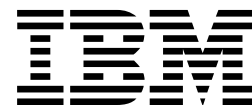




Debug Tool User's Guide and Reference

Release 2



Debug Tool User's Guide and Reference

Release 2

Note!

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

Fifth Edition (March 1998)

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

- Release 4 of OS/390 C/C++ and OS/390 Language Environment (Program Number 5645-001)
- Version 1, Release 2, of IBM COBOL for MVS & VM (Program Number 5688-197), with Version 1, Release 5 of the IBM Language Environment for MVS & VM (Program Number 5688-198),
- Version 2, Release 1 of IBM COBOL for OS/390 & VM (Program Number 5648-A25) with Release 3 of OS/390 Language Environment (Program Number 5645-001)
- Version 1, Release 1, Modification Level 1, of the IBM PL/I for MVS & VM (Program Number 5688-235). with Version 1, Release 4, Modification Level 0, of the IBM Language Environment for MVS & VM (Program Number 5688-198),

and to all subsequent releases and modifications 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 given 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

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. To find out how, see “We'd Like to Hear from You” at the back of this publication.

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, 1998. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xii
Programming interface information	xii
Trademarks and service marks	xii
 About this book	 xiii
What's new	xiii
Who might use this book	xiv
How this book is organized	xiv
How to read the syntax diagrams	xv
 Chapter 1. Before you begin debugging	 1
Debug Tool debugging environments	1
Debug Tool sessions	1
Full-screen session interface	1
Denoting environmental differences	2
Terminology	2
Planning to run your program with Debug Tool	3
 Chapter 2. Preparing to debug your program	 5
Compiling a C program with the compile-time TEST option	5
Using #pragma to specify compile-time TEST option	8
Compiling a C++ program with the compile-time TEST option	9
Placing compiled-in hooks for functions and nested blocks	10
Placing compiled-in hooks for statements and path points	10
Compiling a COBOL program with the compile-time TEST option	10
Compiling a PL/I program with the compile-time TEST option	13
Debugging multilanguage programs	16
Debugging an application fully supported by Language Environment	17
Debugging an application partially supported by Language Environment	17
 Chapter 3. Beginning a debugging session	 18
Using the run-time TEST option	18
Run-time TEST option syntax	19
Run-time TEST option examples	27
Invoking your program when starting a debugging session	28
Invoking Debug Tool under CICS	28
Invoking your program for a debugging session	29
Using alternative Debug Tool invocation methods	31
Invoking Debug Tool with CEETEST	32
Invoking Debug Tool with PLITEST	37
Invoking Debug Tool with the __ctest() function	38
Specifying run-time TEST option with #pragma runopts in C and C++	39
 Chapter 4. Debugging your programs in full-screen mode	 40
Preparing for debugging	40
Invoking your program with Debug Tool	40
Ending a debug session	41
Basic tasks of Debug Tool	41
Debug Tool interface	42
Saving your log file for future use	42

Help	42
Window control	43
Setting a line breakpoint	44
Stepping through or running your program.	44
Using a C program to demonstrate a Debug Tool session	45
C tasks	50
Using a C++ program to demonstrate a Debug Tool session	55
C++ tasks	59
Using a COBOL program to demonstrate a Debug Tool session	66
COBOL tasks	70
Using a PL/I program to demonstrate a Debug Tool session	75
PL/I tasks	79
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
Issuing system commands	90
Using prefix commands	91
Using cursor commands	91
Using Program Function (PF) keys to enter commands	91
Defining PF keys	91
Abbreviating commands	92
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	94
Searching for a character or character string	94
Customizing your session	95
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	98
Customizing settings	99
Getting help during your session	101
Chapter 6. Multiple processes and enclaves	103
Debugging applications within multiple enclaves	103
Invoking Debug Tool within an enclave	103
Using the source window and related windows	103

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	105
Chapter 7. Using Debug Tool in different modes and environments	107
Using Debug Tool in line mode	107
Commands you can use in line mode	107
Getting HELP during a line-mode session	108
Using Debug Tool in batch mode	108
Running multitasking programs with Debug Tool	108
MVS/ESA SP V5R1 with OpenEdition R2 requirement	109
Restrictions when debugging multitasking applications	109
Debugging ISPF applications	109
Debugging DB2 programs	109
Programming considerations	110
Program preparation	110
Precompile requirements	110
Compile requirements	110
Link requirements	111
Bind requirements	111
Using Debug Tool with DB2 programs	111
Batch mode	111
Interactive mode	112
Debugging IMS programs	113
Programming considerations	113
Program preparation	114
Compile requirements	114
Link requirements	114
Using Debug Tool with IMS programs	114
Interactive mode	114
Batch mode	115
Using alternative methods of command input under IMS	115
Debugging CICS programs	116
Debug modes under CICS	116
Invoking Debug Tool under CICS	117
Preparing and using DTCN to invoke Debug Tool under CICS	117
Preparing and using CEEUOPT to invoke Debug Tool under CICS	122
Preparing and using compile-time directives to invoke Debug Tool under CICS	122
Preparing and using CEDF to invoke Debug Tool under CICS	123
Restrictions when debugging under CICS	123
Chapter 8. Debug Tool Support of programming languages	125
Multiple enclaves and interlanguage communication (ILC)	125
Compatible attributes mapped between HLL data types	125
Debug Tool evaluation of HLL expressions	126
Debug Tool interpretation of HLL variables and constants	126
HLL variables	126
HLL constants	126
Debug Tool variables (or intrinsic functions)	127
Modifiable Debug Tool variables	128
Nonmodifiable Debug Tool variables	128

Interpretive subsets	130
Qualifying variables and changing the point of view	131
Qualification	131
Changing the point of view	132
Debug Tool handling of conditions and exceptions	133
Condition handling in Debug Tool	133
Exception handling within expressions (C/C++ and PL/I only)	134
Requesting an attention interrupt during interactive sessions	135
Debug Tool's built-in functions	135
For use with C/C++, COBOL, and PL/I	136
For use with C/C++ and PL/I	136
For use with PL/I	137
Chapter 9. Using Debug Tool with C/C++ programs	138
Debug Tool commands	138
Using C/C++ variables with Debug Tool	138
Accessing program variables	138
Displaying values of C/C++ variables or expressions	139
Declaring temporary variables	139
Assigning values to C/C++ variables	140
Using Debug Tool variables in C/C++	140
C/C++ expressions	145
Using Debug Tool functions with C/C++	148
Debug Tool evaluation of C/C++ expressions	150
Using SET INTERCEPT with C programs	151
Objects and scopes	153
Storage classes	155
Blocks and block identifiers for C	156
Blocks and block identifiers for C++	157
Displaying environmental information	157
Using qualification for C/C++	158
Using qualifiers	160
Changing the point of view	162
Stepping through C++ programs	163
Setting breakpoints in C++	163
AT ENTRY/EXIT	163
AT CALL	164
Examining C++ objects	164
Objects	165
Classes	165
Static data	165
Global data	166
Low-level debugging	166
Chapter 10. Using Debug Tool with COBOL Programs	168
Debugging environment provided for COBOL programs	168
Debug Tool Subset of COBOL commands	168
Restrictions on COBOL-like commands	169
Using COBOL variables with Debug Tool	172
Accessing program variables	172
Assigning values to COBOL variables	172
Declaring temporary variables	174
Displaying values of COBOL variables	174
Using DBCS characters	175

Using Debug Tool variables in COBOL	175
Debug Tool evaluation of COBOL expressions	181
Displaying the results of expression evaluation	181
Using constants in expressions	182
Using Debug Tool functions with COBOL	182
Using %HEX	182
Using the %STORAGE function	182
Using qualification for COBOL	183
Using qualifiers	183
Changing the point of view	185
Chapter 11. Using Debug Tool with PL/I programs	186
Debug Tool Subset of PL/I commands	186
PL/I language statements	186
Using Debug Tool variables in PL/I	187
Conditions and condition handling	188
Freeform input	189
TEST(ERROR, ...)	189
LIST STORAGE	189
Session variables	189
Accessing program variables	189
Structures	190
PL/I expressions	191
PL/I built-in functions	191
Using SET WARNING command with built-ins	192
Unsupported PL/I language elements	192
Positive identification of a compile unit (CU)	192
Chapter 12. Using Debug Tool commands	193
Command modes and language support	193
Entering commands	193
Command format	193
Character set and case	193
Abbreviating keywords	194
Continuation (full-screen and line mode)	195
Significance of blanks	196
Comments	197
Constants	197
Retrieving commands from the log and source windows	198
Online command syntax help	198
Common syntax elements	199
Block_Name	199
Block_Spec	199
Compile_Unit_Name	200
CU_Spec	201
Expression	201
Load_Module_Name	202
Load_Spec	202
References	202
Statement_Id	203
Statement_Id_Range and Stmt_Id_Spec	203
Statement_Label	204
Chapter 13. Debug Tool commands	205

ANALYZE command (PL/I)	205
Assignment command (PL/I)	206
AT command	207
Every_clause	208
AT ALLOCATE (PL/I)	209
AT APPEARANCE	210
AT CALL	212
AT CHANGE	214
AT CURSOR (full-screen mode)	217
AT DATE (COBOL)	218
AT DELETE	218
AT ENTRY/EXIT	219
AT GLOBAL	220
AT LABEL	222
AT LINE	223
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/C++)	233
break command (C/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
CMS command (VM)	244
COMMENT command	245
COMPUTE command (COBOL)	245
CURSOR command (full-screen mode)	246
Declarations (C/C++)	247
C/C++ compatible attributes	249
Declarations (COBOL)	250
DECLARE command (PL/I)	253
PL/I compatible attributes	254
DESCRIBE command	255
DISABLE command	257
DISABLE prefix (full-screen mode)	258
do/while command (C/C++)	259
DO command (PL/I)	259
ENABLE command	262
ENABLE prefix (full-screen mode)	262
EVALUATE command (COBOL)	262
Expression command (C/C++)	264
FIND command	265
for command (C/C++)	266
GO command	267
GOTO command	268
GOTO LABEL command	269
if command (C/C++)	270

IF command (COBOL)	271
IF command (PL/I)	272
IMMEDIATE command (full-screen mode)	273
INPUT command (C/C++ and COBOL)	274
LIST command	274
LIST (blank)	275
LIST AT	275
LIST CALLS	278
LIST CURSOR (full-screen mode)	278
LIST expression	279
LIST FREQUENCY	280
LIST LAST	280
LIST LINE NUMBERS	281
LIST LINES	281
LIST MONITOR	281
LIST NAMES	281
LIST ON (PL/I)	283
LIST PROCEDURES	283
LIST REGISTERS	284
LIST STATEMENT NUMBERS	284
LIST STATEMENTS	285
LIST STORAGE	286
MONITOR command	287
MOVE command (COBOL)	288
Null command	289
ON command (PL/I)	289
PANEL command (full-screen mode)	291
PERFORM command (COBOL)	293
Prefix commands (full-screen mode)	295
PROCEDURE command	295
QUERY command	296
QUERY prefix (full-screen mode)	299
QUIT command	299
RETRIEVE command (full-screen mode)	300
RUN command	301
SCROLL command (full-screen mode)	301
SELECT command (PL/I)	303
SET command	304
SET CHANGE	305
SET COLOR (full-screen and line mode)	306
SET COUNTRY	308
SET DBCS	308
SET DEFAULT LISTINGS (MVS)	309
SET DEFAULT SCROLL (full-screen mode)	309
SET DEFAULT WINDOW (full-screen mode)	310
SET ECHO	310
SET EQUATE	311
SET EXECUTE	312
SET FREQUENCY	312
SET HISTORY	313
SET INTERCEPT (C/C++ and COBOL)	314
SET KEYS (full-screen and line mode)	315
SET LOG	315
SET LOG NUMBERS (full-screen and line mode)	316

SET MONITOR NUMBERS (full-screen and line mode)	316
SET MSGID	317
SET NATIONAL LANGUAGE	317
SET PACE	318
SET PFKEY	318
SET PROGRAMMING LANGUAGE	319
SET PROMPT (full-screen and line mode)	320
SET QUALIFY	321
SET REFRESH (full-screen mode)	322
SET REWRITE	323
SET SCREEN (full-screen and line mode)	323
SET SCROLL DISPLAY (full-screen mode)	324
SET SOURCE	324
SET SUFFIX (full-screen mode)	325
SET TEST	326
SET WARNING (C/C++ and PL/I)	327
SET command (COBOL)	328
SHOW Prefix command (full-screen mode)	329
STEP command	329
switch command (C/C++)	331
SYSTEM command	333
TRIGGER command	334
TSO command (MVS)	336
USE command	336
while command (C/C++)	338
WINDOW command (full-screen mode)	339
WINDOW CLOSE	339
WINDOW OPEN	340
WINDOW SIZE	340
WINDOW ZOOM	341
Appendix A. Coexistence	343
Coexistence with other debuggers	343
Coexistence with unsupported HLL modules	343
Appendix B. Using Debug Tool in a production mode	344
Fine-tuning your programs with Debug Tool	344
Removing hooks, statement tables, and symbol tables	344
Using Debug Tool on optimized programs	345
Appendix C. Using C/C++ Reference Information with Debug Tool	347
C reserved keywords	347
Operators and operands	347
Language Environment conditions and their C/C++ equivalents	348
Appendix D. Using COBOL Reference Information with Debug Tool	350
COBOL listing files	350
Debug Tool interpretive subset of COBOL commands	350
COBOL reserved keywords	350
Allowable comparisons for the Debug Tool IF command	350
Allowable moves for the Debug Tool MOVE command	352
Allowable moves for the Debug Tool SET command	354
Appendix E. Debug Tool Messages	355

Symbols in messages 355

Bibliography 422

High level language publications 422

Related publications 422

Softcopy publications 423

Glossary 424

Index 430

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming interface information

This book is intended to help you debug application programs. This book documents General-use Programming Interface and Associated Guidance Information provided by Debug Tool.

General-use Programming Interfaces allow you to write programs that obtain the services of Debug Tool.

Trademarks and service marks

The following terms, used in this publication, are trademarks or service marks of International Business Machines Corporation in the United States or other countries:

AD/Cycle	MVS
BookManager	MVS/ESA
C/370	Open Class
C/C++ for MVS/ESA	OpenEdition
CICS	OS/2
CICS/ESA	OS/390
COBOL/370	Systems Application Architecture
DATABASE 2	SAA
DB2	SQL/DS
DRDA	System/370
FFST/2	System/390
IBM	VisualAge
IMS	VM/ESA
IMS/ESA	Virtual Machine/
Language Environment	Enterprise Systems Architecture

About this book

Debug Tool combines the richness of the System/370 and System/390 subsystem environments with the power of Language Environment to provide a debugger for programmers to isolate and fix their program bugs and test their applications. Debug Tool gives you the capability of testing programs in batch or using a nonprogrammable terminal in full-screen or line mode to debug your programs interactively.

This book contains instructions and examples to help you use the Debug Tool to debug C, C++, COBOL, and PL/I applications running with Language Environment. 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).

Note: When MVS is used in this book, it refers to both MVS and OS/390 systems.

What's new

The following is a list of enhancements that have been added to Release 2 of IBM Debug Tool:

- Support for COBOL Millenium Language Extensions.
- Scenarios in each language to help you get started using Debug Tool to debug your applications.
- Support for VisualAge COBOL remote debugging for host applications.
- Enhancements to DTCN to allow easier Debug Tool access by users.
- Support in MVS, CICS, and OS/390 OpenEdition services for source level debugging of header file functions which allows you to view the source of a function residing in the header file while it runs.
- Support in MVS for debugging POSIX applications including:
 - Applications whose source code resides in HFS
 - Applications that use POSIX multithreading
 - Applications that use fork/exec, but Debug Tool can be active in one process only
 - Applications that use asynchronous signals, as long as they are handled by the Language Environment condition manager

You **cannot** debug applications that run under the OpenEdition shell. The POSIX applications must run under either TSO or MVS batch and must reside in a PDS or PDSE load module. You cannot run POSIX applications that reside in an HFS executable module.

Release 2 of Debug Tool contains the following enhancements:

- Full debugging support of COBOL applications exploiting the new Object Oriented enhancements, and the improved interoperability with the C language. The new debugging support includes:
 - debugging of classes and methods
 - support for long program names, and mixed case program names
 - support for null terminated strings (Z literals)
- Debug Tool CICS Interactive Run-Time Facility (DTCN). The DTCN panel allows CICS developers to:
 - Dynamically modify the Language Environment run-time options to use Debug Tool
 - Establish the scope of your debugging session in terms of region, transaction, and terminal
 - Debug your application either interactively or in batch
- Interactive online debugging of CICS batch transactions.

Who might use this book

This book is intended for application programmers using Debug Tool to debug HLLs with Language Environment. Throughout this book, these languages are referred to as C/C++, COBOL, and PL/I.

The following operating systems and subsystems are supported:

- OS/390 and MVS
 - TSO
 - CICS
 - JES/Batch
 - IMS
 - DB2
- VM
 - 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

The first four chapters of this book discuss the preparatory work you must complete before using Debug Tool and provides sample scenarios for each supported language to help you begin using Debug Tool. The scenarios include helpful hints when performing some basic debugging tasks. The next two chapters discuss how to customize Debug Tool for your particular environment and gives you information about using Debug Tool in a variety of environments, including MVS/JES batch mode, line mode, Customer Information Control System (CICS), Information Management System (IMS), DATABASE 2 (DB2), and debugging applications that contain Structured Query Language/Data System (SQL/DS) statements.

Chapters 8-11 provide information about Debug Tool's interaction with different programming languages. Debug Tool variables, functions, and expression evaluation are explained.

Chapter 12 contains all the Debug Tool commands, shows their syntax, and provides examples of their use.

The appendixes include the following information:

- Discussion of the coexistence of Debug Tool with HLL modules compiled with previous versions of compilers
- Information on how to optimize your programs while still retaining some debugging capability
- Reference information for C and COBOL that include reserved keywords and Debug Tool interpretive subsets of HLL commands
- Complete list of Debug Tool messages.

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

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).
- A `b` symbol indicates one blank position.

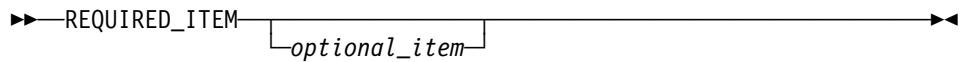
Required items

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

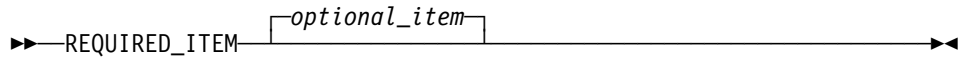
▶—REQUIRED_ITEM————▶▶

Optional Items

Optional items appear below the main path.

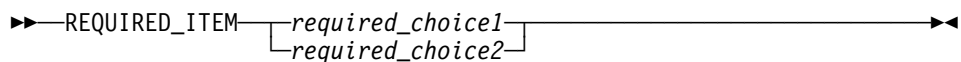


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

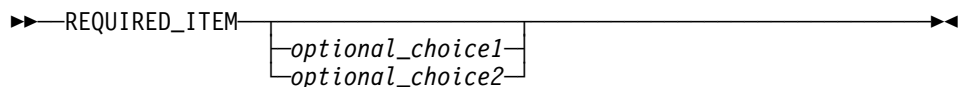


Multiple required or optional items

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.

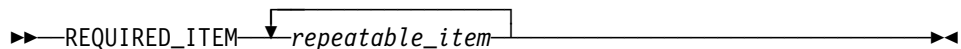


If choosing one of the items is optional, the entire stack appears below the main path.

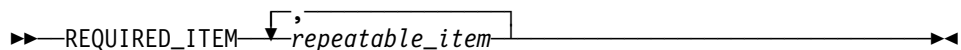


Repeatable items

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



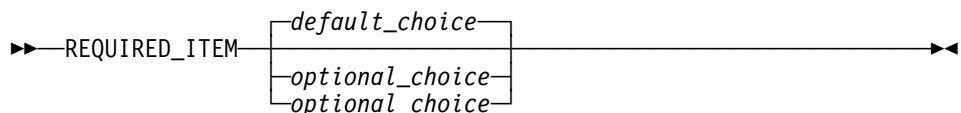
If the repeat arrow contains a comma, you must separate repeated items with a comma.



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.



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/C++, COBOL, or PL/I on OS/390, MVS or VM. Debuggable applications can include other languages, but Debug Tool does not debug those portions of your application. In this book, MVS refers to both MVS and OS/390.

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

Debug Tool debugging environments

Debug Tool provides several debugging environments. The number of platforms 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 systems.

