Explanation: The statement number does not exist or cannot be used. Note that the statement number could exist but is unknown.

EQA1925S There is no phase named *phase name*.

Explanation: Phase name qualification is referring to a phase that cannot be located.

Programmer Response: The phase might be missing or it might have been loaded before Debug Tool was first used. Debug Tool is aware of additional phases ONLY if they were FETCHed after Debug Tool got control for the first time.

EQA1926S There is no cu named *cu_name* within phase *phase name*.

Explanation: The compilation unit might be misspelled or missing.

EQA1927S There are *number* CUs named *cu_name*, but neither belongs to the current phase.

Explanation: The compilation unit you named is not unique.

Programmer Response: Add further qualification so that the correct phase will be known.

EQA1928S The block name *block_name* is ambiguous.

Explanation: There is another block that has the same name as this block.

Programmer Response: Provide further block name qualification—by phase name, by compile unit name, or by additional block names if a nested block.

EQA1929S Explicit qualification is required because the location is unknown.

Explanation: The current location is unknown; as such, the reference or statement must be explicitly qualified.

Programmer Response: Either explicitly set the qualification using the SET QUALIFY command or supply the desired qualification to the command in question.

EQA1930S There is no compilation unit named *CU-name* in the current enclave.

Explanation: The compilation unit (program) that you named could not be located in the current enclave by Debug Tool.

EQA1931S There is no cu named *CU-name* within phase *phase name* in the current enclave.

Explanation: The compilation unit might be misspelled or missing, or it might be outside of the current enclave.

EQA1932S *Block or CU* **block_name** *is not currently available*

Explanation: The block or CU that you named could not be located by Debug Tool.

Programmer Response: Provide further block name qualification--by phase name, by compile unit name, or by additional block names(s) if a nested block.

EQA1940E *variable name* is a not a level-one identifier.

Explanation: You are trying to clear an element of a structure. You must clear the entire structure by naming its level-one identifier.

EQA1941E ATANH(x) is undefined if x is REAL and ABS(x) >= 1.

Explanation: This applies to the PL/I ATANH built-in function.

EQA1942E LOG(z) is undefined if z is COMPLEX and z = 0.

Explanation: This applies to the PL/I LOG built-in function.

EQA1943E *built-in function* (x) is undefined if x is REAL and x <= 0.

Explanation: This applies to the PL/I LOG, LOG2 and LOG10 built-in functions.

EQA1944E *built-in function* (x,y) is undefined if x=0 and y=0.

Explanation: This applies to the PL/I ATAN and ATAND built-in functions.

EQA1950E The MONITOR table is empty. If the first MONITOR command entered is numbered, it must have number 1.

Explanation: A MONITOR n command was issued when the MONITOR table is empty, but n is greater than 1.

EQA1951E The number of entries in the MONITOR table is *monitor-number*. New MONITOR commands must be unnumbered or have a number less than or equal to *monitor-number*.

Explanation: A MONITOR n command was issued but n is greater than 1 plus the highest numbered MONITOR command.

EQA1952E The MONITOR command table is full. No unnumbered MONITOR commands will be accepted.

Explanation: A MONITOR command was issued but the MONITOR table is full.

EQA1953E No command has been set for MONITOR *monitor-number*.

Explanation: A LIST MONITOR n or CLEAR MONITOR n command was issued, but n is greater than the highest numbered MONITOR command.

EQA1954E The command for MONITOR *monitor-number* has already been cleared.

Explanation: A CLEAR MONITOR n command was issued, but MONITOR has already been cleared.

EQA1955E There are no MONITOR commands established.

Explanation: A LIST MONITOR or CLEAR MONITOR command was issued, but there are no MONITOR commands established.

EQA1956E No previous FIND argument exists. FIND operation not performed.

Explanation: A FIND command must include a string to find when no previous FIND command has been issued.

EQA1957E String could not be found.

Explanation: A FIND attempt failed to find the requested string.

EQA1960S There is an error in the definition of variable *variable name*. Attribute information cannot be displayed.

Explanation: The specified variable has an error in its definition or length and address information is not currently available in the execution of the program.

EQA1963S The *command* command is not supported on this platform.

Explanation: The given command is not supported on the current platform.

EQA1964E Source or Listing data is not available.

Explanation: The source or listing information is not available. Some of the possible conditions that could cause this are: The listing file could not be found, the CU was not compiled with the correct compile options, inadequate resources were available.

EQA1965E Attributes of source of assignment statement conflict with target *variable name*. The assignment cannot be performed.

Explanation: The assignment contains incompatible data types; the assignment cannot be made.

EQA1966E The AREA condition would have been raised during an AREA assignment, but since WARNING is on, the assignment will not be performed.

Explanation: The operation, if performed, would result in the AREA condition. The condition is being avoided by rejecting the operation.

EQA1967E The subject of the *built-in function name* pseudovariable (*character string*) must be complex numeric.

Explanation: You are trying to get apply the PL/I IMAG or REAL pseudovariable to a variable that is not complex numeric.

EQA1968W You cannot use the GOTO command at this time.

Explanation: The program environment is such that a GOTO cannot be performed correctly. For example, you could be in control during environment initialization and base registers (supporting the GOTO'd logic) have not been established yet.

EQA1969E GOTO *label-constant* will not be permitted because that constant is the label for a FORMAT statement.

Explanation: There are several statement types that are not allowable as the target of a GOTO. FORMAT statements are one of them.

EQA1970E The *3-letter national language code* national language is not supported for this installation of Debug Tool. Uppercase United States English (UEN) will be used instead.

Explanation: The national-language-specified conflicts with the supported national languages for this installation of Debug Tool.

Programmer Response: Verify that the Language Environment run-time NATLANG option is correct.

EQA1971E The return code in the QUIT command must be nonnegative and less than 1000.

Explanation: For PL/I, the value of the return code must be nonnegative and less than 1000.

EQA1972E variable name *is not a LABEL constant No AT commands can be issued against it*

Explanation: LABEL variables may not be the object of the AT command.

EQA1973E The FIND argument cannot exceed a string length of 64

Explanation: Shorten the search argument to a string length 64 or less.

EQA1974E The FIND argument is invalid, the string length is zero

Explanation: Supply a search argument inside the quotes.

EQA1975E *error message string*

Explanation: Unable to evaluate the expression. See output string provided.

EQA1987E Debugger terminated, execution continues.

Explanation: The initialization of the connection to the specified terminal has failed. The debug tool is terminated and the execution of the batch application continues. Note the accompanying messages as to possible causes.

EQA1995E The VTAM ACB could not be opened to start the Debug Tool 3270 session.

Explanation: The Debug Tool APPL definitions may not have been defined to VTAM or all APPLs defined are currently in use. This message is written to the LE/VSE message file.

EQA1996E The VTAM logical unit defined for the 3270 session could not be acquired.

Explanation: The VTAM 3270 logical unit specified via the MFI parameter in the preferences file suboption of the LE/VSE TEST run-time option may not be defined in the VTAM network or may already be in use by another VTAM application. This message is written to the LE/VSE message file.

EQA1997S The Debug Tool 3270 session could not be initialized.

Explanation: The Debug Tool encountered a problem and was unable to initialize the 3270 session. Possible causes of the problem are:

- insufficient available storage
- an internal logic error.

This message is written to the LE/VSE message file.

EQA1998S The Debug Tool 3270 session has failed.

Explanation: The Debug Tool encountered a problem when sending to or receiving from the 3270 session. Possible causes of the problem are:

- shut down of VTAM
- loss of the terminal connection
- an internal logic error.

This message is written to the LE/VSE message file.

EQA2000E Incorrect data entered

Explanation: The data entered is incorrect. There could be several reasons for this:

- Missing right parenthesis in the Command File or Preference File name.
- Improperly defined Session Parameter

Programmer Response: Correct the entry where the cursor is positioned and invoke the function again. You can use the Help function (PF1) to find the context sensitive help for that field.

EQA2001E DTCN internal error

Explanation: DTCN discovered an internal error.

Programmer Response: Contact IBM service.

EQA2002E Internal CICS error

Explanation: During processing DTCN discovered an internal CICS error

Programmer Response: Correct the error and issue the command again. If the error persists contact your CICS system programmer and/or IBM service.

EQA2003E Key Not Defined.

Explanation: There is no action defined with the PF key used by the user.

Programmer Response: For more information about the actions defined for this panel use PF2 key for general help.

EQA2004E Add failed - profile exists

Explanation: The add command failed because the profile for that terminal & transaction is already stored in the Debug Tool Profile Repository.

Programmer Response: You can use Show(PF7) command to display the profile or modify the TermId+TranId and Add a new profile.

EQA2005E Replace failed - profile does not exist

Explanation: The profile for Terminal & Transaction Id does not exist in the Debug Tool Profile Repository and cannot be updated.

Programmer Response: Specify different Terminal+Transaction Id to update. You can use Next (PF8) command to browse the Profile Repository starting from any point.

EQA2006E Delete failed - profile does not exist

Explanation: The profile for Terminal & Transaction Id does not exist in the Debug Tool Profile Repository and cannot be updated.

Programmer Response: Specify different Terminal+Transaction Id to delete. You can use Next (PF8) command to browse the Profile Repository starting from any point.

EQA2007E Show failed - profile does not exist

Explanation: The profile for Terminal & Transaction Id does not exist in the Debug Tool Profile Repository.

Programmer Response: Specify different Terminal+Transaction Id to display. You can use Next (PF8) command to browse the Profile Repository from any point.

EQA2008E Next failed - profile does not exist

Explanation: There are no more profiles in the Debug Tool Profile Repository.

EQA2010I DTCN closed

Explanation: DTCN deleted all profiles stored in the Debug Tool Profiles Repository. This action affects all users working with that CICS partition.

EQA2011E Blank Terminal Id and Transaction Id not allowed

Explanation: DTCN cannot store debugging profile for blank Terminal Id and Transaction Id.

Programmer Response: Supply nonblank Terminal Id (for debugging application on that terminal) or Transaction Id (for debugging batch CICS transaction or troubleshooting transaction partition wide) or both.

EQA2012I Terminal Id blanked - partition wide debugging - press Enter to Confirm

Explanation: DTCN asks for additional confirmation for partition wide Transaction debugging.

Programmer Response: Blank Transaction Id (for debugging batch CICS transaction or troubleshooting transaction partition wide) requires confirmation. (Enter key) or negation (any other).

EQA2014I Debug Tool profile added

Explanation: A new profile was added to the Debug Tool Profile Repository

EQA2015I Debug Tool profile replaced

Explanation: Existing profile was updated in the Debug Tool Profile Repository.

EQA2016I Debug Tool profile deleted

Explanation: Existing profile was deleted from the Debug Tool Profile Repository

EQA2017E CICS terminal *TERM* is not acquired

Explanation: The terminal id specified to receive the Debug Tool screen was not acquired.

Programmer Response: Correct the Debug Tool Term Id using DTCN Replace function or logon to an already defined terminal.

Bibliography

Debug Tool Publications

Debug Tool for VSE/ESA

Fact Sheet, GC26-8925

User's Guide and Reference, SC26-8797

Installation and Customization Guide, SC26-8798

Language Environment Publications

IBM Language Environment for VSE/ESA

Fact Sheet, GC33-6679

Concepts Guide, GC33-6680

Debugging Guide and Run-Time Messages, SC33-6681

Installation and Customization Guide, SC33-6682

Licensed Program Specifications, GC33-6683

Programming Guide, SC33-6684

Programming Reference, SC33-6685

Run-Time Migration Guide, SC33-6687

Writing Interlanguage Communication Applications, SC33-6686

C Run-Time Programming Guide, SC33-6688

C Run-Time Library Reference, SC33-6689

LE/VSE-Conforming Language Product Publications

IBM C for VSE/ESA

Licensed Program Specifications, GC09-2421

Installation and Customization Guide, GC09-2422

Migration Guide, SC09-2423

User's Guide, SC09-2424

Language Reference, SC09-2425

Diagnosis Guide, GC09-2426

IBM COBOL for VSE/ESA

General Information, GC26-8068

Licensed Program Specifications, GC26-8069

Migration Guide, GC26-8070

Installation and Customization Guide, SC26-8071

Programming Guide, SC26-8072

Language Reference, SC26-8073

Diagnosis Guide, SC26-8528

Reference Summary, SX26-3834

IBM PL/I for VSE/ESA

Fact Sheet, GC26-8052

Programming Guide, SC26-8053

Language Reference, SC26-8054

Licensed Program Specifications, GC26-8055

Migration Guide, SC26-8056

Installation and Customization Guide, SC26-8057

Diagnosis Guide, SC26-8058

Compile-Time Messages and Codes, SC26-8059

Reference Summary, SX26-3836

Related Publications

CICS/VSE

System Definition and Operations Guide, SC33-0706

Customization Guide, SC33-0707

Resource Definition (Macro), SC33-0709

Application Programming Guide, SC33-0712

Application Programming Reference, SC33-0713

Problem Determination Guide, SC33-0716

CSP

Developing Applications, SH20-6435

DFSORT/VSE

Application Programming Guide, SC26-7040

DL/I DOS/VS

Application Programming: CALL and RQDLI Interfaces, SH12-5411

Application Programming: High Level Programming Interface, SH24-5009

DOS/VS PL/I

Programmer's Guide, SC33-0008

QMF

Developing QMF Applications, SC26-4722

© Copyright IBM Corp. 1995, 1996

SQL/DS

Application Programming Guide for VSE, SH09-8098

VS COBOL II

Application Programming Guide for VSE, SC26-4697

VSE/ESA Version 1 Release 4

Planning, SC33-6503

Administration, SC33-6505

Messages and Codes, SC33-6507

Guide to System Functions, SC33-6511

System Control Statements, SC33-6513

System Macros User's Guide, SC33-6515

System Macros Reference, SC33-6516

VSE/VSAM Commands and Macros, SC33-6532

VSE/VSAM User's Guide, SC33-6535

Release Information Guide, SC33-6536

VSE/ESA Version 2

Planning, SC33-6603

Administration, SC33-6605

Messages and Codes, SC33-6607

Guide to System Functions, SC33-6611

System Control Statements, SC33-6613

System Macros User's Guide, SC33-6615

System Macros Reference, SC33-6616

VSE/VSAM Commands and Macros, SC33-6532

VSE/VSAM User's Guide, SC33-6535

Softcopy Publications

The following collection kit contains the LE/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

You can order these publications from Mechanicsburg through your IBM representative.

Glossary

A

abend. Abnormal end of application.

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.

addressing mode. An attribute that refers to the address length that a routine is prepared to handle upon entry. Addresses may be 24 or 31 bits long.

alias. An alternative name for a field used in some high-level programming languages.

AMODE. see *addressing mode*.

animation. The execution of instructions one at a time with a delay between each so that any results of an instruction can be viewed.

application. A collection of one or more routines cooperating to achieve particular objectives.

application program. See *program*

array. An aggregate that consists of data objects, each of which may be uniquely referenced by subscripting.

array element. A data item in an array.

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.

automatic call library. Contains object modules that are to be used as secondary input to the linkage editor to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library may be:

- Sublibraries containing object modules, with or without linkage editor control statements
- The sublibrary containing LE/VSE run-time routines (PRD2.SCEEBASE or PRD2.SCEECICS)

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 Debug Tool which 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.

byte. The basic unit of storage addressability, usually with a length of 8 bits.