The terms *full-screen mode*, *line 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, as well as a discussion of the platforms supported by Debug Tool.

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. |
| Line-Mode Session | Enter Debug Tool commands on the command line and receive debugging information, one line at a time, while you are programming. |
| Batch-Mode Session | Debug Tool command files provide a mechanism to predefine 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 a batch application. |

Full-screen and line-mode sessions are both interactive types of sessions.

Full-screen session interface

Debug Tool provides:

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

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

Before you begin debugging

Figure 1 on page 2 shows the three windows of Debug Tool.

```
COBOL      LOCATION: MULTCU :> 75.1
Command ==>
MONITOR ---+---1---+---2---+---3---+---4---+---5---+---6 LINE: 1 OF 2
***** TOP OF MONITOR *****
0001 1 01 MULTCU:>PROGRAM-USHORT-BIN 00000
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      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
0010 LIST PROGRAM-SSHORT-BIN ;
0011 AT 75 ;
0012 AT 77 ;
0013 AT 79 ;
0014 GO ;
```

Figure 1. 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, not only across platforms but from system to system and from subsystem to subsystem. When this occurs, differences are marked in the text in the following manner:

For MVS Only: MVS-specific information.

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, as well as differing terminology between platforms, a group of common terms has been established. Table 1 on page 3 lists these terms and their equivalency in each language.

Table 1. Terminology

Debug Tool Term	C/C++ Equivalent	COBOL Equivalent	PL/I Equivalent
Compile Unit	C/C++ source file	Program or Class	Program
Block	Function or Compound Statement	Program, Nested Program, Method 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 allows you to set breakpoints that instruct Debug Tool to gain control at selected points during program run time.

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 and accessing symbol tables, see Chapter 2, “Preparing to debug your program” on page 5.

- *Do you want 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 placing hooks and accessing symbol tables, see Chapter 2, “Preparing to debug your program” on page 5.

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

Removing hooks, statement tables, or symbol tables can increase your application's performance and/or decrease its size. See Appendix B, “Using Debug Tool in a production mode” on page 344 for a complete discussion.

- *Do you need to reduce data set I/O?*

The source (for C/C++) and the source listing (for COBOL and PL/I) data sets should be defined with a suitable block size to minimize data set I/O when using Debug Tool. Use a block size that is as large as possible in your environment.

- *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, as well as many options for allowing it to gain control of your test program.

Before you begin debugging

To invoke Debug Tool, you can use the run-time TEST option. This option gives you the choice of invoking Debug Tool either before you run your application, at the occurrence of an HLL condition while your application is running, or at the occurrence of an attention interrupt. Also, Language Environment, as well as certain HLLs, provides a run-time service you can call while your program is executing, at the location of your choice.

After Debug Tool is invoked, it gains control of your program and suspends 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, in line mode, or in batch mode?*

Refer to “Debug Tool sessions” on page 1 to determine how you want to use Debug Tool when debugging your application.

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 your C program with the compile-time TEST option
- Compiling your C++ program with the compile-time TEST option
- Compiling your COBOL program with the compile-time TEST option
- Compiling your PL/I program with the compile-time TEST option
- Debugging multilanguage programs

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, as described below. This option causes the compiler to retain information about your application program that Debug Tool uses.

The TEST suboptions BLOCK, LINE, and PATH regulate 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 regulates the inclusion of symbol tables into the object output of the compiler. Debug Tool uses the symbol tables to obtain information about the variables in the program.

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

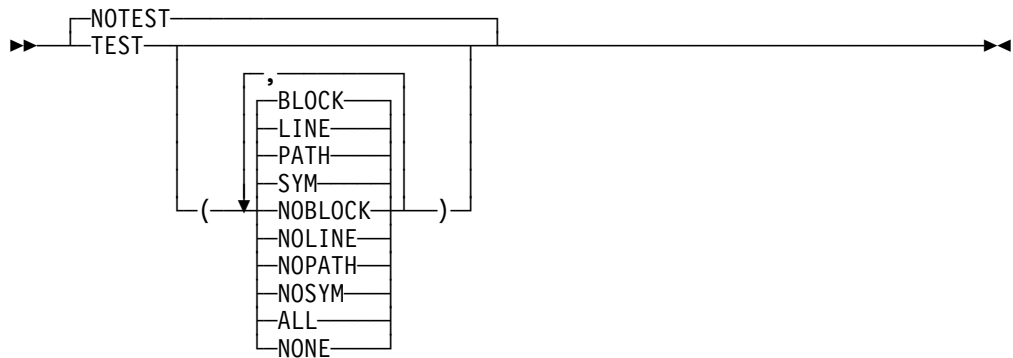
- The C compile-time TEST option generates entry and exit hooks for functions.
- The C compile-time TEST option implicitly specifies the GONUMBER option, which causes the compiler to generate line number tables corresponding to the input source file. You can explicitly remove this option by specifying NOGONUMBER.
- Programs compiled with both the TEST and either OPT(1) or OPT(2) 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 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 *OS/390 C/C++ User's Guide* for more information on the compile-time TEST option.

Compiling a C program with TEST

You can specify any combination of the C TEST suboptions in any order. The default suboptions are BLOCK, LINE, PATH, 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 programs. 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, a Language Environment 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

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 is 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 your program or directly in your program, using a #pragma.

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

If you link together two or more compile units with differing #pragmas, the options specified with the first compile unit are honored. With multiple enclaves, the options specified with the first enclave (or compile unit) invoked *in each new process* are honored.

If you specify options on the command line and in a #pragma, any options entered on the command line override those specified in the #pragma unless you specify NOEXECOPS. Specifying NOEXECOPS, either in a #pragma or with the compile-time EXECOPS option, prevents any command line options from taking effect.

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

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

This is equivalent to TEST(SYM,BLOCK,LINE,PATH). The default PATH means that the PATH breakpoint will be triggered for the 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 and C++” on page 39.

For more information about #pragma options, refer to *OS/390 C/C++ Language Reference*.

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, as described below. This option causes the compiler to retain information about your application program that Debug Tool uses. See “Compiling a C program with the compile-time TEST option” on page 5 for an explanation of the debug information generated by TEST(ALL).



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

NOTEST

Specifies that no debugging information is to be generated. That is, no 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.

TEST

Produces debugging information for Debug Tool to use during batch and interactive debugging. The following restrictions apply when using the TEST option

- The maximum number of lines in a single source file cannot exceed 131,072.
- The maximum number of include files which have executable statements cannot exceed 1024.

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

Placing compiled-in hooks for functions and nested blocks

The following rules apply to the placement of compiled-in entry and exit hooks for functions and nested blocks:

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

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.

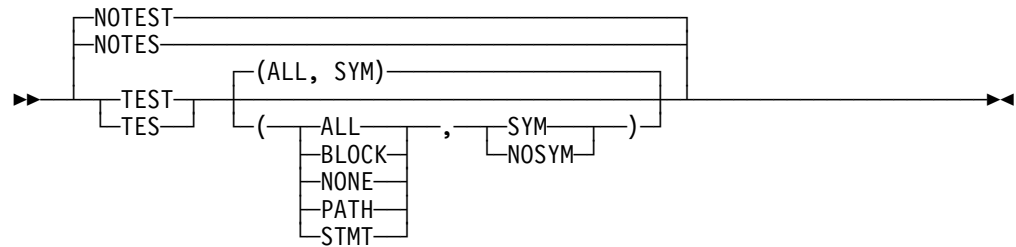
Compiling a COBOL program with the compile-time TEST option

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 344 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 compile-time 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:



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, 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. Using NOTEST produces the following results:

- You cannot STEP through program statements.
- You can suspend execution of the program only at the initialization of the main compile unit.
- You can include calls to CEETEST in your program to allow you to suspend program execution and issue Debug Tool commands.
- You cannot examine or use any program variables.
- You can LIST storage and registers.
- The source listing produced by the compiler cannot be used; therefore, no listing is available during a 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, date processing, and program entry and exit hooks.

- The COBOL compiler only generates compiled-in hooks for date processing statements when either the DATEPROC(FLAG) or DATEPROC(NOFLAG) compile-time option is specified. A date processing statement is any statement that references a date field, or any EVALUATE or SEARCH statement WHEN phrase that references a date field.
- You can set breakpoints at all statements and path points, and STEP through your program.

- Debug Tool can gain control of the program at all statements, path points, date processing statements, labels, and block entry and exit points, allowing you to enter Debug Tool commands.
- Branching to statements and labels using the Debug Tool command GOTO is allowed.

BLOCK

Hooks are inserted at all block entry and exit points.

- Debug Tool gains control at entry and exit of your program, methods, nested programs, and PERFORM group of statements.
- Debug Tool can 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 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 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, at every date processing statement, and at all entry and exit points.

- The COBOL compiler only generates compiled-in hooks for date processing statements when either the DATEPROC(FLAG) or DATEPROC(NOFLAG) compile-time option is specified. A date processing statement is any statement that references a date field, or any EVALUATE or SEARCH statement WHEN phrase that references a date field.
- You can set breakpoints at all statements and STEP through your program.
- Debug Tool cannot gain control at path points unless they are also at statement boundaries.
- Branching to all statements and labels using the Debug Tool command GOTO is allowed.

SYM

Generates 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. Using NOSYM produces the following results:

- You cannot reference program variables by name.
- You cannot use commands such as LIST a variable or expression containing a variable, or DESCRIBE a variable name.
- You cannot use commands such as CALL variable to branch to another program, or GOTO to branch to another label (paragraph or section name).

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

See the COBOL Language Reference publications 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.

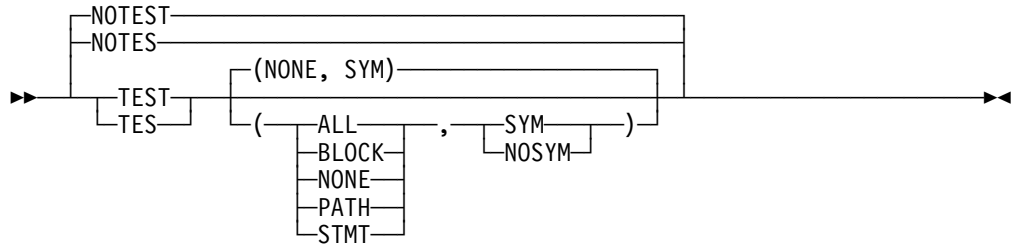
During a debugging session, Debug Tool displays the first file it finds named `userid.pgmname.list` in the Source window. Use the SET SOURCE command to associate your source listing with the program you are debugging. See "SET SOURCE" on page 324 as well as "SET DEFAULT LISTINGS (MVS)" on page 309 (for partitioned data sets).

Compiling a PL/I program with the compile-time TEST option

The PL/I compiler provides support for Debug Tool under control of the compile-time TEST option and its suboptions for hook locations and symbol tables. The hook location suboptions (BLOCK, STMT, PATH, ALL, and NONE) regulates 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 suboption (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:

Compiling a PL/I program with TEST



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, 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 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. Using NOTEST produces the following results:

- You can LIST storage and registers.
- You can include calls to PLITEST or CEETEST in your program so you can suspend running your program and issue Debug Tool commands.
- You cannot STEP through program statements. You can suspend running your program only at the initialization of the main compile unit.
- You cannot examine or use any program variables.
- Because 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 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.

NONES

No hooks are inserted in the program.

- A call to PLITEST or CEETEST can 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 which causes the following results:

- You cannot reference program variables by name.
- You cannot use commands such as LIST a variable or expression containing a variable, or DESCRIBE a variable name.
- You cannot use commands such as CALL variable to branch to another program, or GOTO to branch to another label (procedure or block name).

See the *PL/I for MVS and VM 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, PL/I programs must be compiled using the PL/I compile-time SOURCE option. You must also direct the listing to a nontemporary file that is available during the debugging session.

During a debugging session, Debug Tool displays the first file it finds named `userid.pgmname.list` in the Source window. Use the SET SOURCE command to associate your source listing with the program you are debugging. See “SET SOURCE” on page 324, as well as “SET DEFAULT LISTINGS (MVS)” on page 309 (for partitioned data sets).

Compiling with TEST(STMT), TEST(PATH), or TEST(ALL) 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.

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 Language Environment. Language Environment 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 Language Environment and can be debugged by Debug Tool.
- You need to debug an application written in more than one language, where not all of the languages are supported by Language Environment, nor can they be debugged by Debug Tool.

When writing a multilanguage application, a number of special considerations arise because you must work outside the scope of any single language. The Language Environment initialization process establishes an environment tailored to the set of HLLs constituting the main load module of your application program. This removes the need to make explicit calls to manipulate the environment. Also, termination of

the Language Environment environment is accomplished in an orderly fashion, regardless of the mixture of HLLs present in the application.

Debugging an application fully supported by Language Environment

If you are debugging a program written in a combination of languages supported by Language Environment and compiled by supported compilers, very little is required in the way of special actions. Debug Tool normally recognizes a change in programming languages and automatically switches to the correct language when a breakpoint is reached. If desired, you can use the `SET PROGRAMMING LANGUAGE` command to stay in the language you specify; however, you can only access variables defined in the currently set programming language. For details, see “`SET PROGRAMMING LANGUAGE`” on page 319.

When defining session variables you want to access from compile units of different languages, you must define them with compatible attributes. See “C/C++ compatible attributes” on page 249, “COBOL compatible attributes” on page 252, or “PL/I compatible attributes” on page 254 for a table showing compatible attributes for variables declared in the supported languages.

For more information on creating multilanguage applications, see the *OS/390 Language Environment Programming Guide*

Debugging an application partially supported by Language Environment

Sometimes you might find yourself debugging applications that contain compile units written in languages not supported by either Debug Tool or Language Environment. For example, you can run programs containing mixtures of Assembler, C/C++, COBOL, FORTRAN, and PL/I source code with 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 a Language Environment-enabled compiler, or relink-edited to take advantage of Language Environment 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 determines 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 once again gains control and execute commands.

Chapter 3. Beginning a debugging session

This chapter explains how to begin a debugging session with Debug Tool. It covers the run-time TEST option, which gives you several alternatives for beginning a debugging session when specified during the invocation of your program.

Also covered are Language Environment callable services CEETEST and PLITEST, and the C library function `__ctest()`. These can be inserted into your program to govern the invocation of Debug Tool. The use of `#pragma runopts` to specify the run-time TEST option in C programs is discussed in more detail.

For MVS Only: If your source or listing does not come up in Debug Tool when you start it, press PF4 (LIST) with the cursor on the command line. This puts you in the Source Identification panel. The Source Identification panel indicates the name of the source or listing file that was intended to be used by Debug Tool. With this name you can verify if the file exists or if you have authorization to access it. If your file is stored at a different place, use the SET SOURCE command or type over the Listing/Source file field with the new name to have Debug Tool search for the source or listing there. The SET DEFAULT LISTINGS command provides another method of finding your files provided they are stored to a PDS.

For C/C++ compile units, Debug Tool requires a file containing the source code. By default, when Debug Tool encounters a new C/C++ compile unit, it looks for the source code in a file whose name is the one that was used on the compile step. For COBOL and PL/I compile units, Debug Tool requires a file containing the compiler listing. By default, when Debug Tool encounters a new VS COBOL II or PL/I compile unit, it looks for the listing in a file named *hlq.cuname.LIST*. For COBOL/370, COBOL for MVS, and COBOL for OS/390, Debug Tool looks for the listing in a partitioned data set member named *cuname*.

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 GOT0 or GO BYPASS command. You can even specify that control be given to another program with the CALL command or a C/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 267).

Using the run-time TEST option

You can use the run-time TEST option to invoke Debug Tool and begin testing your program. The simplest form of the TEST option is TEST with no suboption; however, suboptions provide you with more flexibility. There are four suboptions available:

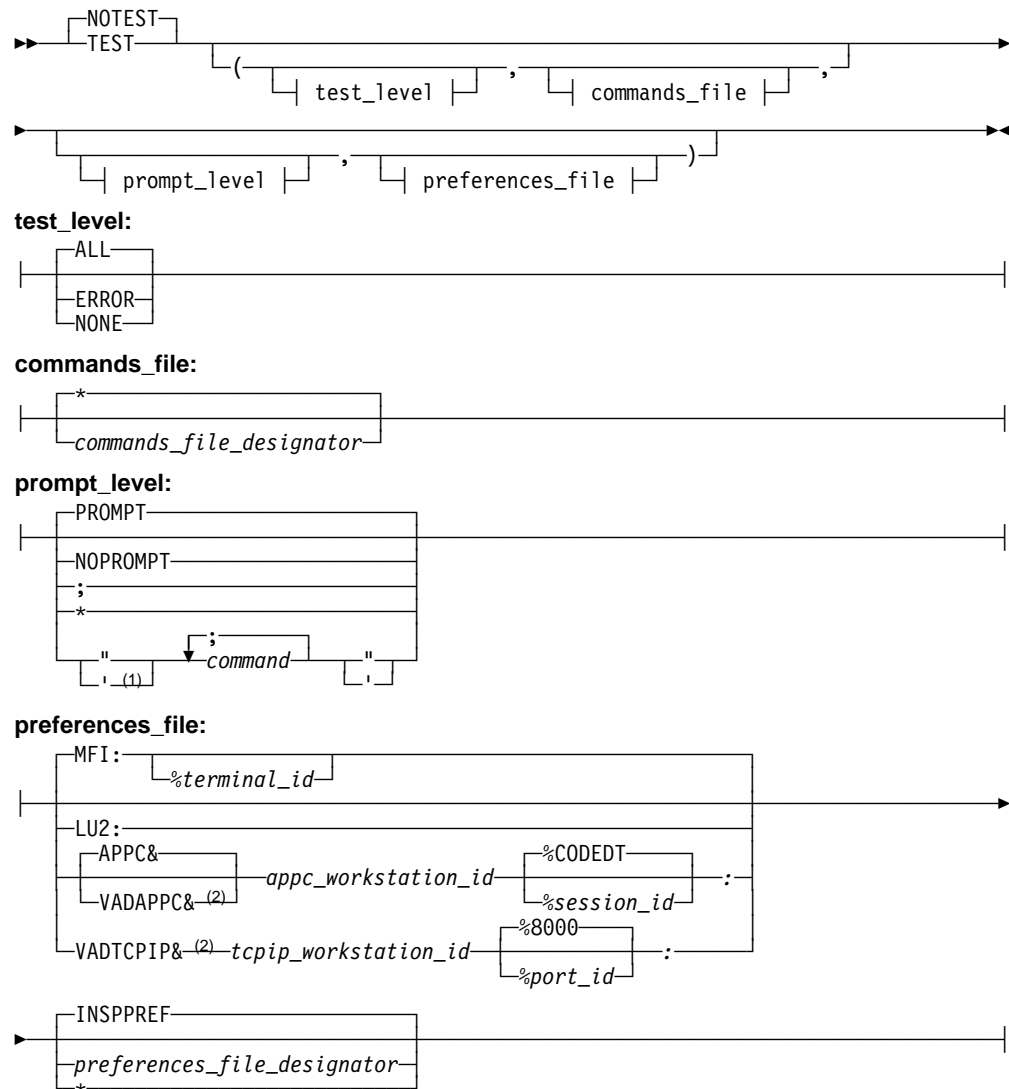
- `test_level` (determines what HLL conditions raised by your program will cause Debug Tool to gain control)

- `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 or workstation)
- `prompt_level` (determines whether an initial commands list is unconditionally executed during program initialization)
- `preferences_file` (specifies the session parameter and a file that you can use to specify default settings for your debugging environment, such as customizing the settings on the Debug Tool Profile panel)

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. 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 page 27.

The syntax for this option is:



Notes:

- ¹ Double quotes for MVS; single quotes for VM.
- ² Supports only VisualAge COBOL programs.

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 was 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 Language

Environment 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 135.

ERROR

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

- For C/C++:
 - An attention interrupt
 - Program termination
 - A predefined Language Environment condition of Severity 2 or above
 - Any C/C++ condition other than SIGUSR1, SIGUSR2, SIGINT or SIGTERM.
- For COBOL:
 - An attention interrupt
 - Program termination
 - A predefined Language Environment condition of Severity 2 or above.
- For PL/I:
 - An attention interrupt, directed at either PL/I or Debug Tool
 - Program termination
 - A predefined Language Environment condition of Severity 2 or above.

Language Environment conditions are described in the *OS/390 Language Environment 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 exists 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 326.

commands_file:

* (or *blank*)

Indicates that no commands file is supplied. The terminal, if available, is used as the source of Debug Tool commands.

commands_file_designator

Valid designation (ddname or data set for MVS, or filedef or file id for CMS) for the primary commands file which is used instead of the terminal as initial source of commands after the preferences file finishes running. If the designator might cause an ambiguity in the list of suboptions, enclose it in single or double quotation marks to differentiate it from the remainder of the list. If you are using a single ddname, no quotation marks are required.