C

callable services. A set of services that can be invoked by an LE/VSE-conforming high level language using the conventional LE/VSE-defined call interface, and usable by all programs sharing the LE/VSE conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CICS. see *Customer Information Control System.*

CICS run unit. Consists of a statically and/or dynamically bound set of one or more phases which can be loaded by a CICS loader. A CICS run unit is equivalent to an LE/VSE *enclave*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

© Copyright IBM Corp. 1995, 1996

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.

common anchor area (CAA). Dynamically acquired storage that represents an LE/VSE thread. Thread-related storage/resources are anchored off the CAA. This area acts as a central communications area for the program, holding addresses of various storage and error-handling routines, and control blocks. The CAA is anchored by an address in register 12.

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 may 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. See also *exception*.

conversational. A transaction type which accepts input from the user, performs a task, then returns to get more input from the user.

currently-qualified. See *qualification*.

Customer Information Control System (CICS). CICS is an OnLine Transaction Processing (OLTP) system that provides specialized interfaces to databases, files and terminals in support of business and commercial applications.

D

data type. A characteristic that determines the kind of value that a field can assume.

DBCS. See *double-byte character set*.

debug. To detect, diagnose, and eliminate errors in programs.

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.

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*.

E

enclave. An independent collection of routines in LE/VSE, one of which is designated as the MAIN program. The enclave is roughly analogous to a program or routine.

entry point. The address or label of the first instruction executed on entering a computer program, routine, or subroutine. A computer program may have a number of different entry points, each perhaps corresponding to a different function or purpose.

environment. A set of services and data available to a program during execution. In LE/VSE, environment is normally a reference to the run-time environment of HLLs at the enclave level.

exception. An abnormal situation in the execution of a program which 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*.

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.

F

file. A named set of records stored or processed as a unit.

file-id. For a sequential disk file, a 1-to 44-character unique name associated with a file on a given disk volume. For a SAM ESDS file, a 1-to 44-character unique name identical to the name of the file in the VSAM catalog.

filename. A 1- to 7-character name used within an application and in JCL to identify a file. The filename provides the means for the logical file to be connected to the physical file.

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 which displays a variety of information about the program you are debugging.

H

hexadecimal. A base 16 numbering system. Hexadecimal digits range from 0 through 9 (decimal 0 to nine) and uppercase or lowercase A through F (decimal ten to fifteen).

high-level language (HLL). A programming language above the level of assembler language and below that of program generators and query languages. 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 at compile-time. Using a hook, you can set breakpoints to instruct Debug Tool to gain control of the program at selected points during its execution.

I

ILC. see *interlanguage communication.*

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*.

initial setting. A value in effect when the user's Debug Tool session begins. Contrast with *default*.

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

interrupt. A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.

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.

J

JCL. see *job control language.*

job control language (JCL). A sequence of commands used to identify a job to an operating system and to describe a job's requirements.

job step. One of a group of related programs complete with the JCL statements necessary for a particular run. Every job step is identified in the job stream by an EXEC statement under one job statement for the whole job.

L

Language Environment. A set of architectural constructs and interfaces that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

Language Environment for VSE/ESA. An IBM software product that provides a common run-time environment and common run-time services for IBM high-level language compilers on the VSE platform.

LE/VSE. see *Language Environment for VSE/ESA.*

library routine. A routine maintained in a program library.

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 phase.

listing. A printout that lists the source language statements of a program with all preprocessor statements, includes, and macros expanded.

Log window. A Debug Tool window that records and displays interactions with Debug Tool during a debugging session.

main program. The first routine in an enclave to gain control from the invoker.

megabyte (M). 1,048,576 bytes.

module. A language construct that consists of procedures or data declarations and can interact with other such constructs. In PL/I, an external procedure.

Monitor window. A Debug Tool window that is used to display output generated by the Debug Tool MONITOR command.

N

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

non-LE/VSE conforming. Any HLL program that does not adhere to LE/VSE's common interface. For example, VS COBOL II, DOS/VS COBOL, and DOS/VS PL/I are all non-LE/VSE conforming HLLs.

nonconversational. A transaction type which accepts input, performs a task, and then ends.

O

object code. Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

object deck. See *object module.*

object module. A portion of an object program suitable as input to a linkage editor. Synonym for *object deck*.

online. (1) Pertaining to a user's ability to interact with a computer. (2) Pertaining to a user's access to a computer via a terminal.

Options. A choice that lets the user customize objects or parts of objects in an application.

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.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested (child) enclave. See also *child enclave* and *nested enclave*.

partition. A fixed-size division of storage.

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.

phase. A program in a form suitable for loading into main storage for execution. The application or routine has been compiled and link-edited; that is, address constants have been resolved.

pointer. A data element that indicates the location of another data element.

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.

preprocessor. A routine that examines application source code for preprocessor statements that are then executed, resulting in the alteration of the source.

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.

process. The highest level of the LE/VSE program management model. It is a collection of resources, both program code and data, and consists of at least one enclave.

profile. A group of customizable settings that govern how the user's session appears, and how it operates (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 phase 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. (1) In programming languages, a language construct designating a declared language object. (2) 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.

return code. A code produced by a routine to indicate its success. It may be used to influence the execution of succeeding instructions.

RMODE. Residence mode. The attribute of a phase that specifies whether the phase, when loaded, must reside below the 16MB virtual storage line or may reside anywhere in virtual storage.

run. (1) To cause a program, utility, or other machine function to execute. (2) 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.

Run. A choice the user can make to start or resume regular execution of a program.

run time. Any instant at which 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

SAM. See *sequential access method*.

SAM ESDS file. A SAM file managed in VSE/VSAM space.

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 a file.

sequential access method (SAM).. A data access method that writes to and reads from an I/O device record after record (or block after block).

sequential disk file.. A disk file in which records are processed in the order in which they are entered and stored.

session. The events that take place between the time the user starts an application and the time the user exits the application.

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.

source. (1) The HLL statements in a file that make up a program. (2) The input to a compiler or assembler, written in a source language. (3) A set of instructions written in a programming language that must be translated to machine language before the program can be run.

source code. See *source*.

source program. See *source*.

Source window. A Debug Tool window that contains a display of either the source code or the listing of the program being debugged.

statement. In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static. In programming languages, pertaining to properties that can be established before execution of a program; for example, the length of a fixed length variable is static. Contrast with *dynamic*.

step. (1) One statement in a computer routine. (2) To cause a computer to execute one or more statements.

storage. (1) A unit into which recorded text can be entered, in which it can be retained, and from which it can be retrieved. (2) The action of placing data into a storage device. (3) A storage device.

suboption. An option that can be used with compile-time and run-time options to further specify the action of the option.

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.

subsystem. A secondary or subordinate system, or programming support, usually capable of operating independently of or asynchronously with a controlling system. Example: CICS.

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 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*.

system abend. An abend caused by the operating system's inability to process a routine; may be caused by errors in the logic of the source routine.

T

temporary variable. See *session variable*.

token. A character string in a specific format that has some defined significance in a programming language.

translator. See *CICS translator*.

trigraph. A group of three characters which, taken together, are equivalent to a single special character.

U

user abend. A request made by user code to the operating system to abnormally terminate a routine. Contrast with *system abend*.

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.

W

window. An area of a screen with visible boundaries within which information is displayed.

word wrap. See *line wrap*.