The *commands_file_designator* has a maximum length of 80 characters.

If the specified ddname is longer than eight characters, it is automatically truncated, but no error message is issued.

When the end of the file is reached, Debug Tool interactively prompts you for commands until a QUIT command or the end of your application is reached.

The use of a primary commands file is required when debugging batch programs with a noninteracting interface, and this suboption enables you to specify a source of commands when using Debug Tool in batch mode. It also allows you to use a log file from one Debug Tool session as a source of commands in a subsequent Debug Tool session to regression test your application.

When not using an interactive interface (for example, VisualAge COBOL workstation), the primary commands file is required for batch debugging sessions. It acts as a surrogate terminal. Debug Tool reads and executes commands from it 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 looks to your terminal, if available, for commands. If your terminal is not available (if you are debugging in batch, for example), Debug Tool forces a G0 until the end of your program is reached.

Note: VisualAge COBOL does not support use of a commands file.

prompt_level:

PROMPT (or ; or *blank*)

Indicates that you want Debug Tool invoked immediately after Language Environment initialization. Commands are read from the preferences file and then any designated primary commands file. If neither file exists, commands are read from your terminal or workstation.

NOPROMPT (or *)

Indicates that you do not want Debug Tool invoked immediately after Language Environment initialization. Instead, your application begins running.

command

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 should be enclosed in double quotation marks (MVS) or single quotation marks (VM). Multiple commands must be separated by a semicolon.

Note: If you include a STEP or G0 in your command string, none of the subsequent commands are processed. The command string operates like a commands file. The VisualAge COBOL workstation interface does not support commands file.

preferences_file:**MFI**

Specifies Debug Tool should be invoked in MFI mode, that is, you are using a 3270-type terminal for your debugging sessions.

terminal_id (for CICS only)

Specifies up to a four-character-length terminal id which receives Debug Tool screen output during dual terminal session. The corresponding terminal should be in service and acquired ready to receive Debug Tool-related I/O.

INSPREF (or *blank*)

Debug Tool-supplied default preferences file ddname. Any preferences file that is specified to Debug Tool becomes the first source of Debug Tool commands after the debugger is invoked. It is often used to set up the Debug Tool environment.

preferences_file_designator

Valid designation (ddname or data set for MVS, or filedef or file id for CMS) specifying the preferences file to be used.

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

- * Specifies that no preferences file is supplied.

Note: INSPREF and preferences_file_designator are not supported when using the VisualAge COBOL workstation interface. * is always assumed.

For Workstation Debugging Only:

Workstation debugging provides the advantage of a GUI interface between the workstation and the host-based Debug Tool. It also provides important additional function such as the ability to interactively debug batch processes. For example, a COBOL batch job running in MVS/JES, or a COBOL CICS batch transaction, can be interactively debugged via a TCP/IP connection to a workstation equipped with VisualAge COBOL.

Currently there are two workstation products that interface with Debug Tool: VisualAge COBOL and CODE/370. When you want to debug your host applications from your workstation, use one of the following protocols to communicate with the host:

- If you have VisualAge COBOL installed on your OS/2 workstation, you can use either APPC or TCP/IP to communicate with the host.
- If you have VisualAge COBOL installed on your Windows workstation, use TCP/IP to communicate with the host.
- If you have CODE/370 installed on your OS/2 workstation, you can use either APPC or LU2 to communicate with the host.

When this type of debugging is performed, the host application invokes Debug Tool, which in turn invokes the workstation interface that you've designated in one of the suboptions. The following suboptions are qualified as to which workstation product they apply.

LU2

Specifies you want to establish a Debug Tool session with a CODE/370 workstation using an LU2 session to provide a GUI access.

APPC&

Specifies you want to establish a Debug Tool session with a workstation that has been set up for APPC communications with the host. This suboption applies only to workstations equipped with CODE/370 and configured for APPC communications.

VADAPPC&

Specifies that Debug Tool is interfacing with an OS/2 workstation equipped with VisualAge COBOL and configured for APPC communications with the host. This suboption is valid only when you have installed and are using VisualAge COBOL on your OS/2 workstation.

appc_workstation_id

A 1-to-8 character alphanumeric name defining your workstation at APPC configuration time. This is the APPC name of the workstation which will display your debug information. An example of this symbolic destination name would be AJSMITH or DEPT87. If you do not define *appc_workstation_id* properly when APPC is configured and your application is running in batch (for example, JES), Debug Tool is not initiated. The batch program continues to run or terminates, depending on its state when the debug session is attempted. If *appc_workstation_id* is improperly defined and your application is running in the TSO foreground, or in CICS when the task has a terminal associated with it, an MFI session is created. This behavior is consistent for APPC sessions attempted with workstations equipped with either VisualAge COBOL or CODE/370.

%CODEDT

Default *session_id* for a CODE/370 workstation.

%*session_id*

Specifies a unique name of the application you want to debug. If you identify your session with the same *session_id* as that of an existing session, an initialization failure for the session being started will occur.

VADTCPIP&

Specifies that Debug Tool is interfacing with either an OS/2 or a Windows NT workstation equipped with VisualAge COBOL and configured for TCP/IP communications with the host. This suboption is valid only when you have installed and are using VisualAge COBOL on your workstation.

tcpip_workstation_id

TCP/IP name of the workstation which will display your debug information.

%8000

Default *port_id*.

%*port_id*

Specifies a unique TCP/IP port on your workstation which is used by the daemon program.

VisualAge COBOL only

If you are using the VADTCP/IP& suboption, consider the following possible errors:

- The `tcpip_workstation_id` or `port_id` parameters must be syntactically or functionally correct. If they are not and you attempt an interactive session, an MFI session will be allocated. For example, if you attempt a session from TSO or CICS with incorrect parameters, you will receive an MFI session at your host window. This error is noted in the MVS SDSF log as an *allocation* failure.
- If the `tcpip_workstation_id` or `port_id` parameters are not syntactically or functionally correct, and you attempt an interactive batch session with Debug Tool, Debug Tool will terminate and the batch application will continue to run as though no debug session was ever attempted. This error occurs when, for example, you run a JES batch job or CICS batch transaction. If the parameters are incorrect, your program will continue to run as if you never attempted to initialize Debug Tool. This error is noted in the MVS SDSF log as an *allocation* failure.
- For TCP/IP sessions, the daemon must be started at the workstation before you initialize Debug Tool. VisualAge COBOL documentation contains information on using the daemon program.

_____ End of VisualAge COBOL only _____

End of Workstation Parameters.

Other run-time TEST option considerations

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

- The Language Environment run-time options have the following order of precedence (from highest to lowest):
 1. Installation options in the CEEDOPT file that were specified as nonoverrideable with the NONOVR attribute.
 2. Options specified by the Language Environment assembler user exit. Debug Tool uses the DTCN transaction in the CICS environment and customized Language Environment 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 117.
 3. Options specified at the invocation of your application, using the run-time TEST option, unless accepting run-time options is disabled by Language Environment (EXECOPS/NOEXECOPS).
 4. Options specified within the source program (with `#pragma` or `PLIXOPT`) or application options specified with CEEUOPT and link-edited with your application.¹
 5. Option defaults specified at installation in CEEDOPT.
 6. IBM-supplied defaults.

Suboptions are processed in the following order:

¹ If the object module for the source program is input to the linkage editor before the CEEUOPT object module, *then* these options override CEEUOPT defaults. You can force the order in which objects modules are input by using linkage editor control statements.

1. Commands entered at the command line override any defaults or suboptions specified at run time.
 2. Commands executed from a preferences file override the command string and any defaults or suboptions specified at run time.
 3. Commands from a commands file override default suboptions, suboptions specified at run time, commands in a command string, and commands in a preferences file.
- In C, C++ or PL/I, you can define TEST with suboptions using a `#pragma runopts` or `PLIXOPT` string, then specifying TEST with no suboptions at run time. This causes the suboptions specified in the `#pragma runopts` or `PLIXOPT` string to take effect.

- Some suboptions are disabled with `NOTEST`, but are still allowed. This means you can start your program using the `NOTEST` option and then specify suboptions you might want to take effect later in your debugging session. The program begins to run without Debug Tool taking control.

To enable the suboptions you specified with `NOTEST`, invoke Debug Tool during your program's run time 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 action is assumed. This occurs even though you have not previously defined a breakpoint for that condition or location using an initial command string or a primary commands file. Control is given to your terminal or to your primary commands file.
- The primary commands file acts as a surrogate terminal. Once it 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 the USE file in that, if a USE file contains a command that returns control to the program (such as `STEP` or `G0`), all subsequent commands are discarded. However, USE files invoked from within a primary commands file take on the characteristics of the primary commands file and can be executed until complete.
- In batch mode, when end-of-file is reached in your commands file, a `G0` 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 Language Environment initialization will complete and give you access to all variables. You can also enter all valid commands.

- If Debug Tool is invoked while your program is running (for example, using a `CEETEST` call), it might not be able to find all compile units associated with your application. Compile units located in load modules that are not currently active are not known to Debug Tool, even if they were run prior to Debug Tool's initialization.

Debug Tool also does not know about compile units that were not compiled with the compile-time TEST option, even if they are active, nor does it know about compile units written in unsupported languages.

For example, suppose load module mod1 contains compile units cu1 and cu2, both compiled with the TEST option. The compile unit cu1 calls cux, contained in load module mod2, which returns after it completes processing. The compile unit cu2 contains a call to the CEETEST library service. When the call to CEETEST initializes Debug Tool, only cu1 and cu2 are known to it. Debug Tool does not recognize cux.

- The results of the execution of the initial commands list or 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 command list, whether it consists of a command string included in the run-time options or a primary commands file, can contain a USE command to get commands from a secondary file. If invoked from the primary commands file, a USE file takes on the characteristics of the primary commands file. See “USE command” on page 336 for details.
- The initial command string is performed only once, when Debug Tool is first initialized in the process.
- Commands in the preferences file are performed only once, when Debug Tool is first initialized in the process.
- You can change the run-time TEST/NOTEST options at any time with the SET TEST command. See “SET TEST” on page 326.
- The primary commands file is shared across multiple enclaves.

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 29, and *OS/390 Language Environment Programming Guide*.

- 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 and C++ allow default suboptions to be selected at compile time using `#pragma runopts`. Similarly, PL/I offers the PLIXOPT string. Language Environment provides the macro CEEXOPT. Using this macro, you can specify installation and program-specific defaults. For more information on using CEEXOPT, see *OS/390 Language Environment Programming Guide*.

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

- TEST(ALL,*,*,*)

Invoking your program when starting a session

Debug Tool is not invoked initially; however, any condition or an attention in your program causes Debug Tool to be invoked, as does a call to `CEESTEST`, `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 `CEESTEST`, `PLITEST`, or `__ctest()`.

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

Debug Tool is invoked at the end of environment initialization, but before the main program prolog has completed. The `ddname prefer` is processed as the preferences file, and subsequent commands are found in data set `test.scenario`. If all commands in the commands file are processed and you issue a `STEP` command when prompted, or a `STEP` command is 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 `test.scenario` repeating this process until end-of-file is reached. At this point, commands are obtained from your terminal.

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

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

- If you are working from a cooperative environment, that is, you are debugging your host application from your workstation, the following examples apply:

```
TEST(,,,LU2:*)           /* Using LU2 suboption      */
TEST(,,,OSCAR:*)          /* Using APPC suboption     */
TEST(,,,APPC&OSCAR:*)     /* Using APPC suboption     */
TEST(,,,VADAPPC&OSCAR:*)  /* Using VADAPPC suboption  */
TEST(,,,VADTCPIP&ERNIE:*) /* Using VADTCPIP suboption */
TEST(,,,VADTCPIP&machine.somewhere.something.com:*)
TEST(,,,VADTCPIP&9.24.104.79:*)
```

where `OSCAR` and `ERNIE` are `workstation_ids`.

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 proper 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" on page 29).

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/ESA, Language Environment, and Debug Tool. See "Debugging CICS programs" on page 116 and the appropriate language 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, pragma, or CEEUOPT(TEST) and 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 with DTCN or CEDF.
- **Batch Mode**

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

See “Debugging CICS programs” on page 116 for more details.

Invoking your program for a debugging session

Invoking Debug Tool varies depending on the environment where you are debugging your program. Before you begin your session, make sure all Debug Tool and program libraries are available and that all necessary Debug Tool files, such as the session log file, the primary commands file, the preferences file, and any desired USE files are defined and created.

Invoking Debug Tool under MVS in TSO

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 data sets are available. This might involve defining them as part of a STEPLIB library.

Note: High-level qualifiers and load library names will be specific to your installation.

The installation options will determine whether or not this step is needed. See the *OS/390 Language Environment Programming Guide* for more information.

2. Access all other data sets containing files your program needs.
3. If you want a session log file, allocate one. This is a file that 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 31.

After accessing all necessary data sets, the command line is used to define the preferences file setup.pref and the session log file session.log as shown in the following example:

```
ALLOCATE FILE(insppref) DATASET(setup.pref) REUSE
ALLOCATE FILE(insplog) DATASET(session.log) REUSE
CALL tstscrt3 '/TEST'
```

No primary commands file is created. The run-time TEST option is entered from the command line during invocation of the COBOL program tstscrt3. Default run-time suboptions are assumed, as well as the Language Environment default run-time options for your installation.

Invoking your program when starting a session

The following CLIST fragment shows how to define Debug Tool-related files and invoke the C program prog1 with the run-time TEST option:

```
ALLOC FI(inspsafe) DA(debug.save)           REUSE
ALLOC FI(insplog)  DA(debug.log)             REUSE
ALLOC FI(insppref) DA(debug.preferen)        REUSE
```

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

Files include the session log file, debug.log; the preferences file, debug.preferen; and the settings file, debug.save, a Debug Tool file that saves Debug Tool settings for use in future debugging sessions. Its Debug Tool-supplied default ddname is inspsafe. All necessary data sets must be available prior to invoking this CLIST.

For more information about Language Environment run-time options like TRAP(ON), see *OS/390 Language Environment Programming Guide*.

Invoking your program from a terminal that works only in line mode results in a line-mode session of Debug Tool. If you want to debug in line mode and you have a 3270-compatible terminal that is capable of sustaining a full-screen session, you must specify SET SCREEN OFF. You can specify this with the run-time TEST option by including the command in a preferences file, or by specifying it as a command string (for example, TEST(*,*, "SET SCREEN OFF", insppref)). For more information on line mode debugging, see "Using Debug Tool in line mode" on page 107.

Invoking Debug Tool under CMS

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

1. Access the product minidisk where Debug Tool resides.
2. Access any other minidisks containing files your programs need.
3. Load any text decks your programs need. For example, to use PL/I, C, COBOL and assembler on VM, the following MACLIB, TXTLIB and LOADLIB definitions would be required:

```
GLOBAL MACLIB SCEEMAC OSMACRO
GLOBAL TXTLIB SCEELKED CMSLIB
GLOBAL LOADLIB SCEERUN
```
4. Create and define any Debug Tool commands files you need, such as a preferences file, a USE file, or a primary commands file.
5. Define the session log file. This is a file that keeps a record of your debugging session, and can be used as a commands file during subsequent sessions.
6. Start your program with the run-time TEST option, specifying the appropriate suboption.

Note: You can also include a call to CEETEST, PLITEST, or __ctest() in the program's source.

After you access all necessary disks and load required text decks, the command line is used to define the preferences file setup pref a and the session log file seslog log a as shown in the following example:


```
FILEDEF insppref DISK setup pref a (LRECL 80 RECFM F
FILEDEF insplog DISK seslog log a (LRECL 72 RECFM F
LOAD tstscrpt2
START * TEST/
```

No primary commands file is created. The run-time TEST option is entered from the command line during invocation of the C program `tstscrpt2`. Default suboptions are assumed.

If you created a load module with GENMOD, enter:

```
FILEDEF insppref DISK setup pref a (LRECL 80 RECFM F
FILEDEF insplog DISK seslog log a (LRECL 72 RECFM F
tstscrpt2 TEST/
```

The REXX EXEC shown below, called `startup exec`, is created to define all Debug Tool-related files and invoke the COBOL program `prog1` with the run-time TEST option. `prog1` must be a load module.

```
'FILEDEF insplog DISK dbg log a (LRECL 72 RECFM F'
'FILEDEF insppref DISK dbg pref a (LRECL 80 RECFM F
'FILEDEF inspin DISK dbg cmds a (LRECL 72 RECFM F'
'FILEDEF inspsafe DISK dbg settings a (LRECL 80 RECFM F'
'GENMOD prog1 '

'prog1 * /TEST(,inspin,;,insppref)'
```

This assumes that the run-time CBL0PTS option was set to ON in the CEEDOPT or CEEUOPT assembly programs containing defaults and user-defined Language Environment options. See *OS/390 Language Environment Programming Guide* for more information.

Files include the session log file, `dbg log a`, and `dbg settings a`, a Debug Tool file that saves Debug Tool settings for use in future debugging sessions. Its Debug Tool-supplied ddname is `inspsafe`. Also defined are two preallocated files: `dbg pref a` (the Debug Tool preferences file) and `dbg cmds a` (the Debug Tool primary commands file).

For more information about `inspsafe`, see “Customizing colors” on page 98 and “Customizing settings” on page 99.

Using alternative Debug Tool invocation methods

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

- Language Environment provides the callable service CEETEST which is invoked from Language Environment-enabled languages.
- For C or C++ programs, you can use a `__ctest()` function call or include a `#pragma runopts` specification in your program.
Note: The `__ctest()` function is not supported in CICS.
- For PL/I programs, you can use a call to PLITEST or by including a PLIXOPT string which specifies the correct run-time TEST suboptions to invoke Debug Tool.

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 "Other run-time TEST option considerations" on page 25 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/C++ Include `leawi.h` header file.

PL/I Include CEEIBMAW and CEEIBMCT. CEEIBMAW is in the Language Environment SCEESAMP data set. See the example on page 35.

Batch and CICS Nonterminal Processes

We strongly recommend that you use feedback codes (fc) when using CEETEST to initiate Debug Tool from a batch process or a CICS nonterminal task; otherwise, results are unpredictable.

The syntax for CEETEST is:

For C/C++

```
►► void CEETEST ( string_of_commands , fc ) ►►
```

For COBOL

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

For PL/I

```
►► CALL CEETEST ( *string_of_commands , *fc ) ►►
```

string_of_commands (input)

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

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

If the `string_of_commands` is omitted, Debug Tool will prompt for commands in interactive mode.

fc (output)

A 12-byte *feedback* code, optional in some languages, that indicates the result of this service.

CEE000 Severity = 0
 Msg_No = Not Applicable
 Message = Service completed successfully

CEE2F2 Severity = 3
 Msg_No = 2530
 Message = A debugger was not available

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

Language Environment provides a callable service called CEEDCOD to help you decode the fields in the feedback code. Requesting the return of the feedback code is recommended. See *OS/390 Language Environment Programming Guide* for details.

For C/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: In this example, a Null command string is passed to Debug Tool and a pointer to the Language Environment feedback code is returned. If no other TEST run-time options have been compiled into the program, the call to CEETEST 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);
}
```

In this example, a string of valid Debug Tool commands is passed to Debug Tool and a pointer to Language Environment feedback code is returned. The call to CEETEST 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 termi-

nation and prompts you for commands. The command LIST(z) is discarded when the command G0 is executed.

Note: If you include a STEP or G0 in your command string, all commands after that are not processed. The command string operates like a commands file.

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

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

    strcpy(commands.string, "AT LINE 23; {LIST(x); LIST(y);} G0; LIST(z)");
    commands.length = strlen(commands.string);
    :
    CEETEST(&commands, &fc);
    :
}
```

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); } G0;");
    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: 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 and line mode)” on page 195.

```

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

01 Parm.
  AA                      Picture 99    Value 14.
  BB                      Picture x(14) Value 'SET SCREEN ON;'.