Index

A

abbreviating commands 91
abbreviating keywords 195
abnormal end of application, setting breakpoint at 131
accessing PL/I program variables 178
active block, definition of 393
%ADDRESS variable
  description of 127
  for C 142
  for COBOL 169
  for PL/I 183
alias, definition of 393
ALLOCATE, AT command (PL/I), syntax 210
allowable comparisons for Debug Tool IF command 349
allowable moves for Debug Tool MOVE command 351
allowable moves for Debug Tool SET command 352
%AMODE variable
  description of 127
  for C 142
  for COBOL 169
  for PL/I 183
ANALYZE command (PL/I), syntax 206
animation, definition of 393
APPEARANCE, AT command, syntax 211
assigning values to variables 140, 164, 181
Assignment command (PL/I), syntax 207
AT ALLOCATE command (PL/I), syntax 210
AT APPEARANCE command, syntax 211
AT CALL command, syntax 213
AT CHANGE command, syntax 215
AT CURSOR command (Full-Screen Mode), syntax 218
AT DELETE command, syntax 219
AT ENTRY command, syntax 220
AT EXIT command, syntax 220
AT GLOBAL command, syntax 221
AT LABEL command, syntax 222
AT LINE command
  See AT STATEMENT command, syntax
AT LOAD command, syntax 224
AT OCCURRENCE command, syntax 225
AT PATH command, syntax 228
AT prefix (Full-Screen Mode), syntax 229
AT STATEMENT command, syntax 230
AT TERMINATION command, syntax 231
AT breakpoint
  LIST AT command 273
  removing 240
AT commands 208, 232
  summary table 208
attention (ATTN) interrupt
  definition of 393
  effect of during interactive sessions 133
  how to initiate 133
  in Debug Tool 133
  required LE/VSE run-time options 133
attention key, definition of 393
attribute, definition of 393
attributes of variables 124
  for C 246
  for COBOL 246
  for PL/I 246

B

basic tasks of Debug Tool 54
batch job, definition of 393
batch mode
  See modes
batch, definition of 393
BEGIN command (PL/I), syntax 232
blanks, significance of 197
block
  definition of 393
  using, for C 155
block command (C), syntax 233
%BLOCK variable
  description of 127
  for C 142
  for COBOL 169
  for PL/I 183
block_name, description of 200
block_spec, description of 200
BOTTOM, SCROLL command
  See SCROLL commands
break command (C), syntax 233
breakpoints
  definition of 393
  in unknown compile unit 212
  removing 240
  using within multiple enclaves 104
built-in functions 134
  Debug Tool, using with C 147
  Debug Tool, using with PL/I 187
  for PL/I 187
  %GENERATION 135
  %HEX 134, 147, 188
  %INSTANCES 134, 148, 188
  %RECURSION 135, 148, 188
  %STORAGE 134, 148, 188

© Copyright IBM Corp. 1995, 1996

C

C
  attributes for variables 140, 246
  declarations, syntax 247
  equivalents for LE/VSE conditions 347
  notes on using 195
  reserved keywords 346
C commands
  block 233
  break 233
  do/while 256
  Expression 261
  for 263
  if 268
  INPUT 271
  interpretive subset of 346
  SET INTERCEPT 311
  SET WARNING 325
  switch 328
  while 335
%CAAADDRESS variable
  description of 127
  for C 143
  for COBOL 169
  for PL/I 183
CALL %DUMP command
  syntax 235
CALL commands 234—240
  summary table 234
CALL entry_name command (COBOL)
  syntax 239
CALL procedure, syntax 240
CALL, AT command, syntax 213
calls, function, for C 146
CALLS, LIST command, syntax 275
CEETEST callable service
  examples, for C 45
  examples, for COBOL 47
  examples, for PL/I 48
  invoking Debug Tool with 43
  syntax 44
CEEUOPT options module
  preparing and using to invoke Debug Tool 114
CEEUOPT run-time options module 116
CHANGE, AT command, syntax 215
CHANGE, SET command, syntax 303
changing point of view
  for C 158
  for COBOL 176
  for PL/I 191
changing source file in window 55
changing window layout in the session panel 95
character set 194
characters, searching 93
CICS
  debug modes under 108
  DTCN utility 109
  invoking Debug Tool with compile-time directives 114
  mechanisms for invoking Debug Tool under 109
  preparing and using CEEUOPT to invoke Debug Tool 114
  preparing to use DTCN utility 109
  requirements for using Debug Tool in 107
  restrictions for debugging 114
CLEAR commands 240—244
CLEAR prefix (Full-Screen Mode), syntax 243
CLOSE, WINDOW command
  See WINDOW commands
closing Debug Tool session panel windows 96
COBOL
  attributes for variables 167, 246
  command format 163
  declarations 250
  notes on using 195
  reserved keywords 349
COBOL commands
  CALL entry_name 239
  COMPUTE 245
  declarations 250
  EVALUATE 260
  IF 269
  INPUT 271
  interpretive subset of 349
  MOVE 286
  PERFORM 291
  SET 326
  SET INTERCEPT 311
coexistence of Debug Tool with other debug tools 342
coexistence with unsupported HLL modules 342
Color Selection panel 98
COLOR, SET command, syntax 304
colors
  changing in session panel 97
COLORS, PANEL command
  See PANEL commands
command format
  Debug Tool 194
  for COBOL 163
Command line, Debug Tool 89
command list, definition of 394
command sequencing, full-screen mode 90
command syntax help, getting for session 102
commands
  abbreviating 91, 195
  delimiting 232
  entering 194
  entering Debug Tool 85
  entering multiple line, without continuation 197
  for C, Debug Tool subset 138

commands *(continued)*
for COBOL, Debug Tool subset 160
for PL/I, Debug Tool subset 177
getting online help for 199
interpretive subsets, description of 129
interpretive subsets, for C 346
interpretive subsets, for COBOL 349
interpretive subsets, for PL/I 353
issuing
in a Debug Tool session 89
multiline 196
order of processing, Debug Tool 90
prefix, using in Debug Tool 90
restrictions, COBOL 161
retrieving from Log and Source windows, Debug Tool 199
retrieving with RETRIEVE command 92
commands file
description 29
using log file as 88
COMMENT command, syntax 244
comments, inserting into command stream 198
common syntax elements 200
compile requirements
for DL/I 115
for SQL/DS 119
compile unit, definition of 394
compile units
general description 130
name area, Debug Tool 89
qualification of, for C 156
qualification of, for COBOL 174
qualification of, for PL/I 189
record of associations, Debug Tool 289
compile-time option, TEST
for C 12
for COBOL 16
for PL/I 19
using #pragma statement to specify 16
using for DL/I 115
using for SQL/DS 119
compile_unit_name, description of 201
compile, definition of 394
compiler, definition of 394
COMPUTE command (COBOL)
restrictions on 161
syntax 245
using to assign values to variables 165
%CONDITION variable
description of 127
for C 143
for COBOL 169
for PL/I 183
condition, definition of 394
conditions
constants, for C 331
conditions *(continued)*
for C 226
handling of 132, 353
LE/VSE, C equivalents 347
constants
Debug Tool interpretation of HLL 125
entering 198
PL/I 186
using in expressions, for COBOL 173
continuation character
Debug Tool 90
for COBOL 163
using in full-screen mode 196
continuing lines 196
conversational, definition of 394
%COUNTRY variable
description of 127
for C 143
for COBOL 169
for PL/I 184
COUNTRY, SET command, syntax 306
__ctest function call
examples 50
invoking Debug Tool with 49
syntax 50
%CU variable
description of 127
for C 143
for COBOL 169
for PL/I 184
cu_spec, description of 201
CURSOR command
syntax 246
using 93
cursor commands, full-screen mode
CLOSE command 96
CURSOR command 93
FIND command 93
OPEN command 96
SCROLL commands 85, 93
SIZE command 96
using in Debug Tool 91
WINDOW ZOOM command 97
CURSOR, AT command, syntax 218
CURSOR, LIST command (Full-Screen Mode), syntax 276
customizing
profile settings 99
session settings 94
customizing screens 84

D

data type, definition of 394
DBCS
definition of 394

DBCS *(continued)*
  SET DBCS command, syntax 306
  using 194
  using with C 195
  using with COBOL 166
  variable, assigning new value to 286
DBCS, SET command, syntax 306
debug session
  C tasks
  COBOL tasks 71
  ending 54
  invoking your program 53
  PL/I tasks 80
  preparing for 53
  using a C program 57
  using a COBOL program 67
  using a PL/I program 76
Debug Tool
  C commands, interpretive subset 138
  COBOL commands, interpretive subset 160
  commands, subset 129
  condition handling 132
  evaluation of HLL expressions 125
  exception handling, for C and PL/I 133
  functions, using with C 147
  functions, using with PL/I 187
  IF command, allowable comparisons 349
  interface 84
  interpretation of HLL variables 125
  MOVE command, allowable moves 351
  multilanguage programs, using 22
  optimized programs, using with 344
  PL/I commands, interpretive subset 177
  procedure, definition of 394
  sample Debug Tool session 5
  SET command, allowable moves 352
  SQL/DS programs, using with 120
  using in batch mode 107
  variable, definition of 394
  variables
    general description 125
    using in C 140
    using in COBOL 167
    using in PL/I 181
Debug Tool interface 54
Debug Tool, invoking your program with 53
debug tools, other, coexistence with 342
debug, definition of 394
debugging in full-screen mode 53
debugging SQL/DS programs 118
declarations, for C, syntax 247
declarations, for COBOL 250
declarations, syntax 246
DECLARE command (PL/I) 251
declaring temporary variables, for C 139
declaring temporary variables, for COBOL 165
declaring temporary variables, for PL/I 180
DEFAULT LISTINGS, SET command, syntax 307
DEFAULT SCROLL, SET command
  See SET DEFAULT SCROLL command
DEFAULT WINDOW, SET command, syntax 308
default, definition of 394
DELETE, AT command, syntax 219
DESCRIBE command
  syntax 253
  using 135, 156
diagnostics, expression, for C 149
DISABLE command, syntax 255
DISPLAY, SET SCROLL command
  See SET SCROLL DISPLAY command
displaying
  DTCN 110
  environment information 135, 156
  lines at top of window, Debug Tool 93
  values of COBOL variables 166
displaying halted location 56
displaying variable value 56
DL/I
  programming considerations 115
  programs, debugging in batch mode 118
  programs, debugging in interactive mode 118
  using Debug Tool with 115
DO command (PL/I), syntax 256
do/while command (C), syntax 256
double-byte character set (DBCS), definition of 394
DOWN, SCROLL command 93
  See also SCROLL commands
DTCN utility
  data entry errors 113
  invoking Debug Tool under CICS 109
  modifying LE/VSE options 113
  PF key definitions 112
  preparing to use 109
  profile repository 113
  screen areas 110
  transaction screen 110
dual terminal mode (CICS) 108
%DUMP, CALL command, syntax 235
  See also CALL %DUMP command
dynamic, definition of 394

E

ECHO, SET command, syntax 308
elements, unsupported, for PL/I 354
ENABLE command, syntax 259
enclave
  definition of 394
  multiple, debugging interlanguage communication application in 124
  multiple, invoking Debug Tool within 103

enclave *(continued)*
multiple, overview 103
ending a debug session 54
ending Debug Tool within multiple enclaves 104
entering commands
in a Debug Tool session 89
using program function keys 91
entering multiline commands without continuation 197
entering PL/I statements, freeform 186
entry point, definition of 394
entry_name, CALL command (COBOL), syntax 239
ENTRY, AT command, syntax 220
%EPA variable
description of 127
for C 143
for COBOL 169
for PL/I 184
%EPRn (floating-point registers) variables
description of 127
for C 142
for COBOL 169
for PL/I 183
equate symbols
See SET EQUATE command
EQUATE, SET command
See SET EQUATE command
error numbers in Log window 57
EVALUATE command (COBOL)
restrictions on 162
syntax 260
evaluating expressions
C 149
COBOL 172
HLL 125
evaluation, expression, for C 145
every_clause, description 209
examples
%HEX function for C 147
%HEX function for COBOL 174
%HEX function for PL/I 188
%INSTANCES function for C 148
%INSTANCES function for PL/I 188
%STORAGE function for C 148
%STORAGE function for COBOL 174
%STORAGE function for PL/I 188
assigning values to variables, for C 140
assigning values to variables, for PL/I 181
CEETEST calls, for PL/I 48
CEETEST function calls, for C 45
CEETEST function calls, for COBOL 47
changing point of view, for C 158
changing point of view, for COBOL 176
changing point of view, for PL/I 191
changing point of view, general 131
ctest function 50
declaring variables, for COBOL 166
examples *(continued)*
displaying program variables, for C 139
displaying program variables, for PL/I 178
displaying results of expression evaluation, for COBOL 173
displaying values of COBOL variables 166
expression evaluation, for C 146
function calls, for C 148
getting online command syntax help 199
line continuation, for C 197
line continuation, for COBOL 197
PLITEST calls for PL/I 51
run-time TEST option 40
sample Debug Tool session 5
scope, for C 153
specifying run-time TEST option with #pragma 41
specifying run-time TEST option with PLIXOPT 42
using #pragma for compile-time TEST option 16
using blocks in C 155
using COMPUTE command to assign values 165
using constants 198
using constants in expressions, for COBOL 173
using continuation characters 196
using Debug Tool with OPTIMIZE compile-time option 344
using MOVE command to assign values 165
using qualification, for C 156, 158
using qualification, for COBOL 175
using qualification, for PL/I 190
using SET command to assign values 164
exception handling for C and PL/I 133
exception, definition of 394
execute, definition of 394
execution time, definition of 394
execution-time environment, definition of 394
EXIT, AT command, syntax 220
expression
definition of 394
description of 202
diagnostics, for C 149
displaying values, for C 139
displaying values, for COBOL 173
displaying values, for PL/I 178
evaluation for C 145, 149
evaluation for COBOL 172
evaluation of HLL 125
evaluation, operators and operands for C 346
for PL/I 186
subset, description of 203
using constants in, for COBOL 173
Expression command (C), syntax 261
Expression, LIST command, syntax 276

F

file, definition of 395
FIND command
  using with windows 93
FIND command, syntax 262
finding characters or strings 93
finding renamed source file 57
finding text in window 54
for command (C), syntax 263
%FPRn (floating-point registers) variables
  description of 127
  for C 142
  for COBOL 168
  for PL/I 183
freeform input, PL/I statements 186
frequency count
  definition of 395
FREQUENCY, LIST command
  See LIST FREQUENCY command
FREQUENCY, SET command, syntax 310
full-screen mode commands
  AT CURSOR 218
  AT prefix 229
  CLEAR prefix 243
  continuation character, using in 196
  CURSOR 91, 93, 246
  definition of 395
  DESCRIBE CURSOR 253
  DISABLE Prefix 255
  ENABLE Prefix 259
  FIND 262
  IMMEDIATE 270
  LIST CURSOR 276
  PANEL 289
  PANEL COLORS 97
  PANEL LAYOUT 95
  PANEL LISTINGS 289
  PANEL PROFILE 99
  PANEL SOURCE 289
  Prefix 292
  QUERY Prefix 297
  RETRIEVE 298
  SCROLL 93
  SET COLOR 304
  SET DEFAULT SCROLL 307
  SET DEFAULT WINDOW 308
  SET KEYS 313
  SET LOG NUMBERS 314
  SET MONITOR NUMBERS 314
  SET SCREEN 321
  SET SCROLL DISPLAY 321
  SET SUFFIX 323
  SHOW Prefix 326
  WINDOW CLOSE 96, 336
  WINDOW OPEN 96, 336
full-screen mode commands *(continued)*
  WINDOW SIZE 96, 337
  WINDOW ZOOM 97, 338
full-screen mode, debugging in 53
full-screen mode, using session panel in 85
function calls, for C 146
function, unsupported for PL/I 354