```

```
CALL Debugger USING Parm. FC.
```

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: 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 */
```

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.

Using alternative invocation methods

```
DCL  ch char(50)
      init('AT STATEMENT 10 D0; LIST(x); LIST(y); END; GO;');

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

DCL  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(assembly) ;

CALL  CEETEST(ch, fb);
```

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 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:

```

▶▶—CALL—PLITEST—┬──────────────────────────────────┴───▶▶
                  └(—character_string_expression—)┘

```

character_string_expression

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

Notes:

1. If Debug Tool executes a command in a CALL PLITEST command string that causes control to return to the program (GO for example), any commands remaining to be executed in the command string are discarded.
2. If you don't want to compile your program with hooks, you can use CALL PLITEST statements as hooks and insert them at strategic points in your program. If you decide to use this method, you still need to compile your application so that symbolic information is created.

Examples of PLITEST calls for PL/I

```
CALL PLITEST;
```

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

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

Using alternative invocation methods

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.

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

CALL PLITEST(ch);
```

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.

Invoking Debug Tool with the `__ctest()` function

You can also use the C/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/C++:

```
__ctest(NULL);
```

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

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 command G0.

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

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.

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

char *ch = "at line 23 printf(\"x.y is %d\\n\", x.y); go;";
char buffer[35.132];

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

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

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 G0 command.

Specifying run-time TEST option with #pragma runopts in C and 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 (test(suboption,suboption...))
```

This #pragma must appear before the first statement in your source file. For example, if you specified the following in the source:

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

then entered TEST on the command line, the result would be

```
TEST(ALL,*,PROMPT).
```

TEST overrides the NOTEST option specified in the #pragma and, because TEST does not contain any suboptions of its own, the suboptions ALL, *, and PROMPT remain in effect.

If you 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 *OS/390 C/C++ User's Guide*.

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 PF key definitions used in this chapter are the default settings.

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 run time. A detailed description of the compile-time TEST option for each language is provided in Chapter 2, "Preparing to debug your program" on page 5.

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

- for C and 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.

Invoking your program with Debug Tool

Invoking your program with Debug Tool in one of the following environments is described in detail in the appropriate sections in Chapter 7, "Using Debug Tool in different modes and environments" on page 107.

TSO
IMS
CICS
DB2

This section includes some helpful hints to provide a simple path to help you learn how to use Debug Tool.

One way to invoke Debug Tool is by using the Language Environment run-time TEST option.

For TSO you need to include the Debug Tool library into your STEPLIB concatenation and invoke your program with the run-time TEST option as shown in the following example for C, C++, and PL/I:

```
MYPROG TEST / prog arg list
```

For COBOL, invoke your program as follows:

```
MYPROG prog arg list / TEST
```

Contact your systems programmer if you do not know the name of the Debug Tool library on your system.

For CICS, make sure Debug Tool is installed in your CICS region before you enter DTCN to start the Debug Tool control transaction. Press *PF4* to save the default debugging profile. Press *PF3* to exit from the DTCN transaction. Enter the name of the transaction you want to debug.

If you build your application using the c89 or C++ OpenEdition Shell Utilities, do the following steps:

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

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

Note: The double quotes in the command-line above are required.

2. Set up your TSO environment, as described above.
3. Debug the program under TSO by entering the following:

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

Note: The single quotes in the command-line above are required.

`ENVAR('PWD=/u/mike/app')` sets the environment variable `PWD` to the path from where the source files were compiled. Debug Tool uses this information to determine from where it should read the source files.

If you are working in the MVS OpenEdition environment, you can put Debug Tool into your STEPLIB and set up the Language Environment run-time `TEST` option before invoking your program by writing a simple shell script as shown in the following example:

```
# rundbg.sh - set up debug environment, and run program.
export STEPLIB=MVSID.TEST.LOAD:\
SYSID.DBGTOOL.SEQAMOD:SYSID.CEE180.SCEERUN
export _CEE_RUNTOPS="TEST POSIX(ON)"
myprogram.exe
```

Ending a debug session

When you have finished debugging your program, you can either press *PF3* (QUIT) 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 views your source code

The *LOG* window records your commands with Debug Tool's response

The *MONITOR* window continuously displays the value of monitored variables and other items depending on the command used.

Saving your log file for future use

To get a record of how many times each line of your code was executed, take the following steps:

1. Allocate the INSPLOG ddname if you want to keep a permanent record of the results. Under CICS, instead of allocating the INSPLOG ddname, you must issue the command:

```
SET LOG ON FILE fileid
```

where *fileid* is the data set name where LOG file output is written.

2. Issue the command:

```
SET FREQUENCY ON;
```

After you have entered the SET FREQUENCY ON command, your source window is updated to show the current frequency count. Remember that this command starts the statistic gathering to display the actual count, so if your application has already executed a section of code, the data for these executed statements will not be available.

If you want statement counts for the entire program, issue:

```
GO ;  
LIST FREQUENCY * ;
```

which lists the number of times each statement was run. When you quit, the results are written to the LOG file. You can issue the LIST FREQUENCY * at any time, but it will only display the frequency count for the currently active compile unit.

Help

You can find 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
```

Now reopen the SOURCE window with:

```
WINDOW OPEN SOURCE
```

to see the results.

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 `PF5` (`FIND`). Pressing `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 `PF7` and down by pressing `PF8`. To page through other windows, place the cursor in the desired window and press `PF7(UP)` or `PF8(DOWN)`.

You can toggle one of the *SOURCE*, *LOG*, or *MONITOR* windows to full screen (temporarily not displaying the others) by moving the cursor into the window you want to zoom and pressing `PF10(ZOOM)`. Another `PF10` will toggle back. `PF11(ZOOM LOG)` 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 overwrite the name after *SOURCE*: on the top line of the *SOURCE* window with the desired name. This only works if the CU is already known to Debug Tool. You might want to issue the `LIST NAMES CUS` command first to determine which CUs are known.

Alternately you can enter the command:

```
LIST NAMES CUS
```

and a list of Compilation Units will be written to the *LOG* window, as shown in the following example:

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

You can overwrite/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 "USERID.MFISTART.C(READTKN)"
```

and then press `ENTER` to issue the command. Overtyping of a line in the *LOG* and issuing them as commands is a way to save keystrokes and errors in long commands.

Basic tasks of Debug Tool

Pressing *PF4* (LIST) 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. Overtyping the Listings/Source File field with the new name.

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

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 VS COBOL II or PL/I compile unit, it looks for the listing in a file named *hlq.cuname.LIST*. For COBOL/370, COBOL for MVS, and COBOL for OS/390, the Debug Tool looks for the listing in a partitioned data set member named *cuname*.

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 *PF6*(AT/CLEAR) 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 (including C++ constructors and static object initialization).

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

Pressing *PF9*(G0) 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*.

The command STEP OVER runs the called function without stepping into it. If you accidentally step into a function when you meant to step over it, issue the STEP RETURN command which steps to the return point (just after the call point).

Displaying a variable's value

To LIST the contents of a single variable, move the cursor to the variable name and press *PF4*(LIST). The value of the variable is displayed in the LOG window.

Continuously displaying a variable's value

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

```
MONITOR LIST num ;
```

and the output for this command 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 *PF1* to be the STEP OVER command with the message STEPOVER appearing under the *PF1* key. You do it by entering:

```
SET PF1 "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

```
EQAI807E 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 E, "Debug Tool Messages" on page 355.

Finding a renamed source or Listing file using Debug Tool

At compile time, the source or listing files might have had different names than they do now.

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

Using a C program for Debug Tool session

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



```

/*----- 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)
        {
            case T_STOP:
                break;
            case T_INTEGER:
                num = atoi(word);
                push(&stack,num);    /* CALC1 statement */
                break;
            case T_PLUS:
                push(&stack, pop(&stack)+pop(&stack) );
                break;
            case T_MINUS:
                num = pop(&stack);
                push(&stack, num-pop(&stack));
                break;
            case T_TIMES:
                push(&stack, pop(&stack)*pop(&stack) );
                break;
            case T_DIVIDE:
                num = pop(&stack);
                push(&stack, num/pop(&stack));    /* 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;
}

```

Using a C program for Debug Tool session

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

```

/*----- FILE READTOKN.C -----*/
/*
/* A function to read input and tokenize it for a simple calculator */
/*-----*/
#include <ctype.h>
#include <stdio.h>
#include "calc.h"
/*-----*/
/* action: get next input char, update index for next call */
/* return: next input char */
/*-----*/
static char nextchar(void)
{
/*-----*/
/* input action: */
/* 2 push 2 on stack */
/* 18 push 18 */
/* + pop 2, pop 18, add, push result (20) */
/* = output value on the top of the stack (20) */
/* 5 push 5 */
/* / pop 5, pop 20, divide, push result (4) */
/* = output value on the top of the stack (4) */
/*-----*/
char * buf_in = "2 18 + = 5 / = ";
static int index; /* starts at 0 */
char ret;
ret = buf_in[index];
++index;
return ret;
}
/*-----*/
/* output: buf - null terminated token */
/* return: token type */
/* action: reads chars through nextchar() and tokenizes them */
/*-----*/
Token read_token(char buf[])
{
int i;
char c;
/* skip leading white space */
for( c=nextchar();
isspace(c);
c=nextchar())
;
buf[0] = c; /* get ready to return single char e.g. "+" */
buf[1] = 0;
switch(c)
{
case '+' : return T_PLUS;
case '-' : return T_MINUS;
case '*' : return T_TIMES;
case '/' : return T_DIVIDE;
case '=' : return T_EQUALS;
default:
i = 0;
while (isdigit(c)) {
buf[i++] = c;
c = nextchar();
}
buf[i] = 0;
if (i==0)
return T_STOP;
else
return T_INTEGER;
}
}

```

C tasks

The following sections identify typical tasks you might want to perform while using Debug Tool with your C program and explanations on 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.

Modifying the value of a variable

To LIST the contents of a single variable, move the cursor to the variable name and press `PF4(LIST)`. The value is displayed in the LOG window. This is equivalent to entering `LIST TITLED variable` on the command line. For instance, run the CALC program to the statement labeled **CALC1**. Move the cursor over `num` and press `PF4(LIST)`. The following appears in the LOG window:

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

To modify the value of `num` to 22, overwrite 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 `PF2(STEP)` and step until the statement labeled `PUSHPOP2` is reached. To view the attributes of variable `ptr`, issue the Debug Tool command:

```
DESCRIBE ATTRIBUTES *ptr;
```

The result in the LOG window is:

```
ATTRIBUTES for * ptr  
  struct int_link {  
    struct int_link *next;  
    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 G0. 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) G0; }
```

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. Depending on the compiler you are using, or if `"USERID.MFISTART.C(PUSHPOP)"` is fetched later on by the application, this compile unit might not be known to Debug Tool. If it is displayed, enter:

```
SET QUALIFY CU "USERID.MFISTART.C(PUSHPOP)"
AT ENTRY push;
G0 ;
```

or

```
AT ENTRY "USERID.MFISTART.C(PUSHPOP)" :>push
G0;
```

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

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" ;
G0 ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" AT ENTRY push; G0;
```

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) ;
```

Debugging a DLL

Build *PUSHPOP.C* as a DLL, exporting `push()` and `pop()`. Build *CALC.C* and *READTKN.C* as the program which imports `push()` and `pop()` from the DLL named *PUSHPOP*. When the application *CALC* starts the DLL, *PUSHPOP* will not be known to Debug Tool. Use the `AT APPEARANCE` breakpoint to gain control in the DLL the first time code in that compilation unit appears, as shown in the following example:

```
AT APPEARANCE "USERID.MFISTART.C(PUSHPOP)" ;
GO ;
```

The only purpose of this `APPEARANCE` breakpoint is to gain control the first time a function in the *PUSHPOP* compilation unit is run. When this happens, you can set breakpoints in *PUSHPOP*.

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 "USERID.MFISTART.C(READTKN)" :> read_token.
From LINE 18 in C function "USERID.MFISTART.C(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 data set `USERID.DTUSE(TRACE)` that contains the following Debug Tool commands:

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

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

```
USE USERID.DTUSE(TRACE)
```

The trace of running the program listed below after executing the use file will be displayed in the log window.

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

The following trace in the LOG window is displayed after running the sample program, with the Use file as a source of input for Debug Tool commands:

```
>main
>foo
<foo
<main
```

If you do not want to create the Use file, you can enter the commands through the command line, and the same effect is achieved.

Finding unexpected storage overwrite errors

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:

```
TEST STORAGE(FD,FB,F9)
```

the first subparameter of STORAGE is the fill byte for storage allocated from the heap. For example, storage allocated through *malloc()* is filled with the byte 0xFD. If you see this byte repeated through storage, it is likely uninitialized heap storage.

The second subparameter of STORAGE is the fill byte for storage allocated from the heap but then freed. For example, storage freed by calling *free()* might be filled with the byte 0xFB. If you see this byte repeated through storage, it is likely storage that was allocated on the heap, but has been freed.

The third subparameter of STORAGE is the fill byte for auto storage variables in a new stack frame. If you see this byte repeated through storage, it is likely uninitialized auto storage. The values chosen 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 G0 BYPASS command. This allows you to continue your debugging session without raising a condition.

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.

```
/*----- FILE CALC.HPP -----*/
/*
/* Header file for CALC.CPP PUSHPOP.CPP READTKN.CPP
/* a simple calculator
/*-----*/
typedef enum toks {
    T_INTEGER,
    T_PLUS,
    T_TIMES,
    T_MINUS,
    T_DIVIDE,
    T_EQUALS,
    T_STOP,
} Token;
extern "C" Token read_token(char buf[]);
class IntLink {
private:
    int i;
    IntLink * next;
public:
    IntLink();
    ~IntLink();
    int get_i();
    void set_i(int j);
    IntLink * get_next();
    void set_next(IntLink * d);
};
class IntStack {
private:
    IntLink * top;
public:
    IntStack();
    ~IntStack();
    void push(int);
    int pop();
};
```

Using a C++ program for Debug Tool session

```
/*----- FILE CALC.CPP -----*/
/*
/* A simple calculator which does operations on integers which
/* are pushed and popped on a stack
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include "calc.hpp"
IntStack stack;
int main()
{
    Token tok;
    char word[100];
    char buf_out[100];
    int num;
    for(;;)
    {
        tok=read_token(word);
        switch(tok)
        {
            case T_STOP:
                break;
            case T_INTEGER:
                num = atoi(word);
                stack.push(num);    /* CALC1 statement */
                break;
            case T_PLUS:
                stack.push(stack.pop()+stack.pop());
                break;
            case T_MINUS:
                num = stack.pop();
                stack.push(num-stack.pop());
                break;
            case T_TIMES:
                stack.push(stack.pop()*stack.pop() );
                break;
            case T_DIVIDE:
                num = stack.pop();
                stack.push(num/stack.pop()); /* CALC2 statement    */
                break;
            case T_EQUALS:
                num = stack.pop();
                sprintf(buf_out,"= %d ",num);
                stack.push(num);
                break;
        }
        if (tok==T_STOP)
            break;
    }
    return 0;
}
```

```

/*----- FILE: PUSHPOP.CPP -----*/
/*
/* Push and pop functions for a stack of integers
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include "calc.hpp"
/*-----*/
/* input: num - value to push on the stack
/* action: get a link to hold the pushed value, push link on stack
/*-----*/
void IntStack::push(int num) {
    IntLink * ptr;
    ptr = new IntLink;
    ptr->set_i(num);
    ptr->set_next(top);
    top = ptr;
}
/*-----*/
/* return: int value popped from stack (0 if stack is empty)
/* action: pops top element from stack and get return value from it
/*-----*/
int IntStack::pop() {
    IntLink * ptr;
    int num;
    ptr = top;
    num = ptr->get_i();
    top = ptr->get_next();
    delete ptr;
    return num;
}
IntStack::IntStack() {
    top = 0;
}
IntStack::~IntStack() {
    while(top)
        pop();
}
IntLink::IntLink() { /* constructor leaves elements unassigned */
}
IntLink::~IntLink() {
}
void IntLink::set_i(int j) {
    i = j;
}
int IntLink::get_i() {
    return i;
}
void IntLink::set_next(IntLink * p) {
    next = p;
}
IntLink * IntLink::get_next() {
    return next;
}

```

Using a C++ program for Debug Tool session

```
/*----- FILE READTKN.CPP -----*/
/*
/* A function to read input and tokenize it for a simple calculator */
/*-----*/
#include <ctype.h>
#include <stdio.h>
#include "calc.hpp"
/*-----*/
/* action: get next input char, update index for next call */
/* return: next input char */
/*-----*/
static char nextchar(void)
{
    /*    input    action
    *    ----    -
    *    2        push 2 on stack
    *    18       push 18
    *    +        pop 2, pop 18, add, push result (20)
    *    =        output value on the top of the stack (20)
    *    5        push 5
    *    /        pop 5, pop 20, divide, push result (4)
    *    =        output value on the top of the stack (4)
    */
    char * buf_in = "2 18 + = 5 / = ";
    static int index; /* starts at 0 */
    char ret;
    ret = buf_in[index];
    ++index;
    return ret;
}
/*-----*/
/* output: buf - null terminated token */
/* return: token type */
/* action: reads chars through nextchar() and tokenizes them */
/*-----*/
extern "C"
Token read_token(char buf[])
{
    int i;
    char c;
    /* skip leading white space */
    for( c=nextchar();
        isspace(c);
        c=nextchar())
        ;
    buf[0] = c; /
    * get ready to return single char e.g. "+" */
    buf[1] = 0;
    switch(c)
    {
        case '=' : return T_PLUS;
        case '-' : return T_MINUS;
        case '*' : return T_TIMES;
        case '/' : return T_DIVIDE;
        case '=' : return T_EQUALS;
        default:
            i = 0;
            while (isdigit(c)) {
                buf[i++] = c;
                c = nextchar();
            }
            buf[i] = 0;
            if (i==0)
                return T_STOP;
            else
                return T_INTEGER;
    }
}
```

C++ tasks

The following sections identify typical tasks you might want to perform while using Debug Tool with your C++ program and explanations on 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

You need to include the C++ signature along with the function name to set an AT ENTRY or AT CALL breakpoint for a C++ function.

To facilitate entering the breakpoint, you can display *PUSHPOP.CPP* in the source window by overtyping the name of the file on the top line of the source window. This makes *PUSHPOP.CPP* your currently qualified program. You can then issue the command;

```
LIST NAMES
```

which displays the names of all the blocks and variables for the currently qualified program. Debug Tool displays the following in the LOG window:

```
IntStack::~IntStack()
IntStack::IntStack()
IntLink::get_i()
IntLink::get_next()
IntLink::~IntLink()
IntLink::set_i(int)
IntLink::set_next(IntLink*)
IntLink::IntLink()
```

Now you can save some strokes by inserting the command next to the block name.

To halt just before `IntStack::push(int)` is called, insert `AT CALL` next to the function signature, and by pressing the enter key, the entire command will be placed in the command line. Now, with `AT CALL IntStack::push(int) ;` in the command line, you can enter the command:

```
AT CALL IntStack::push(int) ;
```

To halt just after `IntStack::push(int)` is called, issue the command:

```
AT ENTRY IntStack::push(int) ;
```

in the same way as the `AT CALL` command.

To be able to halt, the file with the calling code must be compiled with the compile-time `TEST` option.

Modifying the value of a variable

To `LIST` the contents of a single variable, move the cursor to the variable name and press `PF4(LIST)`. The value is displayed in the LOG window. This is equivalent to entering `LIST TITLED variable` on the command line.

For example, run the *CALC* program and step into the first call of function `IntStack::push(int)` until just after the `IntLink` has been allocated.