G

%GENERATION function, for PL/I 135, 188
GLOBAL, AT command, syntax 221
GO command
  syntax 265
GOTO command
  syntax 266
GOTO LABEL command, syntax 267
%GPRn (general purpose registers) variables
  description of 127
  for C 142
  for COBOL 168
  for PL/I 182

H

H constant (COBOL) 198
halted location, displaying 56
%HARDWARE variable
  description of 127
  for C 143
  for COBOL 169
  for PL/I 184
header fields, Debug Tool session panel 84
help, getting
  command syntax 199
  for session 102
help, how to find 54
%HEX function 147, 188
  for C 134
  for COBOL 134, 174
  for PL/I 134
high-level language (HLL), definition of 395
highlighting, changing in Debug Tool session panel 97
HISTORY, SET command, syntax 311
HLL, definition of 395
hooks
  compiling with, C 12
  compiling with, COBOL 16
  compiling with, PL/I 19
  definition of 395
  general description 4
  removing from application 343
  rules for placing 15

I

I/O, definition of 395
if command (C), syntax 268
IF command (COBOL)
  allowable comparisons, Debug Tool 349
  restrictions on 162
  syntax 269
IF command (PL/I), syntax 269
IMMEDIATE command, syntax 270
improving Debug Tool performance 343
inactive block, definition of 395
information
  displaying environmental 135, 156
initial setting, definition of 395
input areas, order of processing, Debug Tool 90
INPUT command
  syntax 271
instances 148, 188
%INSTANCES function
  for C 134
  for PL/I 134
interactive, definition of 395
INTERCEPT, SET command
  See SET INTERCEPT command
interlanguage communication (ILC) application, debugging 124
interlanguage programs, using with Debug Tool 22
interpretive subset
  general description 129
  of C commands 138, 346
  of COBOL commands 160, 349
  of PL/I commands 177, 353
interrupt, attention (ATTN)
  effect of during interactive sessions 133
  how to initiate 133
  in Debug Tool 133
  required LE/VSE run-time options 133
intrinsic functions, Debug Tool 125
invoking Debug Tool
  CEETEST callable service 43—49
  ctest function 49—51
  overview 42
  PLITEST built-in subroutine 51
  run-time TEST option 33—37
  under CICS 43
  within an enclave 103
invoking Debug Tool under CICS with compile-time directives 114
invoking your program 53
issuing commands
  in a Debug Tool session 89
  using program function keys 91

K

KEYS, SET command, syntax 313
keywords
  abbreviating 195
  reserved for C 346
  reserved for COBOL 349
  reserved for PL/I 353
  using with C 195

L

LABEL, AT command, syntax 222
LANGUAGE, SET NATIONAL command, syntax 315
LANGUAGE, SET PROGRAMMING command, syntax 317
LAST, LIST command, syntax 278
LAYOUT, PANEL command
  See PANEL commands
LE/VSE
  definition of 395
LEFT, SCROLL command 93
  See also SCROLL commands
librarian sublibrary
  See sublibrary
library routine, definition of 395
line breakpoint, setting 56
line continuation 196
LINE NUMBERS, LIST command, syntax 282
%LINE variable
  description of 127
  for C 143
  for COBOL 169
  for PL/I 184
line wrap, definition of 395
LINE, AT command, syntax 230
LINES, LIST command, syntax 283
link requirements
  for DL/I 116
  for SQL/DS 120
link-edit, definition of 395
linkage editor, definition of 395
LIST (blank) command, syntax 273
LIST AT command, syntax 273
LIST CALLS command, syntax 275
LIST commands 272—284
LIST CURSOR command (Full-Screen Mode), syntax 276
LIST Expression command, syntax 276
LIST FREQUENCY command
  syntax 277
LIST LAST command, syntax 278
LIST MONITOR command, syntax 279
LIST NAMES command
  syntax 279

LIST ON command (PL/I), syntax 281
LIST PROCEDURES command, syntax 281
LIST REGISTER command
  syntax 282
  using 137
LIST STATEMENT NUMBERS command, syntax 282
LIST STATEMENTS command, syntax 283
LIST STORAGE command
  syntax 283
  using with PL/I 181
listing
  definition of 396
  files used by Debug Tool 27
  registers 137
  SET DEFAULT LISTINGS command, syntax 307
LISTINGS, PANEL command
  See PANEL commands
literal constants, entering 198
%LOAD variable
  description of 128
  for C 143
  for COBOL 170
  for PL/I 184
load_spec, description of 203
LOAD, AT command, syntax 224
log file
  clearing 240
  default names 88
  description 30
  specifying 88
  using 87
  using as a commands file 88
LOG NUMBERS, SET command, syntax 314
Log window, Debug Tool
  Debug Tool 90
  definition of 396
  retrieving input lines from 199
  retrieving lines from 92
  using 87
Log window, error numbers in 57
LOG, SET command, syntax 313
low-level debugging 136
%LPRn (floating-point registers) variables
  description of 127
  for C 142
  for COBOL 168
  for PL/I 183

M

MainFrame Interface (MFI)
  general description 2
mechanisms for invoking Debug Tool under CICS 109
MFI (MainFrame Interface)
  See MainFrame Interface (MFI)
modes
  batch
    definition of 393
    FIND command 262
    using Debug Tool in 107
  debugging DL/I programs in 118
  debugging SQL/DS programs in 120
  dual terminal (CICS) 108
  non-terminal mode (CICS) 108
  single terminal (CICS) 108
MONITOR command
  clearing 240
  syntax 284
  viewing output from, Debug Tool 87
MONITOR NUMBERS, SET command, syntax 314
Monitor window, Debug Tool
  closing 96
  definition of 396
  general description 87
  invoking 96
MONITOR, LIST command, syntax 279
monitoring program execution
  in Debug Tool 87
more than one language, debugging programs with 22
MOVE command (COBOL)
  allowable moves, Debug Tool 351
  restrictions on 161
  syntax 286
  using to assign values to variables 165
moving around windows in Debug Tool 93
moving the cursor, Debug Tool 93
MSGID, SET command, syntax 315
multilanguage programs, using with Debug Tool 22
multiline commands
  using continuation character 196
  without continuation character 197
multiple enclaves
  ending Debug Tool 104
  interlanguage communication application,
    debugging 124
  invoking Debug Tool with 103
  multiple enclaves, overview 103
  using breakpoints 104

N

name scoping
  See point of view, changing
NAMES, LIST command
  See LIST NAMES command
NATIONAL LANGUAGE, SET command, syntax 315
navigating session panel windows 93
NEXT, SCROLL command
  See SCROLL commands
%NLANGUAGE variable
  description of 128

%NLANGUAGE variable *(continued)*
for C 143
for COBOL 170
for PL/I 184
non-terminal mode (CICS) 108
nonconversational, definition of 396
Null command, syntax 287
NUMBERS, LIST STATEMENT command, syntax 282
NUMBERS, SET LOG command, syntax 314
NUMBERS, SET MONITOR command, syntax 314

O

objects, C 153
OCCURRENCE, AT command, syntax 225
ON command (PL/I), syntax 287
ON, LIST command (PL/I), syntax 281
online help, getting for session 102
OPEN, WINDOW command
See WINDOW commands
opening Debug Tool session panel windows 96
operators and operands for C 346
OPTIMIZE compile-time option, using Debug Tool with 344
options module, CEEUOPT run-time 116
options, compile-time
for C 12
for COBOL 16
for PL/I 19
TEST, using for DL/I 115
TEST, using for SQL/DS 119
Options, definition of 396
options, run-time
TEST 33

P

PACE, SET command, syntax 316
PANEL commands
changing session panel colors and highlighting 97
syntax 289
using 95
panels, full-screen mode
Color Selection 98
definition of 396
header fields, Session 84
Profile 99
Session 85, 89
Source Identification 289
Window Layout Selection 95
parameter, definition of 396
path point
definition of 396
differences between languages 229
PATH, AT command, syntax 228
%PATHCODE variable
description of 128
for C 144
for COBOL 170
for PL/I 184
PERFORM command (COBOL)
restrictions on 162
syntax 291
performance, improving Debug Tool 343
PF key, setting 56
PF keys, defining in Debug Tool 91
PFKEY, SET command
See SET PFKEY command
phase_name, description of 202
phase, definition of 396
PL/I
attributes for variables 181, 246
built-in functions 187
condition handling 353
constants 186
expressions 186
notes on using 195
reserved keywords 353
session variables 180
statements 177
PL/I commands
ANALYZE 206
Assignment 207
AT ALLOCATE 210
BEGIN 232
conditions 353
DECLARE 251
DO 256
IF 269
interpretive subset of 353
LIST ON 281
ON 287
SELECT 301
SET WARNING 325
PL/I structures 179
%PLANGUAGE variable
description of 128
for C 144
for COBOL 171
for PL/I 185
PLITEST built-in subroutine
examples 51
syntax 51
PLIXOPT string
specifying run-time TEST option with 41
point of view
changing
for C 158
for COBOL 176
for PL/I 191
general description 131

positioning lines at top of windows 93
#pragma statement
  specifying compile-time TEST option 16
  specifying run-time TEST option with 41
preferences file
  default name 29
  description 29
Prefix area, Debug Tool 89
prefix area, definition of 396
prefix commands
  AT 229
  CLEAR 243
  DISABLE 255
  ENABLE 259
  QUERY 297
  SHOW 326
  syntax 292
  using in Debug Tool 90
preparing and using CEEUOPT to invoke Debug Tool 114
preparing for debugging 53
preparing programs
  compile-time TEST option, for C 12
  compile-time Test option, for COBOL 16
  compile-time TEST option, for PL/I 19
  considerations, size and performance 343
  for DL/I 115
  for SQL/DS 119
  run-time TEST option 33
preprocessor requirements for SQL/DS 119
primary entry point, definition of 394
PROCEDURE command
  syntax 293
procedure, CALL, syntax 240
procedure, definition of 396
PROCEDURES, LIST command, syntax 281
process, definition of 396
profile repository for DTCN utility 113
profile settings
  changing in Debug Tool 99
  file used to preserve 29
  panel 99
PROFILE, PANEL command
  See PANEL commands
profile, definition of 396
Program
  stepping through 56
program hooks
  compiling with, C 12
  compiling with, COBOL 16
  compiling with, PL/I 19
  general description 4
  removing 343
  rules for placing 15
program unit, definition of 396
%PROGRAM variable
  description of 128
  for C 144
  for COBOL 171
  for PL/I 185
program variable, definition of 397
PROGRAMMING LANGUAGE, SET command, syntax 317
programs
  CICS, debugging 107
  definition of 396
  DLI, debugging 115
  invoking for a session 42
  preparing DL/I 115
  preparing SQL/DS 119
  qualification of
    for C 156
    for COBOL 174
    for PL/I 189
    general description 130
  reducing size 343
  SQL/DS, debugging 118
  variables, accessing for C 138
  variables, accessing for COBOL 164
  variables, accessing for PL/I 178
pseudo-conversational transaction, definition of 397
PX contant (PL/I) 198

Q

qualification
  definition of 397
  description of, for C 156
  description of, for COBOL 174
  description of, for PL/I 189
  general description 130
  using, for C 157
  using, for COBOL 174
  using, for PL/I 189
QUALIFY, SET command
  See SET QUALIFY command
QUERY command, syntax 294
QUERY Prefix, syntax 297
QUIT command
  syntax 297

R

%RC (return code) variable
  description of 128
  for C 144
  for COBOL 171
  for PL/I 185
record format, definition of 397
record, definition of 397

recording
  commands, Debug Tool 87
  session with the log file 87
recursion 148, 188
%RECURSION function 148, 188
  for C 135
  for PL/I 135
reference, definition of 397
references, description of 203
REFRESH, SET command
  See SET REFRESH command
REGISTER, LIST command
  See LIST REGISTER command
registers, listing 137
removing statement and symbol tables 343
repeating breakpoints 209
requirements
  compile, for DL/I 115
  compile, for SQL/DS 119
  for debugging CICS programs 107
  link, for DL/I 116
  link, for SQL/DS 120
  preprocessor, for SQL/DS 119
reserved keywords
  for C 346
  for COBOL 349
  for PL/I 353
restrictions
  debugging under CICS 114
  expression evaluation, for COBOL 172
  on COBOL-like commands 161
RETRIEVE command
  syntax 298
  using 92
retrieving commands
  from the Log and Source windows 199
  with RETRIEVE command 92
retrieving lines from Log or Source windows 92
REWRITE, SET command, syntax 320
RIGHT, SCROLL command 93
  See also SCROLL commands
RUN command
  See GO command
run time
  definition of 397
  environment, displaying attributes of 135, 156
  option, TEST 33
  options module, CEEUOPT 116
  specifying TEST option with #pragma 41
  specifying TEST option with PLIXOPT 41
  suboption processing order 37
run unit, definition of 397
run-time environment, definition of 397
run-time TEST option
  processing order for suboptions 37
  specifying with #pragma 41
run-time TEST option *(continued)*
  specifying with PLIXOPT 41
  syntax 33
  using 33
run, definition of 397
Run, definition of 397
%RUNMODE variable
  description of 128
  for C 144
  for COBOL 172
  for PL/I 185
running a program 56

S

SAM ESDS file, definition of 397
SBCS (single-byte character set), definition of 397
scopes, C 153
SCREEN, SET command, syntax 321
screens, customizing 84
Scroll area, Debug Tool 89
SCROLL commands
  syntax 299
  using 85, 93
  using SCROLL TO 93
SCROLL DISPLAY, SET command
  See SET SCROLL DISPLAY command
SCROLL, SET DEFAULT command, syntax 307
scrolling
  session panel windows 93
scrolling Source window 55
searching for characters or strings 93
SELECT command (PL/I), syntax 301
semantic error, definition of 397
sequence number, definition of 397
session panels, Debug Tool
  changing colors and highlighting in 97
  changing window layout 95
  closing 96
  definition of 397
  header fields, Debug Tool 84
  modifying 89
  opening 96
  sizing windows in 96
  using 84
  variables, for PL/I 180
  windows, description of 85
session settings
  changing in Debug Tool 94
session variables
  definition of 397
SET CHANGE command, syntax 303
SET COLOR command, syntax 304
SET command (COBOL)
  allowable moves, Debug Tool 352
  restrictions on 162