```
LIST TITLED;
```

will display in the LOG window all the variables that are valid for the current context. Below is the output from the command:

```
this = 0x7042708
ptr = 0x7042A00
num = 2
```

To modify the value of `num` to 22, overtype the `num = 2` line to be `num = 22`, press ENTER to put it on the command line and press ENTER again to issue the command.

You can enter most C++ expressions on the command line.

To view the attributes of variable `ptr` in `IntStack::push(int)` issue the Debug Tool command:

```
DESCRIBE ATTRIBUTES *ptr;
```

The result in the LOG window is:

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

So for most classes, structures, and unions, this can act as a simple browser.

You can list all the values of the data members of the class object pointed to by `ptr` with the command:

```
LIST *ptr ;
```

with results in the LOG window similar to:

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

You can change the value of data member of a class object by issuing the assignment as a command as in this example:

```
(* ptr).i = 33 ;
```

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 fails under certain conditions. You don't want to set a simple line breakpoint because you will have to keep entering G0. For example, in `main` you want to stop in `T_DIVIDE` only if the divisor is 0 (before the exception occurs). Set the breakpoint like this:

```
AT 39 { if(num != 0) G0; }
```

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

Viewing and modifying data members of the `this` pointer

If you step into a class method, for example, one for class `IntLink`, the command:

```
LIST TITLED ;
```

responds with a list which includes `this`. With the command:

```
DESCRIBE ATTRIBUTES *this ;
```

you will see the types of the data elements pointed to by the `this` pointer. With the command:

```
LIST *this ;
```

you will list the data member of the object pointed to and see something like:

```
LIST * this ;
(* this).i = 4
(* this).next = 0x0
```

in the *LOG* window. To modify element `i`, enter either the command:

```
i = 2001;
```

or, if you have ambiguity (for example, you also have an auto variable named `i`), enter:

```
(* this).i = 2001 ;
```

Debugging when only a few parts are compiled with TEST

Suppose you want to set a breakpoint at entry to function `IntStack::push(int)` in the file *PUSHPOP.CPP*. *PUSHPOP.CPP* has been compiled with `TEST` but the other files have not. Debug Tool comes up with an empty *SOURCE* window. To display the 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.

Depending on the compiler you are using, or if `USERID.MFISTART.CPP(PUSHPOP)` is fetched later on by the application, this compile unit might or might not be known to Debug Tool, and the PDS member *PUSHPOP* might or might not be displayed. If it is displayed, enter:

```
SET QUALIFY CU "USERID.MFISTART.CPP(PUSHPOP)"
AT ENTRY IntStack::push(int) ;
GO ;
```

or

```
AT ENTRY "USERID.MFISTART.CPP(PUSHPOP)":>push
GO
```

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

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" ;
GO ;
```

You can also combine the breakpoints as follows:

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" AT ENTRY push; GO;
```

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

```
AT ENTRY IntStack::push(int) ;
```

Capturing output to stdout

To redirect stdout to the LOG window, issue the following command:

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

With this set you will not only capture stdout from your program, but also from interactive function calls. For example, you can interactively use cout on the command line to display a null terminated string by entering:

```
cout << sptr ;
```

You might find this easier than using LIST STORAGE.

Note: SET INTERCEPT is not supported under CICS.

Invoking interactive function calls

You can invoke a library function (such as strlen) or one of the programs functions interactively by calling it on the command line. The same is true of C linkage functions such as read_token. You cannot call C++ linkage functions interactively.

In the example below, we call read_token interactively.

```
AT CALL read_token;
GO;
read_token(word);
```

The calculator produces different results than before because of the additional token removed from input.

Displaying raw storage

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 shown in this example:

```
puts(ptr) ;
```

Debugging a DLL

Build *PUSHPOP.CPP* as a DLL, exporting *IntStack::push(int)* and *IntStack::pop()*. Build *CALC.CPP* and *READTOKN.CPP* as the program which imports *IntStack::push(int)* and *IntStack::pop()* from the DLL named *PUSHPOP*. When the application *CALC* starts, the DLL *PUSHPOP* is not known to Debug Tool. Use the AT APPEARANCE breakpoint, as shown in the following example, to gain control in the DLL the first time code in that compile unit appears.

```
AT APPEARANCE "USERID.MFISTART.CPP(PUSHPOP)" ;
GO ;
```

The only purpose of this APPEARANCE breakpoint is to gain control the first time a function in the *PUSHPOP* compilation unit is run. When this happens, you can set breakpoints in *PUSHPOP*.

Getting a function traceback

Often when you get close to a programming error, you want to know how you got into that situation, especially what the traceback of calling functions is. To get this information, issue the command:

```
LIST CALLS ;
```

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

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

the LOG window will contain something like:

```
At ENTRY in C function "USERID.MFISTART.CPP(READTKN)" :> read_token.
From LINE 18 in C function "USERID.MFISTART.CPP(CALC)" :> main :> %BLOCK2.
```

which shows the traceback of callers.

Getting a run-time frequency count

To get a record of how many times each line of your code is executed, take the following steps:

1. Allocate the INSPLOG ddname if you want to keep a permanent record of the results. Under CICS, instead of allocating the INSPLOG ddname, issue the following command:

```
SET LOG ON FILE fileid
```

where fileid is the data set name of where LOG file output is written.

2. Issue the command:

```
SET FREQUENCY ON;
```

After you have entered the SET FREQUENCY ON command, your source window is updated to show the current frequency count. One thing to keep in mind is that the command actually starts the statistic gathering to display the actual count. If your application has already executed a section of code, the data for those previously executed statements won't be available.

If you want statement counts for the entire program, issue the following commands:

```
GO ;
LIST FREQUENCY * ;
```

which list the number of times each statement was run. When you quit, the results are written to the LOG file. At any time, you can issue the LIST FREQUENCY *; but it will only display the frequency count for the currently active compile unit.

Tracing the run-time path for code compiled with TEST

To trace a program showing the entry and exit of that program without requiring any changes to it, place the following Debug Tool commands, shown in the example below, in a file and USE them when Debug Tool initially displays your program. Assume you have a data set that contains USERID.DTUSE(TRACE) and contains the following Debug Tool commands:

```

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

```

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

```
USE USERID.DTUSE(TRACE)
```

The trace of running the program listed below after executing the use file is displayed in the log window:

```

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

```

The following trace in the LOG window is displayed after running the sample program, using the use file as a source of input for Debug Tool commands:

```

>main
>foo(int,int)
<foo(int,int)
<main

```

If you do not want to create the use file, you can enter the commands through the command line, and the same effect will be achieved.

Finding unexpected storage overwrite errors

During program run time, some storage might unexpectedly change its value and you would like to find out when and where this happened. Consider this simple example where function `set_i` changes more than the caller expects it to change.

```

struct s { int i; int j;};
struct s a = { 0, 0 };

/* function sets only field i */
void set_i(struct s * p, int k)
{
    p->i = k;
    p->j = k; /* error, it unexpectedly sets field j also */
}
main() {
    set_i(&a,123);
}

```

Find the address of `a` with the command:

```
LIST &(a.j) ;
```

Suppose the result is 0x7042A04. To set a breakpoint 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:

```
TEST STORAGE(FD,FB,F9)
```

the first subparameter of STORAGE is the fill byte for storage allocated from the heap. For example, storage allocated through operator new is filled with the byte 0xFD. If you see this byte repeated throughout storage, it is likely uninitialized heap storage.

The second subparameter of STORAGE is the fill byte for storage allocated from the heap but then freed. For example, storage freed by the operator delete might be filled with the byte 0xFB. If you see this byte repeated throughout storage, it is likely storage that was allocated on the heap, but has been freed.

The third subparameter of STORAGE is the fill byte for auto storage variables in a new stack frame. If you see this byte repeated throughout storage, it is likely that it is uninitialized auto storage. The values chosen 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 command 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 70. It is a simple program which calls two subprograms to calculate a loan payment amount and the future value of a series of cash flows. Several COBOL intrinsic functions are utilized.

```
*****
* COBCALC                                     *
*                                           *
* A simple program which allows financial functions to *
* be performed using intrinsic functions.           *
*                                           *
*****
IDENTIFICATION DIVISION.
PROGRAM-ID. COBCALC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PARM-1.
    05  CALL-FEEDBACK      PIC XX.
01  FIELDS.
    05  INPUT-1            PIC X(10).
01  INPUT-BUFFER-FIELDS.
    05  BUFFER-PTR        PIC 9.
    05  BUFFER-DATA.
        10  FILLER        PIC X(10)  VALUE "LOAN".
        10  FILLER        PIC X(10)  VALUE "PVALUE".
        10  FILLER        PIC X(10)  VALUE "pvalue".
        10  FILLER        PIC X(10)  VALUE "END".
    05  BUFFER-ARRAY      REDEFINES BUFFER-DATA
                          OCCURS 4 TIMES
                          PIC X(10).

PROCEDURE DIVISION.
    DISPLAY "CALC Begins." UPON CONSOLE.
    MOVE 1 TO BUFFER-PTR.
    MOVE SPACES TO INPUT-1.
    * Keep processing data until END requested
    PERFORM ACCEPT-INPUT UNTIL INPUT-1 EQUAL TO "END".
    * END requested
    DISPLAY "CALC Ends." UPON CONSOLE.
    GOBACK.
    * End of program.
```

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

```

*
* Accept input data from buffer
*
ACCEPT-INPUT.
    MOVE BUFFER-ARRAY (BUFFER-PTR) TO INPUT-1.
    ADD 1 BUFFER-PTR GIVING BUFFER-PTR.
* Allow input data to be in UPPER or lower case
    EVALUATE FUNCTION UPPER-CASE(INPUT-1) CALC1
        WHEN "END"
            MOVE "END" TO INPUT-1
        WHEN "LOAN"
            PERFORM CALCULATE-LOAN
        WHEN "PVALUE"
            PERFORM CALCULATE-VALUE
        WHEN OTHER
            DISPLAY "Invalid input: " INPUT-1
    END-EVALUATE.
*
* Calculate Loan via CALL to subprogram
*
CALCULATE-LOAN.
    CALL "COBLOAN" USING CALL-FEEDBACK.
    IF CALL-FEEDBACK IS NOT EQUAL "OK" THEN
        DISPLAY "Call to COBLOAN Unsuccessful.".
*
* Calculate Present Value via CALL to subprogram
*
CALCULATE-VALUE.
    CALL "COBVALU" USING CALL-FEEDBACK.
    IF CALL-FEEDBACK IS NOT EQUAL "OK" THEN
        DISPLAY "Call to COBVALU Unsuccessful.".

```

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

```

*****
* COBLOAN                                     *
*                                             *
* A simple subprogram which calculates payment amount *
* for a loan.                               *
*                                             *
*****
IDENTIFICATION DIVISION.
PROGRAM-ID. COBLOAN.
ENVIRONMENT DIVISION.
DATA DIVISION.

```

Figure 3 (Part 1 of 2). Sample COBOL Program - Subroutine COBLOAN

```

WORKING-STORAGE SECTION.
01 FIELDS.
   05 INPUT-1          PIC X(26).
   05 PAYMENT          PIC S9(9)V99 USAGE COMP.
   05 PAYMENT-OUT      PIC $$$,$$$,$$9.99 USAGE DISPLAY.
   05 LOAN-AMOUNT      PIC S9(7)V99 USAGE COMP.
   05 LOAN-AMOUNT-IN   PIC X(16).
   05 INTEREST-IN      PIC X(5).
   05 INTEREST         PIC S9(3)V99 USAGE COMP.
   05 NO-OF-PERIODS-IN PIC X(3).
   05 NO-OF-PERIODS    PIC 99 USAGE COMP.
   05 OUTPUT-LINE      PIC X(79).
LINKAGE SECTION.
01 PARM-1.
   05 CALL-FEEDBACK    PIC XX.
PROCEDURE DIVISION USING PARM-1.
  MOVE "NO" TO CALL-FEEDBACK.
  MOVE "30000 .09 24 " TO INPUT-1.
  UNSTRING INPUT-1 DELIMITED BY ALL " "
    INTO LOAN-AMOUNT-IN INTEREST-IN NO-OF-PERIODS-IN.
* Convert to numeric values
  COMPUTE LOAN-AMOUNT = FUNCTION NUMVAL(LOAN-AMOUNT-IN).
  COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).
  COMPUTE NO-OF-PERIODS = FUNCTION NUMVAL(NO-OF-PERIODS-IN).
* Calculate annuity amount required
  COMPUTE PAYMENT = LOAN-AMOUNT *
    FUNCTION ANNUITY((INTEREST / 12 ) NO-OF-PERIODS).
* Make it presentable
  MOVE SPACES TO OUTPUT-LINE
  MOVE PAYMENT TO PAYMENT-OUT.
  STRING "COBLOAN: Repayment_amount_for_a_" NO-OF-PERIODS-IN
    "_month_loan_of_" LOAN-AMOUNT-IN
    "_at_" INTEREST-IN "_interest_is:_"
    DELIMITED BY SPACES
    INTO OUTPUT-LINE.
  INSPECT OUTPUT-LINE REPLACING ALL "_" BY SPACES.
  DISPLAY OUTPUT-LINE PAYMENT-OUT.
  MOVE "OK" TO CALL-FEEDBACK.
  GOBACK.

```

Figure 3 (Part 2 of 2). Sample COBOL Program - Subroutine COBLOAN

```

*****
* COBVALU                                     *
*                                           *
* A simple subprogram which calculates present value *
* for a series of cash flows.               *
*                                           *
*****
IDENTIFICATION DIVISION.
PROGRAM-ID. COBVALU.
ENVIRONMENT DIVISION.
DATA DIVISION.

```

Figure 4 (Part 1 of 2). Sample COBOL Program - Subroutine COBVALU

```

WORKING-STORAGE SECTION.
01 CHAR-DATA.
   05 INPUT-1          PIC X(10).
   05 PAYMENT-OUT      PIC $$$,$$$,$$9.99 USAGE DISPLAY.
   05 INTEREST-IN      PIC X(5).
   05 NO-OF-PERIODS-IN PIC X(3).
   05 INPUT-BUFFER     PIC X(10) VALUE "5069837544".
   05 BUFFER-ARRAY     REDEFINES INPUT-BUFFER
                       OCCURS 5 TIMES
                       PIC XX.
   05 OUTPUT-LINE      PIC X(79).
01 NUM-DATA.
   05 PAYMENT          PIC S9(9)V99 USAGE COMP.
   05 INTEREST         PIC S9(3)V99 USAGE COMP.
   05 COUNTER          PIC 99 USAGE COMP.
   05 NO-OF-PERIODS    PIC 99 USAGE COMP.
   05 VALUE-AMOUNT     OCCURS 99 PIC S9(7)V99 COMP.
LINKAGE SECTION.
01 PARM-1.
   05 CALL-FEEDBACK PIC XX.
PROCEDURE DIVISION USING PARM-1.
  MOVE "NO" TO CALL-FEEDBACK.
  MOVE ".12 5 " TO INPUT-1.
  UNSTRING INPUT-1 DELIMITED BY "," OR ALL " "
    INTO INTEREST-IN NO-OF-PERIODS-IN.
  * Convert to numeric values
  COMPUTE INTEREST = FUNCTION NUMVAL(INTEREST-IN).
  COMPUTE NO-OF-PERIODS = FUNCTION NUMVAL(NO-OF-PERIODS-IN).
  * Get cash flows
  PERFORM GET-AMOUNTS VARYING COUNTER FROM 1 BY 1 UNTIL
    COUNTER IS GREATER THAN NO-OF-PERIODS.
  * Calculate present value
  COMPUTE PAYMENT =
    FUNCTION PRESENT-VALUE(INTEREST VALUE-AMOUNT(ALL) ).
  * Make it presentable
  MOVE PAYMENT TO PAYMENT-OUT.
  STRING "COBVALU:_Present_value_for_rate_of_"
    INTEREST-IN "_given_amounts_"
    BUFFER-ARRAY (1) ",_ "
    BUFFER-ARRAY (2) ",_ "
    BUFFER-ARRAY (3) ",_ "
    BUFFER-ARRAY (4) ",_ "
    BUFFER-ARRAY (5) "_is:_"
    DELIMITED BY SPACES
    INTO OUTPUT-LINE.
  INSPECT OUTPUT-LINE REPLACING ALL "_" BY SPACES.
  DISPLAY OUTPUT-LINE PAYMENT-OUT.
  MOVE "OK" TO CALL-FEEDBACK.
  GOBACK.
*
* Get cash flows for each period
*
GET-AMOUNTS.
  MOVE BUFFER-ARRAY (COUNTER) TO INPUT-1.
  COMPUTE VALUE-AMOUNT (COUNTER) = FUNCTION NUMVAL(INPUT-1).

```

Figure 4 (Part 2 of 2). 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.

Setting a breakpoint to halt when certain routines are called

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

```
AT CALL COBLOAN ;
```

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

```
AT APPEARANCE COBVALU ;
```

You can display a list of all compile units that are known to Debug Tool by entering the command:

```
LIST NAMES CUS ;
```

The Debug Tool log displays something similar to:

```
LIST NAMES CUS ;
The following CUs are known in *:
COBCALC
COBLOAN
COBVALU
```

Additionally, you can combine the breakpoints as follows:

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

The purpose for the APPEARANCE breakpoint is to gain control the **first** time the COBVALU compile unit is run.

To take advantage of either AT ENTRY or AT APPEARANCE, you must compile the routine (COBVALU in the above example) program with the 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. The Debug Tool Log will display something similar to:

```
QUERY LOCATION ;
You were prompted because STEP ended.
The program is currently entering block COBVALU.
```

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). (Remember that Debug Tool starts **after** program initialization but **before** symbolic COBOL or static C++ variables have been initialized, so you cannot view or modify the contents of variables until you have performed a step or run.) The value is displayed in the Log window. This is equivalent to entering LIST TITLED variable on the command

line. For instance, run the COBCALC program to the statement labeled **CALC1**. To run to the statement labeled **CALC1**, enter AT 46 ; G0 ; on the Debug Tool command line. Move the cursor over INPUT-1 and press LIST (PF4). The following appears in the Log window:

```
LIST ( INPUT-1 ) ;
INPUT-1 = 'LOAN'
```

To modify the value of INPUT-1, enter on the command line:

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

You can enter most COBOL expressions on the command line.

Now step into the call to COBVALU by pressing STEP (PF2) and step until the statement labeled **VALU2** is reached. To view the attributes of the variable INTEREST, issue the Debug Tool command:

```
DESCRIBE ATTRIBUTES INTEREST ;
```

The result in the Log window is:

```
ATTRIBUTES FOR INTEREST
  ITS LENGTH IS 4
  ITS ADDRESS IS 00011DC8
  02 COBVALU:>INTEREST  S999V99 COMP
```

You can use this action as a simple browser for group items and data hierarchies. For example, you can list all the values of the elementary items for the CHAR-DATA group with the command:

```
LIST CHAR-DATA ;
```

with results in the Log window appearing something like this:

```
LIST CHAR-DATA ;
02 COBVALU:>INPUT-1 of 01 COBVALU:>CHAR-DATA = '.12 5'
Invalid data for 02 COBVALU:>PAYMENT-OUT of 01 COBVALU:>CHAR-DATA is found.
02 COBVALU:>INTEREST-IN of 01 COBVALU:>CHAR-DATA = '.12'
02 COBVALU:>NO-OF-PERIODS-IN of 01 COBVALU:>CHAR-DATA = '5'
02 COBVALU:>INPUT-BUFFER of 01 COBVALU:>CHAR-DATA = '5069837544'
SUB(1) of 02 COBVALU:>BUFFER-ARRAY of 01 COBVALU:>CHAR-DATA = '50'
SUB(2) of 02 COBVALU:>BUFFER-ARRAY of 01 COBVALU:>CHAR-DATA = '69'
SUB(3) of 02 COBVALU:>BUFFER-ARRAY of 01 COBVALU:>CHAR-DATA = '83'
SUB(4) of 02 COBVALU:>BUFFER-ARRAY of 01 COBVALU:>CHAR-DATA = '75'
SUB(5) of 02 COBVALU:>BUFFER-ARRAY of 01 COBVALU:>CHAR-DATA = '44'
```

Note: If you use the LIST command to list the contents of an uninitialized variable, or a variable that contains invalid data, Debug Tool 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 or equal to -1 (before the exception occurs). First run COBCALC, step into COBVALU, and stop at the statement labeled **VALU1**. To accomplish this, issue these Debug Tool commands at the start of COBCALC:

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

Now set the breakpoint like this:

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

Line 44 is the statement labeled **VALU3**. The command causes Debug Tool to stop at line 44. If the value of INTEREST is greater than -1, the program continues. The command causes Debug Tool to remain on line 44 only if the value of INTEREST is less than or equal to -1.

To force the discount rate to be negative, enter the Debug Tool command:

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

Run the program by issuing the GO command. Debug Tool will halt the program at line 44. Display the contents of INTEREST by issuing the LIST INTEREST command. To view the effect of this breakpoint when the discount rate is positive, begin a new debug session and repeat the Debug Tool commands shown in this section. However, do not issue the MOVE '-2 5' TO INPUT-1 command. The program execution will not be stopped at line 44 and the program will run to completion.

Debugging when only a few parts are compiled with TEST

Suppose you want to set a breakpoint at entry to COBVALU. COBVALU has been compiled with TEST but the other programs have not. Debug Tool comes up with an empty Source window. You can use the LIST NAMES CUS command to determine if the COBVALU compile unit is known to Debug Tool and then set the appropriate breakpoint using either the AT APPEARANCE or AT ENTRY command. See also “Setting a breakpoint to halt when certain routines are called” on page 70 for more information on setting breakpoints.

Instead of setting a breakpoint at entry to COBVALU in this example, you could issue a STEP command when Debug Tool initially displays the empty Source window. Debug Tool runs the program until it reaches the entry for the first routine compiled with TEST, COBVALU in this case.

Capturing I/O to the system console

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

```
SET INTERCEPT ON CONSOLE ;
```

For example, if you run COBCALC and issue the Debug Tool SET INTERCEPT ON CONSOLE command, followed by the STEP 3 command, you will see the following output displayed in the Debug Tool Log:

```
SET INTERCEPT ON CONSOLE ;  
STEP 3 ;  
CONSOLE : CALC Begins.
```

The CALC Begins. phrase is displayed by the statement DISPLAY "CALC Begins." UPON CONSOLE in COBCALC.

The SET INTERCEPT ON CONSOLE command not only captures output to the System Console, but also allows you to input data from your Debug Tool terminal instead of the System Console by using the Debug Tool INPUT command. For example, if the next COBOL statement executed is ACCEPT INPUT-DATA FROM CONSOLE, the following message appears in the Debug Tool Log:

CONSOLE : IGZ0000I AWAITING REPLY.

The program is waiting for input from CONSOLE.

Use the INPUT command to enter 114 characters for the intercepted fixed-format file.

Continue execution by replying to the input request by entering the following Debug Tool command:

INPUT some data ;

Note: Whenever Debug Tool intercepts System Console I/O, and for the duration of the intercept, the display in the Source window is empty and the Location field in the session panel header at the top of the screen shows *Unknown*.

Displaying raw storage

You can display the storage for a variable by using the LIST STORAGE command. For example, to display the storage for the first 12 characters of BUFFER-DATA enter:

LIST STORAGE(BUFFER-DATA,12)

Getting a routine traceback

Often when you get close to a programming error, you want to know how you got into that situation, and especially what the traceback of calling routines is. To get this information, issue the command:

LIST CALLS ;

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

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

the log contains something like:

```
AT APPEARANCE COBVALU
  AT ENTRY COBVALU ;
GO ;
GO ;
LIST CALLS ;
At ENTRY in COBOL program COBVALU.
From LINE 67.1 in COBOL program COBCALC.
```

which shows the traceback of callers.

Tracing the run-time path for 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 or data set and USE them when Debug Tool initially displays your program. Assuming you have a PDS member, USERID.DT.COMMANDS(COBCALC), that contains the following Debug Tool commands:

- * Commands in a COBOL USE file must be coded in columns 8-72.
- * If necessary, commands can be continued by coding a '-' in
- * column 7 of the continuation line.