SET command (COBOL) *(continued)*
  syntax 326
  using to assign values to variables 164
SET commands 302—326
SET COUNTRY command, syntax 306
SET DBCS command, syntax 306
SET DEFAULT LISTINGS command, syntax 307
SET DEFAULT SCROLL command
  syntax 307
  using 85
SET DEFAULT WINDOW command, syntax 308
SET ECHO command, syntax 308
SET EQUATE command
  clearing 240
  creating 92
  syntax 309
  using 92
SET EXECUTE command, syntax 310
SET FREQUENCY command, syntax 310
SET HISTORY command, syntax 311
SET INTERCEPT command
  syntax 311
  using with C programs 150
SET KEYS command, syntax 313
SET LOG command, syntax 313
SET LOG NUMBERS command, syntax 314
SET MONITOR NUMBERS command, syntax 314
SET MSGID command, syntax 315
SET NATIONAL LANGUAGE command, syntax 315
SET PACE command, syntax 316
SET PFKEY command
  syntax 316
  using in Debug Tool 91
SET PROGRAMMING LANGUAGE command, syntax 317
SET QUALIFY command
  syntax 318
  using, for C 158
  using, for COBOL 176
  using, for PL/I 191
SET REFRESH command
  syntax 320
SET REWRITE command, syntax 320
SET SCREEN command, syntax 321
SET SCROLL DISPLAY command
  syntax 321
  using 85
SET SOURCE command, syntax 322
SET SUFFIX command, syntax 323
SET TEST command, syntax 323
SET WARNING command (C and PL/I)
  syntax 325
  using with PL/I 187
setting a line breakpoint 56
setting a PF key 56
settings
  changing Debug Tool profile 99
  changing Debug Tool session 94
  profile settings file 29
short forms for commands, using 91
SHOW Prefix command, syntax 326
single terminal mode (CICS) 108
single-byte character set (SBCS), definition of 397
SIZE, WINDOW command
  See WINDOW commands
sizing session panel windows 96
source
  definition of 397
  displaying with Debug Tool 86
  files used by Debug Tool 27
Source Identification panel, Debug Tool 289
Source window
  Debug Tool 86, 90
  definition of 398
  retrieving input lines from, Debug Tool 199
  retrieving lines from, Debug Tool 92
SOURCE, SET command, syntax 322
SOURCES, PANEL command
  See PANEL commands
specifying a range of statements 204
SQL/DS
  programming considerations 119
  programs, debugging in batch mode 120
  programs, debugging in interactive mode 121
  using Debug Tool with 118
starting Debug Tool
  See invoking Debug Tool
STATEMENT NUMBERS, LIST command, syntax 282
statement tables, removing 343
%STATEMENT variable
  description of 128
  for C 145
  for COBOL 172
  for PL/I 185
statement_id_range, description of 204
statement_id, description of 203
statement_label, description of 205
STATEMENT, AT command, syntax 230
STATEMENTS, LIST command, syntax 283
statements, PL/I 177, 186
statements, specifying a range 204
static, definition of 398
STEP command
  syntax 327
step, definition of 398
stepping through a program 56
stmt_id_spec, description of 204
storage
  definition of 398
storage classes, C 154

%STORAGE function 148, 188
for C 134
for COBOL 134, 174
for PL/I 134
STORAGE, LIST command
See LIST STORAGE command
string substitution, using 92
strings
searching for using window 93
sublibrary
subroutine, definition of 398
subset
of C commands 346
of COBOL commands 349
of commands, general description 129
of PL/I commands 353
substitution, using string 92
%SUBSYSTEM variable
description of 129
for C 145
for COBOL 172
for PL/I 186
suffix area, definition of 398
SUFFIX, SET command, syntax 323
switch command (C), syntax 328
symbol tables, removing 343
synonyms
removing 240
syntactic analysis, definition of 398
syntax
common elements 200
definition of 398
error, definition of 398
examples of how to read 195
syntax diagrams
%GENERATION function 135
%HEX function 134
%INSTANCES function 134
%RECURSION function 135
%STORAGE function 134
block_spec 200
C compile-time TEST option 13
CEETEST, for C 44
CEETEST, for COBOL 44
CEETEST, for PL/I 44
COBOL compile-time TEST option 17
compile-time PRTEXIT option 23
ctest function 50
cu_spec 201
load_spec 203
PL/I compile-time TEST option 19
PLITEST built-in subroutine 51
run-time TEST option 33
statement_id_range 204
stmt_id_spec 204
%SYSTEM variable
description of 129
for C 145
for COBOL 172
for PL/I 186

T

temporary variables
declaring, for C 139
declaring, for COBOL 165
declaring, for PL/I 180
definition of 398
TERMINATION, AT command, syntax 231
TEST compile-time option
for C 12
for COBOL 16
for PL/I 19
using #pragma statement to specify 16
using for DL/I 115
using for SQL/DS 119
TEST option, compile-time 53
TEST, SET command, syntax 323
TO line, SCROLL command
See SCROLL commands
token, definition of 398
TOP, SCROLL command
See SCROLL commands
TRAP, LE/VSE run-time option 131, 133
TRIGGER command, syntax 331
trigraphs
definition of 398
searching for 94
using with C 195
truncating commands 91, 195

U

unsupported HLL modules, coexistence with 342
unsupported PL/I language elements 354
UP, SCROLL command 93
See also SCROLL commands
USE command, syntax 333
used by Debug Tool
commands file 29
determining userid in filenames 30
log file 30
preferences file 29
profile settings file 29
source and listing files 27
specifying 31
userid, determining default 30
using constants in expressions, for COBOL 173
hex 174
storage 174

utility, definition of 398

V

values
  assigning to C variables 140
  assigning to COBOL variables 164
  assigning to PL/I variables 181
variable value, displaying 56
variables
  accessing program, for C 138
  accessing program, for COBOL 164
  accessing program, for PL/I 178
  assigning values to, for C 140
  assigning values to, for COBOL 164
  assigning values to, for PL/I 181
  compatible attributes in multiple languages 124
  DBCS, assigning new value to 286
  Debug Tool
    detailed descriptions 126
    general description 125
    interpretation of HLL 125
    using in C 140
    using in COBOL 167
    using in PL/I 181
  definition of 398
  description of Debug Tool, for COBOL 167
  displaying, for C 139
  displaying, for COBOL 166
  displaying, for PL/I 178
  qualification of
    for C 156
    for COBOL 174
    for PL/I 189
    general description 130
  removing 240
  session, for PL/I 180
  temporary, declaring
    for C 139
    for COBOL 165
    for PL/I 180

W

warning, for PL/I 187
  generation 188
WARNING, SET command
  See SET WARNING command (C and PL/I)
while command (C), syntax 335
WINDOW commands
  CLOSE, syntax 336
  OPEN, syntax 336
  opening and closing session panel windows with 96
  SIZE, syntax 337
  sizing session panel windows with 96
  syntax 335
WINDOW commands *(continued)*
  ZOOM, syntax 338
window control, Debug Tool
  changing source files 55
  displaying halted location 56
  finding text 54
  scrolling 55
Window id area, Debug Tool 90
Window Layout Selection panel 95
window, error numbers in 57
WINDOW, SET DEFAULT command, syntax 308
windows, Debug Tool 54
windows, Debug Tool session panel
  changing configuration 95
  changing session settings of 94
  closing 96
  Log 87
  Monitor 87
  opening 96
  scrolling 93
  sizing 96
  Source 86
  using 85
  zooming 97
word wrap, definition of 398

Z

ZOOM, WINDOW command
  See WINDOW commands
zooming a window, Debug Tool 97

We'd Like to Hear from You

Debug Tool for VSE/ESA
User's Guide and Reference
Release 1

Publication No. SC26-8797-00

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use this Internet ID:
  - Internet: comments@vnet.ibm.com

  Be sure to include the following with your comments:

  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

Debug Tool for VSE/ESA
User's Guide and Reference
Release 1

Publication No. SC26-8797-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

May we contact you to discuss your comments? Yes No

Would you like to receive our response by E-Mail?

____

Your E-mail address

______________________________ ______________________________
Name Address

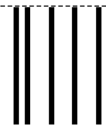
______________________________ ______________________________
Company or Organization

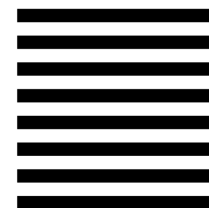
______________________________ ______________________________
Phone No.

Readers' Comments
SC26-8797-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape





BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H3
P.O. Box 49023
San Jose, CA
United States of America 95161-9023

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-8797-00

Spine information:

Debug Tool

User's Guide and Reference

Release 1