```
01 LEVEL PIC 99 USAGE COMP;
MOVE 1 TO LEVEL;
AT ENTRY * PERFORM;
    COMPUTE LEVEL = LEVEL + 1;
    LIST ( "Entry:", LEVEL, %CU);
    GO;
    END-PERFORM;
AT EXIT * PERFORM;
    LIST ( "Exit:", LEVEL);
    COMPUTE LEVEL = LEVEL - 1;
    GO;
    END-PERFORM;
```

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

```
USE USERID.DT.COMMANDS(COBCALC)
```

If, after executing the USE file, you run COBCALC, the following trace (or something similar) is displayed in the Log window:

```
ENTRY:
LEVEL = 00002
%CU = COBCALC
ENTRY:
LEVEL = 00003
%CU = COBLOAN
EXIT:
LEVEL = 00003
ENTRY:
LEVEL = 00003
%CU = COBVALU
EXIT:
LEVEL = 00003
ENTRY:
LEVEL = 00003
%CU = COBVALU
EXIT:
LEVEL = 00003
EXIT:
LEVEL = 00002
```

If you do not want to create the USE file, you can enter the commands through the command line, and the same effect is achieved.

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.

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

Find the address of FIELD-2 with the command:

```
DESCRIBE ATTRIBUTES FIELD-2
```

Suppose the result is X'0000F559'. To set a breakpoint which watches for a change in storage values starting at that address for the next 8 bytes, issue the command:

```
AT CHANGE %STORAGE(H'0000F559',8)
```

When the program 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 79. 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.

Before running PLICALC, you need to allocate SYSPRINT to the terminal. For MVS under TSO, enter the following command:

```
ALLOC FI(SYSPRINT) DA(*)
```

For VM, enter the following command:

```
FILEDEF SYSPRINT TERMINAL
```

```

plicalc: proc options(main);
/*-----*/
/*
/* 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 1 stack,
      2 stkptr fixed bin(15,0) init(0),
      2 stknum(50) fixed bin(31,0);
dcl 1 bufin,
      2 bufptr fixed bin(15,0) init(0),
      2 bufchr char (100) varying;
dcl 1 tok char (100) varying;
dcl 1 tstop char(1) init ('s');
dcl 1 ndx fixed bin(15,0);
dcl num      fixed bin(31,0);
dcl i        fixed bin(31,0);
dcl push entry external;
dcl pop  entry returns (fixed bin(31,0)) external;
dcl readtok entry returns (char (100) varying) external;
/*-----*/
/* input  action:
/*  2      push 2 on stack
/*  18     push 18
/*  +      pop 2, pop 18, add, push result (20)
/*  =      output value on the top of the stack (20)
/*  5      push 5
/*  /      pop 5, pop 20, divide, push result (4)
/*  =      output value on the top of the stack (4)
/*-----*/
bufchr = '2 18 + = 5 / =';
do while (tok ~= tstop);
  tok = readtok(bufin);          /* get next 'token' */
  select (tok);
    when (tstop)
      leave;
    when ('+') do;
      num = pop(stack);
      call push(stack,num);      /* CALC1 statement */
    end;
    when ('-') do;
      num = pop(stack);
      call push(stack,pop(stack)-num);
    end;
  end;
end;

```

Figure 5 (Part 1 of 2). Sample PL/I Program - Main Program PLICALC

```

    when ('*')
        call push(stack,pop(stack)*pop(stack));
    when ('/') do;
        num = pop(stack);
        call push(stack,pop(stack)/num); /* CALC2 statement */
    end;
    when ('=') do;
        num = pop(stack);
        put list ('PLICALC: ', num) skip;
        call push(stack,num);
    end;
    otherwise do; /* must be an integer */
        num = atoi(tok);
        call push(stack,num);
    end;
end;
return;

```

Figure 5 (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 6. Sample PL/I Program - TOK Function

```

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

```

Figure 7. Sample PL/I Program - PUSH Function

```
pop: procedure(stack) returns (fixed bin(31,0));
/*-----*/
/*
/* a simple pop function for a stack of integers
/*
/*-----*/
dcl 1 stack connected,
      2 stkpnr fixed bin(15,0),
      2 stknum(50) fixed bin(31,0);
stkptr = stkpnr - 1;
return (stknum(stkpnr+1));
end pop;
```

Figure 8. Sample PL/I Program - POP Function

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

Figure 9 (Part 1 of 2). Sample PL/I Program - READTOK Function

```

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 9 (Part 2 of 2). 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 the AT ENTRY command, you must compile your program with the 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. The Debug Tool Log will display something similar to:

```
QUERY LOCATION ;
```

You are executing commands in the ENTRY READTOK breakpoint. The program is currently entering blocvk READTOK.

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**. To run to the statement labeled **CALC1**, enter AT 22 ; GO ; on the Debug Tool command line. Move the cursor over NUM and press LIST (PF4). The following appears in the Log window:

```
LIST NUM ;
NUM =                18
```

To modify the value of NUM to 22, overwrite the NUM = 18 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 0003944C AND ITS LENGTH IS 200
  PUSH : STACK.STKNUM(50) FIXED BINARY(31,0) REAL PARAMETER
  ITS ADDRESS IS 0003944C AND ITS LENGTH IS 4
```

You can list all the values of the members of the structure pointed to by STACK with the command:

```
LIST STACK;
```

with results in the Log window appearing something like this:

```
LIST STACK ;
STACK.STKPTR =          2
STACK.STKNUM(1) =          2
STACK.STKNUM(2) =          18
STACK.STKNUM(3) =        233864
...
STACK.STKNUM(50) =       121604
```

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

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

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 31 DO; IF NUM ^= 0 THEN GO; END;
```

Line 31 is the statement labeled **CALC2**. The command will cause Debug Tool to stop at line 31. If the value of NUM is not 0, the program will continue. The command causes Debug Tool to stop on line 31 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. 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 17.1 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 or data set and USE them when Debug Tool initially displays your program. Assuming you have a PDS member, USERID.DT.COMMANDS(PLICALL), 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 USERID.DT.COMMANDS(PLICALL)
```

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

```
PLICALL: PROC OPTIONS(MAIN);
...
CALL PLISUB ;
...
END PLICALL;

PLISUB: PROCEDURE;
...
CALL PLISUB1 ;
...
END PLISUB;

PLISUB1: PROCEDURE;
...
CALL PLISUB2 ;
...
END PLISUB1;

PLISUB2: PROCEDURE;
...
END PLISUB2;
```

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

```
'>PLICALL      '
' >PLISUB      '
' >PLISUB1     '
' >PLISUB2     '
' <PLISUB2     '
' <PLISUB1     '
' <PLISUB      '
'<PLICALL      '
```

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 the following example where the program changes more than the caller expects it to change.

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

Find the address of FIELD2 with the command:

```
DESCRIBE ATTRIBUTES FIELD2
```

Suppose the result is X'00521D42'. To set a breakpoint 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 of them to view simultaneously:

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

See Figure 12 on page 86 for an example of a Debug Tool screen containing all three windows.

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 12 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 another (navigate), and how to arrange their appearance and content.

Session Panel header fields

Figure 10 and Figure 11 on page 85 show two examples. The first is a header for a program under MVS/TSO, and the second is a header for a program under CMS. Descriptions of the specific areas follow the figures.

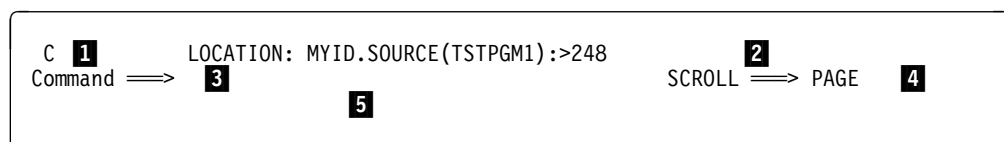


Figure 10. Session Panel Header Fields for a C Program under TSO

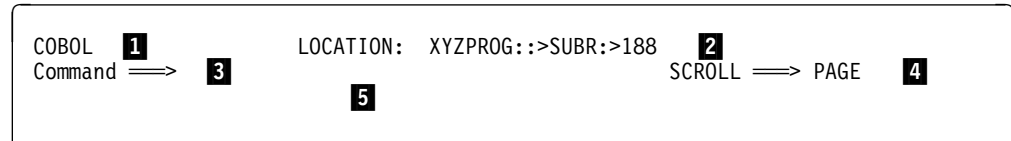


Figure 11. Session Panel Header Fields for a COBOL Program under CMS

1 C/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.

Note: Debug Tool does not differentiate between C and C++ programs, so if there is a C++ program in the source window, only C will be displayed in this field.

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 MYID.SOURCE(TSTPGM1) is suspended at line 248. In the second example, execution in XYZPROG is suspended at XYZPROG::>SUBR:>118, or line 118 of subroutine SUBR.

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 12 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: IBTUF4 :> 100.1
Command ==>
MONITOR --+---1---+---2---+---3---+---4---+---5---+---6 LINE: 1 OF 3
***** TOP OF MONITOR *****
0001 1 77 IBTUF4:>VARBL2    21
0002 2 77 IBTUF4:>VARBL1    11
0003 3 77 IBTUF4:>X        1
***** BOTTOM OF MONITOR *****
SOURCE: IBTUF4 --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
0017     LIST VARBL1 ;
0018 MONITOR
0019     LIST X ;

```

Figure 12. 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 91.

Source Display Area: The source display area shows the source code (for a C/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 line mode, and their responses, are automatically appended to the Log window except the full-screen commands PANEL, FIND, CURSER, 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 at 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/C++ or PL/I programs):

- All command output

- Commands from USE files
- Commands specified on a `__ctest()` function call
- Commands specified on a `CALL CEETEST` statement
- Commands specified on a `CALL PLITEST` statement
- Commands specified in the run-time TEST command string suboption
- QUIT commands
- Debug Tool messages about the program execution (intercepted console messages, exceptions, etc.)

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

Creating the Log file

To create the log file in Debug Tool, allocate a ddname INSPLOG in the CLIST, JCL, or exec you use to run your program. This ddname must have a logical record length greater than or equal to 32 and less than or equal to 256. If the LRECL is outside these limits, Debug Tool issues a message and does not create a log file. On MVS, the log file must be a sequential data set. The record format and blocksize have no restrictions.

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.

Note: Under CICS, SET LOG OFF is the default. To start the log, you must issue: SET LOG ON FILE TSTPINE.DT.LOG;.

Make sure the default of SET LOG ON is still in effect. If you have issued SET LOG OFF, output to the log file is suppressed. If Debug Tool is never given control, the log file is not created.

When the default log file (INSPLOG) is created during initialization, any existing file with the same name will be overwritten. However, entering the SET LOG ON FILE xxx command will append the log output to the existing file.

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 logddn;
```

This creates the log file LOGDDN, after you allocate a file to ddname LOGDDN.

Note: Do not use MVS partitioned data sets to store session logs.

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 all modes of operation: line mode, 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 13.

```

C          LOCATION: "ICFSSCU1" :> 89
Command ==> 1
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;
84     int VARBL2 = 20;
85     int R = 1;
86
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 ;
0014 STEP ;
0015 STEP ;

```

Figure 13. 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 ICCFSSCU2 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.

Entering commands

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

Using the command line

You can type any Debug Tool command in this field, as well as any CMS or TSO command that is prefixed by `SYSTEM`, `CMS`, or `TSO`. Commands can be up to 48 SBCS characters or 23 DBCS characters in length. If you need to enter a lengthy command, Debug Tool provides a command continuation character, the SBCS hyphen (-). When the current programming language is C/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 it was begun with a left brace ({) that has not been matched by a right brace (}). If you do need to continue your command, Debug Tool provides a **MORE** `====>` prompt that is equivalent to another command line. You can continue to request additional command lines with continuation characters until you complete your command.

Issuing system commands

During your Debug Tool session you can still access your base operating system using the `SYSTEM` command. The string following the `SYSTEM` command is passed on to your operating system. You can communicate with CMS in a CMS environment, or TSO in a TSO environment. For example, if you want to see a CMS filelist while in a debugging session, enter `SYSTEM FILELIST;`.

For CMS Only: If you enter `SYSTEM` without a system command, you enter CMS subset mode. To return to Debug Tool, enter `RETURN`.

For TSO Only:

- A command is required after the `SYSTEM` keyword. Do not enter any required parameters. Debug Tool prompts you.
- If you are debugging in batch and need system services, you can include commands and their requisite parameters in a CLIST and substitute the CLIST name in place of the command.
- If you want to enter several TSO commands, you can include them in a USE file, a procedure, or other commands list. Or you can enter:

`SYSTEM ISPF;`

This invokes ISPF and displays an ISPF panel on your host emulator screen that you can use to issue commands.

For CICS Only: The SYSTEM command is not supported.

The SYSTEM command has two synonyms: CMS for the CMS environment, and TSO for the TSO environment. Truncation of the CMS and TSO commands is not allowed.

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 before which they are typed. For example, the AT command typed in the prefix area of a specific line sets a statement breakpoint only at that line.

You can use prefix commands to specify the particular verb or statement in the line where you want the command to apply. For example, AT typed in the prefix area 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, system keywords such as CMS and TSO, 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, and SYSTEM can be abbreviated only as SYS.

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 previous to that, then the command previous 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 311.

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`, and the cursor is already in the Source window, issue the following command:

```
FIND "var1";
```

- If the cursor is not in the Source window, 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 Source 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 listing or source file is found.

- If you want to search the Source window for the next occurrence of `var1`, just enter:

```
FIND
```

You do not need to provide the variable name, because the Debug Tool remembers the string you last searched for. Again, the Source 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 14. The configuration panel displays six possible ways you can change your Debug Tool session panel window placements.

Window Layout Selection Panel

Command ==>

1

M

S

L

2

--

-

3

-

-

--

4

5

--

-

6

--

-

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

Legend:

L - Log
 M - Monitor
 S - Source

To reassign the Source, Monitor, and Log windows, type over the current settings or underscores with L, M, or S.

Figure 14. 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 14.

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 14, configuration **1** is the chosen 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 rows 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 14) 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 14), 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 14), 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 15 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 ==>				
Title	: field headers	TURQ	NONE	HIGH
	output fields	GREEN	NONE	LOW
Monitor:	contents	TURQ	REVERSE	LOW
	line numbers	TURQ	REVERSE	LOW
Source	: listing area	WHITE	REVERSE	LOW
	prefix area	TURQ	REVERSE	LOW
	suffix area	YELLOW	REVERSE	LOW
	current line	RED	REVERSE	HIGH
	breakpoints	GREEN	NONE	LOW
Log	: program output	TURQ	NONE	HIGH
	test input	YELLOW	NONE	LOW
	test output	GREEN	NONE	HIGH
	line numbers	BLUE	REVERSE	HIGH
Command line		WHITE	NONE	HIGH
Window headers		GREEN	REVERSE	HIGH
Tofeof delimiter		BLUE	REVERSE	HIGH
Search target		RED	NONE	HIGH
Enter	QUIT to return with current settings saved.			
	CANCEL to return without current settings saved.			

Valid Color:
White Yellow Blue
Turq Green Pink Red

Valid Intensity:
High Low

Valid Highlight:
None Reverse
Underline Blink

Color and Highlight
are valid only with
color terminals.

Figure 15. Color Selection Panel with Default Settings

Initially, the session panel areas and fields have the default color and attribute values shown in Figure 15.

To change the color and attribute settings for your Debug Tool session, enter the desired colors or attributes over the existing values of the fields you want to change. The changes you make are saved when you enter `QUIT`.

You can also change the colors or intensity of selected areas by issuing the equivalent `SET COLOR` command from the command line. Either specify the fields explicitly, or use the cursor to indicate what you want to change. Changing a color or highlight with the equivalent `SET` command changes the value on the Color Selection Panel.

Settings remain in effect for the entire debug session.

To preserve any changes you make to the default color fields, specify a file before you begin your session using the `ddname inspsafe` and the `dsname` or `fileid` of

your choice. Debug Tool recognizes any file with this ddname as the file where it saves session panel settings for use during subsequent sessions. If you do not allocate this file before your session, Debug Tool begins the next debugging session with the values shown in Figure 15.

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 16 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 PDS name(MVS only)		
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 16. 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 PDS name

If specified, the data set where Debug Tool looks for the source/listing. This field appears only if you are debugging on MVS. 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/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 16.

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 linemode 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 default profile settings, specify a file before you begin your session using the ddname `inspsafe` and the dsname or fileid of your choice. Debug Tool recognizes any file with this ddname as the file where it saves session panel settings for use during subsequent sessions. 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 you do not allocate this file before your session, Debug Tool begins the next debugging session with the values shown in Figure 16.

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 processes and enclaves

This chapter discusses invocation of Debug Tool with more than one process, debugging multiple enclaves, and using Debug Tool features with multiple enclaves.

The following topics are covered in this chapter:

- Debugging applications within multiple enclaves
- Invoking Debug Tool within an enclave
- Using the source window and related windows
- 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

Debugging applications within multiple enclaves

There is a single Debug Tool session across all enclaves in a process. Breakpoints set in one process are restored when the new process begins in the new session.

Invoking Debug Tool within an enclave

Once an enclave in a process activates Debug Tool, it remains active throughout subsequent enclaves in the process, regardless of whether the run-time options for the enclave specify TEST or NOTEST. Debug Tool retains the settings specified from the TEST run-time option for the enclave that activated it, until you modify them with SET TEST (see “SET TEST” on page 326). If your Debug Tool session includes more than one process, the settings for TEST are reset according to those specified on the run-time TEST option of the first enclave that activates Debug Tool in each new process.

If Debug Tool is first activated in a nested enclave of a process, and you STEP or GO back to the parent enclave, you can debug the parent enclave. However, if the parent enclave contains COBOL but the nested enclave does not, Debug Tool is not active for the parent enclave, even upon return from the child enclave.

Upon activation of Debug Tool, the initial commands string, primary commands file, and the preferences file are run. They run only once, and affect the entire Debug Tool session. A new primary commands file cannot be invoked for a new enclave.

Using the source window and related windows

A particular enclave's Source or Listing windows and their related windows (Compact Source, Local Breakpoint, and Local Monitor 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. Unless you clear or disable this breakpoint, it will be triggered when the process finishes execution. During run time of a termination breakpoint, G0 and STEP are valid commands that cause your program to continue running the next process in the series.

Ending a Debug Tool session

You cannot specify NOPROMPT as the third suboption in the TEST run-time option for the next process on the host. This is to ensure that STATEMENT/LINE, ENTRY, EXIT, and LABEL breakpoints are properly restored when the next process starts. If you have not used these breakpoint types, you can specify NOPROMPT.

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 Language Environment message CEE2529S. The system is attempting to cleanly terminate all enclaves in the process.

Normally, the condition causes the current enclave to terminate. Then, the same condition will be raised in the parent enclave, which will also terminate. This continues until all enclaves in the process have been terminated. As a result, you will see a CEE2529S message for each enclave that is terminated.

There is one case where Debug Tool raises Language Environment 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 CMS Only: Under CMS, an unhandled condition in a nested enclave causes an Language Environment abend 4094 with reason code 40.

or CICS and MVS Only: Under CICS and MVS, an abend appears on the application terminal. For Language Environment 1.3 it is 4038. An abend at termination of a nested enclave is normal and should be expected.

Using Debug Tool commands within multiple enclaves

Some Debug Tool commands and variables have a specific scope for enclaves and processes. Table 2 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 205.

Table 2 (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		<p>Applies to all systems except MVS batch and MVS with TSO. Under MVS batch and MVS with TSO, LIST CALLS lists the call chain for the current active thread in the current active enclave.</p> <p>For programs containing interlanguage communication (ILC), routines from previous enclaves are only listed if they are coded in a language that is active in the current enclave.</p> <p>Also lists compile units in parent enclaves under CMS if the enclave was created using view SVC LINK. If the enclave was created with the system() function or the CMSCALL macro, compile units in parent enclaves will not be listed.</p> <p>Note: Only compile units in the current thread will be listed for PL/I multitasking applications.</p>
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. In the Debug Frame window, compile units in parent enclaves are marked as deactivated.

Multiple processes and enclaves

Table 2 (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
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 Language Environment or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.
SET EQUATE ¹		X	
SET INTERCEPT ¹		X	For C, intercepted streams or files cannot be part of any C I/O redirection during the execution of a nested enclave. For example, if stdout is intercepted in program A, program A cannot then redirect stdout to stderr when it does a system() call to program B. Also, not supported for PL/I.
SET NATIONAL LANGUAGE ¹	X		This setting affects both your application and Debug Tool. At the beginning of an enclave, the settings are those provided by Language Environment or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.
SET PROGRAMMING LANGUAGE ¹	X		Applies only to programming languages in which compile units known in the current enclave are written (a language is "known" the first time it is entered in the application flow).
SET QUALIFY ¹	X		Can only be issued for load modules, compile units, and blocks that are known in the current enclave.
SET TEST ¹		X	
TRIGGER condition ²	X		Applies to triggered conditions. ² Conditions can be either an Language Environment symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.
TRIGGER AT	X	X	In addition to triggering breakpoints set in the current enclave, TRIGGER AT can trigger global breakpoints.
Note: <ol style="list-style-type: none"> SET commands other than those listed in this table affect the entire Debug Tool session. 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 line or batch mode
- Running multitasking programs with Debug Tool.
- Debugging ISPF applications
- Programming considerations for DB2, IMS, and CICS programs
- Requirements when using Debug Tool with DB2, IMS, and CICS programs
- Suggestions on invoking Debug Tool in DB2, IMS, and CICS environments
- Using Debug Tool CICS Interactive Run-Time Facility (DTCN)

Using Debug Tool in line mode

If you only have access to a typewriter-like terminal, you need to use Debug Tool in line mode.

Note: Line mode is not supported in CICS.

To start a line-mode Debug Tool session, make sure the setting of SCREEN is off by specifying it in either your primary commands file, preferences file, or the initial command string included in the TEST run-time option. Then follow the steps outlined in “Invoking your program for a debugging session” on page 29 to begin a Debug Tool session in CMS or MVS with TSO. If you are using a terminal that does not support a full-screen session, Debug Tool defaults to line mode.

Debug Tool issues a message indicating that execution has begun.

After control is given to Debug Tool, it displays the following prompt when it is ready for a command:

TEST:

or

TEST (qualify:>location):

where qualify:>location is replaced by Debug Tool's current location in the program. The prompt used depends on the current PROMPT setting (SHORT or LONG). Enter your commands at the prompt.

If you need to continue a command, use the command continuation character, the hyphen (-), and the prompt is replaced by the word PENDING.... When you are finished with Debug Tool in line mode, end your session by entering QUIT.

Commands you can use in line mode

You can use most, but not all, Debug Tool commands in line mode. The commands that you cannot use are those designed to control your full-screen session, such as PANEL commands, WINDOW commands, and cursor-sensitive commands. For more information on the commands and their appropriate usage, see Chapter 12, “Using Debug Tool commands” on page 193.

To help make line-mode debugging more efficient, use the LIST command to list source statements. See “LIST command” on page 274 for more information.

Getting HELP during a line-mode session

Online command syntax help is available for each Debug Tool command, similar to the help described in “Getting help during your session” on page 101.

You must issue a separate request for syntax help for each command.

Using Debug Tool in batch mode

Debug Tool can run in batch mode, creating a noninteractive session.

In batch mode, Debug Tool receives its input from either the primary commands file, the USE file, or the command string specified in the run-time TEST option, and writes its normal output to a log file.

Note: You must allocate a data set for the 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 an application in its native batch environment, such as MVS/JES or CICS batch.

When Debug Tool is reading commands from a specified data set or file and no more commands are available in that data set or file, it forces a GO command until the end of the program is reached.

When debugging in batch mode, use QUIT to end your session.

Running multitasking programs with Debug Tool

You can run your multitasking programs with Debug Tool. When more than one task is involved in your program, Debug Tool might be invoked by any or all of them. Because conflicting use of the terminal or log file, for example, could occur if Debug Tool is operating on multiple tasks, its use is single-threaded. So, if your program runs as two tasks (task A and task B) and task A calls Debug Tool, Debug Tool accepts the request and begins operating on behalf of task A. If, during that period, task B calls Debug Tool, the request from task B is held until the request from task A is complete (for example, you issued a STEP or GO command). Debug Tool is then released and can accept any pending invocation.

See *OS/390 Language Environment Programming Guide*, for more information.

MVS/ESA SP V5R1 with OpenEdition R2 requirement

MVS/ESA SP V5R1 9 (or later) with OpenEdition (OE) R2 is required to run multitasking applications. The OpenEdition R2 must be installed and activated in order to run multitasking applications.

The OpenEdition R2 is needed so that the POSIX-defined multithreading functions can be used to support multitasking.

Restrictions when debugging multitasking applications

The following is a list of restrictions when debugging multitasking applications:

- Debugging applications that create another process because of conflicting use of the terminal.
- Only variables and symbol information for compile units in the task currently being debugged are accessible.
- The LIST CALL command only provides a traceback of the compile units in the current task.
- The source file can reside on an HFS file system, but executables that are stored on an HFS file system cannot be debugged.

Debugging ISPF applications

When debugging ISPF applications or applications using line mode I/O, issue the SET REFRESH ON command.

This command is executed and is displayed in the log output area of the Command/Log window. Note that SET REFRESH ON modifies the Debug Tool environment. Consequently, the REFRESH setting is saved in the preferences file (inspsafe), and it is preserved between Debug Tool invocations. So, you only need to specify it once; Debug Tool uses the same setting on subsequent invocations.

For general information about defining the Debug Tool environment and the preferences file, see “Customizing your session” on page 95.

When you are debugging ISPF applications, Debug Tool and the application share the same emulator session. Consequently, it is necessary to press PA2 after each ISPF panel display. PA2 refreshes the ISPF application panel and removes residual Debug Tool output from the emulator session. This is necessary only if Debug Tool sends output to the emulator session between ISPF application panel displays.

Debugging DB2 programs

When you are planning to use Debug Tool to debug your DB2 programs, certain steps need to be taken. These steps are described in detail below.

Programming considerations

There are no special coding techniques for any DB2 programs you might want to debug using Debug Tool. For details on how to code your program to access a DB2 database, see the HLL Programming Guides and the *IBM DATABASE 2 Application Programming Guide*.

To communicate with DB2, you should:

- Delimit SQL statements with EXEC SQL and END-EXEC statements
- Declare SQLCA in working storage
- Declare the host variables
- Code the appropriate SQL statements
- Test the DB2 return codes

Program preparation

Program preparation includes the DB2 precompiler, the compiler, the Language Environment prelinker, the linkage editor, and DB2 bind. For C/C++, Debug Tool uses, as the program source file, the input to the compiler (the output from the DB2 precompiler). The precompiler output must be saved in a permanent data set.

Precompile requirements

Before your program can be compiled, the SQL statements must be prepared using the DB2 precompiler. For details about the precompiler, see *IBM DATABASE 2 Application Programming Guide*. No special preparations are needed in the precompile 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 host language code. The modified source output from the preprocessor contains the original SQL statements in comment form. For this reason, the source or listing view displayed during a debugging session can look very different from the original source.
- The host language code inserted by the SQL preprocessor 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 precompiler 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 5
- “Compiling a C++ program with the compile-time TEST option” on page 9
- “Compiling a COBOL program with the compile-time TEST option” on page 10
- “Compiling a PL/I program with the compile-time TEST option” on page 13

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 your source (if you are working with C/C++ language) or listing (if you are working with COBOL or PL/I) is stored in a permanent data set that is available to Debug Tool.

Link requirements

The output from the compiler must be linked into your program load library. You can include the user run-time options module, CEEUOPT, by doing the following:

1. Find the user run-time options program CEEUOPT in the Language Environment SCEESAMP library.
2. Change the NOTEST parameter into a default TEST parameter:
 old: NOTEST=(ALL,*,PROMPT,INSPREF),
 new: TEST=(*,*,*,*),
3. Assemble the CEEUOPT program and keep the object code.
4. Link-edit the CEEUOPT object code with any program to invoke Debug Tool.

See “Using alternative Debug Tool invocation methods” on page 31 for more information.

Figure 17 on page 112 shows the modified assembler program, 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.

If your system programmer has not already done so, include all the proper libraries in the SYSLIB concatenation. For example, the ISPLOAD library for ISPLINK calls, and the DB2 DSNLOAD library for the DB2 interface modules (DSNxxxx).

Bind requirements

Before you can run your DB2 program, you must run a DB2 bind in order to bind your program with the relevant DBRM output from the precompiler step. No special requirements are needed for Debug Tool.

Using Debug Tool with DB2 programs

You can debug DB2 programs 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 command input 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, either by STEPLIB or through the LINKLIB.
2. Provide all the data set definitions in the form of DD statements (example: Log, Preference, list, and so on).

```

*/*****
*/*    LICENSED MATERIALS - PROPERTY OF IBM    */
*/*    5688-198 (C) COPYRIGHT IBM CORP. 1994.  ALL RIGHTS RESERVED. */
*/*    SEE COPYRIGHT INSTRUCTIONS.            */
*/*****/
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
CEEUOPT ABPERC=(NONE),
        AIXBLD=(OFF),
        ALL31=(OFF),
        ANYHEAP=(32K,16K,ANYWHERE,FREE),
        BELOWHEAP=(32K,16K,FREE),
        CBLOPTS=(ON),
        CBLPSHPOP=(ON),
        CBLQDA=(ON),
        CHECK=(ON),
        COUNTRY=(US),
        DEBUG=(ON),
        ERRCOUNT=(20),
        HEAP=(64K,64K,ANYWHERE,KEEP,16K,16K),
        INTERRUPT=(OFF),
        LIBSTACK=(32K,16K,FREE),
        MSGFILE=(SYSOUT),
        MSGQ=(15),
        NATLANG=(ENU),
        TEST=(*,*,*,*),
        RPTOPTS=(OFF),
        RPTSTG=(OFF),
        RTEREUS=(OFF),
        SIMVRD=(OFF),
        STACK=(512K,512K,BELOW,KEEP),
        STORAGE=(NONE,NONE,NONE,8K),
        TERMTHDACT=(MSG),
        TRAP=(ON),
        UPSI=(00000000),
        VCTRSAVE=(OFF),
        XUFLOW=(OFF)
DC      C'5688-198 (C) COPYRIGHT IBM CORP. 1994'
DC      C'LICENSED MATERIAL - PROGRAM PROPERTY OF IBM'
END

```

Figure 17. Run-time options module CEEUOPT

3. Specify your debug commands in the command input file.
4. Run your program through the TSO batch facility.

Interactive mode

In this mode, you can decide at debug time what debugging commands you want issued during the test.

Using TSO Commands

1. Ensure that either you or your system programmer has allocated all the required data sets through a CLIST or REXX EXEC.
2. Issue the DSN command to invoke DB2.
3. Issue the RUN subcommand to execute your program. The run-time TEST option can be specified as a parameter on the RUN subcommand. An example for a COBOL program is:

```
RUN PROG(program) PLAN(planname) LIB('user.library')
PARMS('/TEST(*,*,*,*)')
```

Using TSO/Call Access Facility (CAF)

1. Link-edit the CAF language interface module DSNALI with your program.
2. Ensure that the data sets required by Debug Tool and your program have been allocated through a CLIST or REXX procedure.
3. Issue the TSO CALL command CALL 'DSN230.RUNLIB.LOAD(name of your program)', to start your program. DSN230 is a default high-level qualifier and DB2 might be installed elsewhere on your system. Include the run-time TEST option as a parameter in this command.

After your program has been initiated, debug your program by issuing the required Debug Tool commands.

Note: If your source does not come up in Debug Tool when you launch it, check that the listing or source file name corresponds to the MVS library name, and that you have at least read access to that MVS library. For more information see “PANEL command (full-screen mode)” on page 291, “SET DEFAULT LISTINGS (MVS)” on page 309, and “SET SOURCE” on page 324.

The program listing or source that Debug Tool displays and uses for the debugging session is the output from the compile step, and thus includes all the DB2 expansion code produced by the DB2 precompiler.

For more information on accessing the correct DB2 program library, and on using the TSO and call attachment facilities, see *IBM DATABASE 2 Administration Guide*.

Debugging IMS programs

When testing IMS online transaction programs, use the Batch Terminal Simulator (BTS) Full-Screen Image Support (FSS) to display your MFS screen formats on the TSO terminal. This enables you to enter data on-screen in the same way as it would be entered in IMS.

FSS is the default option when BTS is started in the TSO foreground, and is only available when you are running BTS in the TSO foreground. FSS can only be turned off by specifying TSO=NO on the .O command. When running in the TSO foreground, all call traces are displayed on your TSO terminal by default. This can be turned off by parameters on either the .O or .T commands. For more details on BTS and the FSS facility, see *IMS/VS Batch Terminal Simulator Program Reference and Operations Manual*.

Programming considerations

When using Debug Tool to debug your IMS application programs:

- 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 IMS 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/C++ language) or listing (if you are working with COBOL or PL/I) is stored in a permanent data set that is available to Debug Tool.

Link requirements

When you link your program, do the following:

- Include a run-time options module in your program link. With an IMS program, run-time options cannot be specified at program start. They must be coded and assembled in a user-defined run-time option module. For instructions on how to create the CEEUOPT run-time options module using the CEEXOPT macro, follow steps 1 to 4 on page 111 under “Link requirements.”
- When you link your COBOL program, do the following:
 - Include the IMS interface module DFSLI000 from the IMS RESLIB library.
 - Do not use the ENTRY DLITCBL linkage editor control statement.

Using Debug Tool with IMS programs

Debug Tool can be used with the IMS Batch Terminal Simulator (BTS) to debug IMS programs in one of three ways:

1. To test your IMS program interactively, use Debug Tool while running BTS in the TSO foreground. The IMS program still executes in batch; however, it invokes a CLIST which runs interactively. This is the only way to use the interactive mode of Debug Tool.
2. Run BTS as a batch job. Only the batch mode of Debug Tool can be used with BTS running as a batch job.
3. Test your program as an IMS batch job (without BTS). Only the batch mode of Debug Tool can be used without BTS.

Interactive mode

The only way to invoke Debug Tool in interactive mode is to run BTS in the TSO foreground. In interactive mode, Debug Tool commands can be entered as required.

If you want to debug an IMS batch program using the interactive mode of Debug Tool, do the following under BTS:

1. Define a *dummy* transaction code on the ./T command to initiate your program
2. Include a *dummy* transaction in the BTS input stream
3. Start BTS in the TSO foreground

Note: If your source or listing does not come up in Debug Tool when you launch it, check that the source or listing file name corresponds to the MVS library name, and that you have at least read access to that MVS library. For more information see “PANEL command (full-screen mode)” on page 291 and “SET SOURCE” on page 324.

Currently, Debug Tool can only be used to debug one iteration of a transaction at a time. When the program terminates you must close down Debug Tool before you can view the output of the transaction.

Therefore, if you use an input data set, you can only specify data for one transaction in that data set. The data for the next transaction must be entered from your TSO terminal.

A new debugging session will be started automatically for the next transaction. When using FSS, you must enter the `/*` command on your TSO terminal to terminate the BTS session.

Batch mode

You can use Debug Tool to debug IMS programs in batch mode. The debug commands must be predefined and included in one of the Debug Tool command files, or in a command string. The command string can be specified as a parameter either in the run-time TEST option, or when `CALL CEETEST` or `__ctest` is used. Although batch mode consumes fewer resources, you must know beforehand exactly which debug commands you are going to issue. When you run BTS as a batch job, the batch mode of Debug Tool is the only mode available for use.

For example, you can allocate a data set, `userid.CODE.BTSINPUT` with individual members of test input data for IMS transactions under BTS.

Under IMS, you can invoke Debug Tool in the following ways:

- Use the compiler run-time option (`#pragma runopts` for C and C++)
- Include `CEEUOPT` csect when linking your program (for C/C++)
- Use the Language Environment callable service `CEETEST` (`__ctest()` for C/C++)

Using alternative methods of command input under IMS

You can issue Debug Tool commands in different ways, depending on which mode you are running under.

In TSO/BTS, commands are interactive.

- Run-time TEST options (primary commands file, preferences file, or command string)
- line mode
- full-screen mode

Outside BTS, run-time TEST options (primary commands file, preferences file, or command string) are in batch IMS mode.

Under BTS, run-time TEST options (primary commands file, preferences file, or command string) are in BTS batch mode.

Debugging CICS programs

Before you can debug your programs under CICS, make sure your Systems Programmer has made the appropriate changes to your CICS region to support Debug Tool (see your compiler Installation Guide or Program Directory). 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/C++) must be retained in a permanent data set for Debug Tool to read when you debug your program.

Note: For C/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 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:**

This is probably the mode you will use the most. A single 3270 session is used by both Debug Tool and the application, swapping displays on the terminal as required.

As you step through your application, the terminal shows Debug Tool screens, but when an EXEC CICS SEND command is issued, that screen will be displayed. Debug Tool holds that screen on the terminal for you to review—simply press ENTER to return to a Debug Tool screen. When your application issues EXEC CICS RECEIVE, the application screen again appears, so you can fill in the screen details.

- **Dual Terminal Mode:**

This mode can be useful if you are debugging screen I/O applications. Debug Tool displays its screens on a separate 3270 session than the terminal displaying the application.

You step through the application using the Debug Tool terminal and, whenever the application issues an EXEC CICS SEND, the screen is sent to the application display terminal. Note that, if you do not code IMMEDIATE on the EXEC CICS SEND command, the buffer of data might be held within CICS Terminal Control until an optimum opportunity to send it is encountered—usually the next EXEC CICS SEND or EXEC CICS RECEIVE. When the application issues an EXEC CICS RECEIVE, the Debug Tool terminal will wait until you respond to the application terminal.

- **Interactive Batch Mode:**

Use this mode if you are debugging a transaction which does not have a terminal associated with it. The transaction continues to run without a CICS principal facility, but Debug Tool screens are displayed on a 3270 session that you name.

- **Noninteractive Batch Mode:**

In this mode, Debug Tool does not have a terminal associated to it at all. 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.

Invoking Debug Tool under CICS

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

- DTCN, a full-screen CICS transaction that allows you to dynamically modify any Language Environment run-time TEST/NOTEST option with which your application was originally link-edited. You can also use DTCN to modify other Language Environment run-time options that are not specific to Debug Tool.

DTCN is the recommended mechanism for invoking Debug Tool sessions and supports all the modes outlined in “Debug modes under CICS” on page 116.

- Language Environment CEEUOPT module link-edited into your application, containing an appropriate TEST option, which tells Language Environment to invoke Debug Tool every time the application is run.

This mechanism can be useful during initial testing of new code when you will want to run Debug Tool frequently. For information on preparing CEEUOPT in this manner, see “Preparing and using CEEUOPT to invoke Debug Tool under CICS” on page 122.

- A compile-time directive within the application, such as `#pragma runopts(test)` (for C/C++) 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 runs 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.

- CICS CEDF utility where you can invoke a debug session in Dual Terminal mode alongside CEDF, using a special option on the CEDF command.

This mechanism does not require you to change the application link-edit options or code, so it can be useful if you need to debug programs which have been compiled with the TEST option, but which do not have invocation mechanisms built into them. For information on using CEDF under CICS, see “Preparing and using CEDF to invoke Debug Tool under CICS” on page 123.

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 Language Environment user exit, CEEBXITA, into the CICS program you want to debug, using one of the following methods:

1. If your installation is not using this user exit, link-edit member EQADCCXT, which contains the CSECT CEEBXITA, from library EQAW.V1R2M0.SEQAMOD into your main program.
2. If your installation is already using CEEBXITA, request the name and location of the DTCN customized exit from your CICS system administrator and link that exit with your main program.

Once you have successfully link-edited your program, the application is ready to run. However, before you begin debugging your application, make sure you use the DTCN transaction to define a Debug Tool run-time startup profile and define the Terminal and/or Transaction Id you plan to use for debugging. Once the profile has been created, store it in the Debug Tool repository, described in “Profile repository”

on page 121. You are now ready to run your application on the Terminal Id you defined in the startup profile.

To start the DTCN utility, enter DTCN on the system command line. 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 you provide is then stored in the Debug Tool run-time startup profile repository. You can do this by pressing PF4.

DTCN screen

When you start DTCN, the initial DTCN screen with no user profile appears. The data entry fields in this screen show the default values for single terminal mode as displayed in the following example:

DTCN	DEBUG TOOL CICS Interactive Run-Time Facility	S07CICP2
Item	Choice	Possible choices
Terminal Id	==> tttt	Terminal to debug
Transaction Id	==>	Transaction to debug
Session Type	==> MFI	MFI, TCP, APPC, LU2
PWS Type	==>	VAD, CODE
Port/Sessionid	==>	TCP Port or APPC Session Identifier
Display Id	==>	
Test Option	==> Test	Test/Notest
Test Level	==> All	All/Error/None
Command File	==>	
Prompt Level	==> Prompt	Prompt/Noprompt/*/;
Preference File	==> *	
Any other valid Language Environment Options		
==>		
EQA2507E 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		

The sections that follow provide a detailed description for each area of the initial DTCN screen shown above. For help on an individual field, you can use the context sensitive help function by pressing PF1 with the cursor placed on that field.

Header area

DTCN	DEBUG TOOL CICS Interactive Run-Time Facility	S07CICP2
Item	Choice	Possible choices

The Header area contains:

- Identifier of the transaction - DTCN
- Application Id of the CICS region in which the transaction is running
- Column descriptions

Input area

The *Input Area* of the DTCN main panel is used to display and enter the data for the debugging profile. The *Input Area* is divided into four sections and is described below:

Area 1:

Terminal Id	==>	Terminal to debug
Transaction Id	==>	Transaction to debug

This section contains the Terminal Id and Transaction Id which, when concatenated together become the key used by DTCN to process debugging profiles. The definitions are:

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:

Session Type	==> MFI	MFI, TCP, APPC, LU2
PWS Type	==>	VAD, CODE
Port/Session Id	==>	TCP Port or APPC Session Identifier
Display Id	==>	

This section describes how you want to start the Debug Tool user interface in order to debug your application transaction. You can establish a debugging session using the host interface (MFI) or a cooperative session using either APPC, TCP/IP, or an LU2 communication protocol. The default is MFI.

Session Type Choose one of the following session types:

MFI Indicates that Debug Tool will initialize on a 3270 type terminal.

TCP Indicates that you will interface with Debug Tool from your workstation using the TCP/IP protocol.

APPC Indicates that you will interface with Debug Tool from your workstation using the APPC protocol.

LU2 An LU2 cooperative debugging session on the workstation with OS/2. LU2 applies **only** if you have the Workstation feature of CODE/370 installed on your OS/2 workstation.

PWS Type

Identifies which one of the following tools you plan to use when debugging your application program:

CODE You plan to use CODE/370 to debug your application

VAD You plan to use VisualAge Remote Debugger to debug your application

Port/Session Id	Allows you to have multiple workstation sessions so you can debug two or more applications at the same time.
Display Id	Depending on the <i>Session Type</i> you've selected, the <i>Display Id</i> is one of the following: <ul style="list-style-type: none">• For MFI - A CICS terminal for Dual Terminal MFI debugging• For TCP - Either an IP address or Host Name• For APPC - The LU name

Area 3:

Test Option	==> Test	Test/Notest
Test Level	==> All	All/Error/None
Command File	==>	
Prompt Level	==> Prompt	Prompt/Noprompt/*/;
Preference File	==> *	

For more detailed information on the following fields, see “Run-time TEST option syntax” on page 19. The definitions are:

TEST Option	TEST/NOTEST specifies the conditions under which Debug Tool assumes control during the initialization of your application.
Test Level	ALL/ERROR/NONE specifies what conditions need to be met for Debug Tool to gain control.
Command File	A valid fully qualified data set name specifying the primary commands file for this run. Note: Enclosing the name of the data set in single or double quotes is not allowed.
Prompt Level	PROMPT/NOPROMPT/*/; specifies whether Debug Tool is invoked at Language Environment initialization.
Preference File	A valid fully qualified data set name specifying the preference file to be used. Note: Enclosing the name of the data set in single or double quotes is not allowed.

Area 4:

Any other valid Language Environment Options
--

This area is provided to allow you to specify additional run-time options needed to debug your application.

Message line

==>
EQA2507E Show failed - profile does not exist

DTCN displays the messages at the bottom of the screen. For successful SHOW or NEXT command, DTCN displays full Language Environment options stored in the

Debugging Repository. Options longer than 79 characters are truncated, but their contents are properly displayed in the *Input Field* section. Check Appendix E, “Debug Tool Messages” on page 355 for an explanation and programmer response.

DTCN PF key definitions

PF1=HELP 2=GHELP 3=EXIT 4=ADD 5=REPLACE 6=DELETE 7=SHOW 8=NEXT 10=CLOSE DTCN

The PF keys on the DTCN main screen 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 DTCN	Deletes the profile repository with all stored debugging profiles

Profile repository

DTCN allows you to build Debug Tool run-time startup profiles that are used when you run your application. 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 that contains all the entries for the CICS region you are working in. DTCN uses a key that uniquely identifies each entry in the Profile Repository. That 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.

The scope of the Debugging Profile depends on the contents of the Terminal Id and Transaction Id. The scope of the profiles is as follows:

Terminal Id (blank) Debugging profile applies to any invocation of a given Transaction Id region wide (generic transaction debugging profile). This is used to interactively debug CICS batch transactions or troubleshooting programs run with:
(TEST(ERROR,,NOPROMPT,*))

Terminal Id, Transaction Id (blank) Any enabled transaction run on a specified terminal uses 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 Language Environment is the profile with the narrowest scope: specific for terminal and transaction, then generic terminal one, then generic transaction one.

DTCN data entry verification

DTCN performs data verification on the data you entered in the DTCN panel. When DTCN discovers an error, it places the cursor in the erroneous field and displays a message. You can use context sensitive help (PF1) to find what is wrong with the input.

Once you have entered your debug requirements and saved them, you can activate Debug Tool. Debug Tool will run according to the options you specified.

After you have finished debugging your program, use DTCN again to turn off your debug profile by pressing PF3 to exit. You do not need to remove EQADCCXT from the load module; in fact, it's a good idea to leave it there for the next time you want to invoke Debug Tool.

Modifying other options

You can dynamically change any other Language Environment options defined in your CICS installation as overrideable except the STACK option. For additional information about Language Environment options, see the various Language Environment publications or contact your CICS system programmer.

Preparing and using CEEUOPT to invoke Debug Tool under CICS

To request that Language Environment invoke Debug Tool every time the application is run, assemble a CEEUOPT module with an appropriate run-time TEST option. It is a good idea to link-edit the CEEUOPT module into a library and just add an `INCLUDE LibraryDDname(CEEUOPT-MemberName)` statement to the link-edit options when you link your application. Once the application program has been placed in the load library (and NEWCOPY'd if required), whenever it is run Debug Tool will be invoked.

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. For information on the run-time TEST option, see "Using the run-time TEST option" on page 18.

To invoke Debug Tool, simply run the application. Don't forget to remove the CEEUOPT containing your run-time TEST option when you have finished debugging your program.

Preparing and using compile-time directives to invoke Debug Tool under CICS

To request the program itself to invoke Debug Tool, add the appropriate directive to your application as described in "Invoking Debug Tool with CEETEST" on page 32 and "Specifying run-time TEST option with #pragma runopts in C and C++" on page 39. Whenever these directives are processed by your program, Debug Tool will be invoked in Single Terminal mode (this method supports only Single Terminal mode).

Preparing and using CEDF to invoke Debug Tool under CICS

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

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

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

To invoke Debug Tool, enter the CEDF transaction as follows:

```
CEDF xxxx,ON,I
```

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

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

```
TRAN
```

where TRAN is the id for the transaction being debugged.

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

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

```
CEDF T305,ON,I
```

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

```
TRAN
```

When you run your application on T305, Debug Tool is invoked on T304. Terminal T305 displays only application output, that is, a specific CICS command to write to the screen.

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 CDT# transaction is a special Debug Tool service transaction, and is not intended for activation by direct terminal input. If CDT# 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.
- CICS does not support Debug Tool line mode.
- Data definition (ddname) is not supported. All files, including the log file, USE files, and preferences file, must be referred to by their full data set names.
- The TS0, SET INTERCEPT, and SYSTEM commands cannot be used.
- 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.)

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
- Treats constants and variables

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 multiple-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 load module).
- 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 might 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 load module of a different language. See the Attribute Mapping tables in “Declarations (C/C++)” on page 247, “Declarations (COBOL)” on page 250, or “DECLARE command (PL/I)” on page 253 for more information on how session data attributes are mapped across programming languages. Attributes not shown in the tables cannot be mapped to other programming languages.

Also remember that variables with incompatible attributes cannot be accessed from another programming language.

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 might be different than 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 you fully qualify all program variables. This assures that proper context information (such as load module, block, etc.) is passed with the expression to the language run time when the statement is run. If this is not done, the context might not be the one you intended when you set the breakpoint, and the language run time might 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 supported by Debug Tool 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

HLL variables

Some variable references require language-specific evaluation, such as pointer referencing or subscript evaluation. Once again, the Debug Tool interprets each case in the manner of the HLL in question. Below is a list of some of the areas where Debug Tool accepts a different form of reference depending on the current programming language:

- Structure qualification
 - C/C++ and PL/I:** dot (.) qualification, high-level to low-level
 - COBOL:** IN or OF keyword, low-level to high-level
- Subscripting
 - C/C++:** name [subscript1] [subscript2] ...
 - COBOL and PL/I:** name(subscript1,subscript2,...)

HLL constants

You can use both string constants and numeric constants. Debug Tool accepts both types of constants in C/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 3 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 3. Descriptions of Debug Tool Variables and Their Corresponding Languages

Debug Tool Variable	C/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 might not be unique within a compile unit.
%CAADDRESS	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 load module 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 ih1 id=fprn print='%FPRn'.FPRn

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 module. If the current program is part of a fetched or called module, %LOAD contains the name of that load module.

%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/C++” on page 140 for your C program or “Using Debug Tool variables in COBOL” on page 175 for your COBOL 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.

Note: 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:

LINE
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), %STATEMENTS 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
IMS
TSO
NONE

Subsystems only occur on MVS, so %SUBSYSTEM is only valid on MVS. Listing this variable while working with CMS displays NONE.

%SYSTEM

Contains the name of the operating system supporting the program. Possible values are:

MVS
VM

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 module in a different language, the current programming language is automatically updated.

The following types of Debug Tool commands have the same syntax (or a subset of it) as the corresponding statements (if defined) in each supported programming language:

Assignment These commands allow you to assign a value to a variable or reference.

Conditional These commands evaluate an expression and control the flow of execution of Debug Tool commands according to the resulting value.

Declarations These commands allow you to declare 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 reference all variables in a program, no matter how many subroutines it contains. The assignment `x = 5` does not appear difficult for Debug Tool to process. However, if you declare `x` in more than one subroutine, the situation is no longer obvious. If `x` is not in the currently executing compile unit, you need a way to tell Debug Tool how to determine the proper `x`.

You also need a way to change the Debug Tool's point of view to allow it to reference variables it cannot currently see (that is, variables that are not within the scope of the currently executing block or compile unit, depending upon the HLL's concept of name scoping).

Qualification

Qualification is a method you can use to specify to what procedure or load module a particular variable belongs. You do this by prefacing the variable with the block, compile unit, and load module (or as many of these labels as are necessary), separating each label with a colon (or double colon following the load module specification) and a greater-than sign (`:`), as follows:

```
LOAD_NAME::>CU_NAME:>BLOCK_NAME:>object
```

This procedure, known as *explicit qualification*, lets Debug Tool know precisely where the variable is.

If required, `LOAD_NAME` is the load module name. It is required only when the program consists of multiple load modules and when you want to change the qualification to other than the current load module. `LOAD_NAME` can be the Debug Tool variable `%LOAD`.

If required, `CU_NAME` is the compile unit name. The `CU_NAME` is required only when you want to change the qualification to other than the currently qualified compile unit. `CU_NAME` can be the Debug Tool variable `%CU`.

If required, `BLOCK_NAME` is the program block name. The `BLOCK_NAME` is required only when you want to change the qualification to other than the currently qualified block. `BLOCK_NAME` can be the Debug Tool variable `%BLOCK`.

For PL/I Only:

In PL/I, the primary entry name of the external procedure is the same as the compile unit name. When qualifying to the external procedure, the procedure name of the top procedure in a compile unit fully qualifies the block. Specifying both the compile unit and block name results in an error. For example:

```
LM::>PROC1:>variable  
is valid.  
LM::>PROC1:>PROC1:>variable  
is not valid.
```

For C++ Only:

You must specify the full function qualification including formal parameters where they exist. For example:

1. For function (or block) ICCD2263() declared as void ICCD2263(void) within CU "USERID.SOURCE.LISTING(ICCD226)" the correct block specification for C++ would include the parenthesis () as follows:

```
qualify block %load::>"USERID.SOURCE.LISTING(ICCD226)":>ICCD2263()
```

2. For CU ICCD0320() declared as int ICCD0320(signed long int SVAR1, signed long int SVAR2) the correct qualification for AT ENTRY is:

```
AT ENTRY "USERID.SOURCE.LISTING(ICCD0320)":>ICCD0320(long,long)
```

Notes: Use the Debug Tool command DESCRIBE CUS to give you the correct BLOCK or CU qualification needed.

Use the LIST NAMES command to show all polymorphic functions of a given name. For the example above, LIST NAMES "ICCD0320*" would list all polymorphic functions called ICCD0320.

You do not have to preface variables in the currently executing compile unit. These are already known to Debug Tool; in other words, they are *implicitly* qualified.

In order for attempts at qualifying a variable to work, each block must have a name. Blocks that have not received a name are named by Debug Tool, using the form: %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

```
LOAD_NAME::>CU_NAME:>BLOCK_NAME
```

Each time you update any of the three Debug Tool variables %CU, %PROGRAM, or %BLOCK, all four variables (%CU, %PROGRAM, %LOAD, and %BLOCK) are automatically updated to reflect the new point of view. If you change %LOAD using SET QUALIFY LOAD, only %LOAD is updated to the new point of view. The other three Debug Tool variables remain unchanged. For example, suppose your program is currently suspended at loadx::>cux:>blockx. Also, the load module loadz, containing the compile unit cuz and the block blockz, is known to Debug Tool. The settings currently in effect are:

```
%LOAD = loadx  
%CU = cuz  
%PROGRAM = cuz  
%BLOCK = blockx
```

If you enter any of the following commands:

```
SET QUALIFY BLOCK blockz;
```

```
SET QUALIFY BLOCK cuz:>blockz;
```

```
SET QUALIFY BLOCK loadz::>cuz:>blockz;
```

the following settings are in effect:

```
%LOAD = loadz
```

```
%CU = cuz
```

```
%PROGRAM = cuz
```

```
%BLOCK = blockz
```

If you are debugging a program that has multiple enclaves, SET QUALIFY can be used to identify references and statement numbers in any enclave by resetting the point of view to a new block, compile unit, or load module.

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 Language Environment'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 might be relying in data that was not properly stored or handled.

When debugging with Debug Tool, you have a choice (depending on your host system) of either instructing the debugger to handle program exceptions and conditions, or passing them on to your own exception handler. Programs also have access to Language Environment services to deal with program exceptions and conditions.

Condition handling in Debug Tool

You can use either or both of the two methods during a debugging session to ensure that Debug Tool gains control at the occurrence of HLL conditions.

If you specify TEST(ALL) as a run-time option when you begin your debugging session, Debug Tool gains control at the occurrence of most conditions.

Note: Debug Tool recognizes all Language Environment conditions that are detected by the Language Environment error handling facility.

You can also direct Debug Tool to respond to the occurrence of conditions by using the AT OCCURRENCE command to define breakpoints. These breakpoints halt processing of your program when a condition is raised, after which Debug Tool is given control. It then processes the commands you specified when you defined the breakpoints. For more information on OCCURRENCE breakpoints, see "AT OCCURRENCE" on page 225.

For a description of HLL conditions, see the corresponding language references and *OS/390 Language Environment 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 throws an object.
- A C/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/C++ and PL/I).

When a condition occurs

When an HLL condition occurs and you have defined a breakpoint with associated actions, those actions are first performed. What happens next depends on how the actions end.

- Your program's execution can be terminated with a QUIT command.
- 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, 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/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 is 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. For example, you can stop what appears to be an unending loop, stop the display of voluminous output at your terminal, or stop the execution of the `STEP` command.

An attention interrupt should not be confused with the `ATTENTION` condition. If you set an `AT OCCURRENCE` or `ON ATTENTION`, the commands associated with that break-point are not run at an attention interrupt.

Language Environment run-time options `TRAP` and `INTERRUPT` should both be set to `ON` in order for attention interrupts that are recognized by the host operating system to be also recognized by Language Environment. The test level suboption of the run-time `TEST` option should *not* be set to `NONE`. See *OS/390 Language Environment Programming Guide*.

For CICS Only:

An *attention interrupt* key is not supported in CICS.

For MVS Only:

For C, using an *attention interrupt*, use `SET INTERCEPT ON FILE stdout` to intercept messages to the terminal. This is required because messages do not go to the terminal after an *attention interrupt*.

The correct key might not be marked `ATTN` on every keyboard. Often the following keys are used:

- Under TSO: PA1 key
- Under CMS: PA1 key twice
- Under IMS: PA1 key

When you request an *attention interrupt*, control is given to Debug Tool:

- At the next hook if Debug Tool has previously gained control or if you 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/C++, COBOL, and PL/I

The following Debug Tool built-in functions are for use with C/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/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/C++ or an H constant in COBOL.

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/C++ and PL/I

The following Debug Tool built-in functions are for use only with C/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:

$$1 \leq N \leq \text{ALLOCATION}(X)$$

The oldest instance of X is referenced by %GENERATION(X,1), and the most recent by %GENERATION(X,ALLOCATION(X)).

Chapter 9. Using Debug Tool with C/C++ programs

This chapter provides information on using C/C++ variables and expressions with Debug Tool. It covers the Debug Tool subset of C/C++ commands and reserved words, accessing program variables, declaring temporary variables (also known as session variables), displaying values of C/C++ variables, assigning values to C/C++ variables, and using Debug Tool variables.

It also covers expressions, including discussions of function calls, operators, and C/C++ unique statements; and qualification and multiple load modules.

Debug Tool commands

Debug Tool's command language is a subset of C/C++ commands and has the same syntactical requirements. Debug Tool allows you to work in a language you are familiar with so learning a new set of commands is not necessary.

The interpretive subset of C/C++ commands recognized by Debug Tool is shown in Table 19 on page 347. This subset of commands is valid only when the current programming language is C or C++.

For specific usage notes concerning each command, see the appropriate section of Chapter 13, "Debug Tool commands" on page 205.

In addition to the subset of C/C++ commands that you can use is a list of reserved keywords used and recognized by C/C++ that you cannot abbreviate, use as variable names, or use as any other type of identifier. This list is shown in Table 20 on page 347. These keywords are reserved only when the current programming language is C or C++.

For explanations of command usage and keyword meaning, see *OS/390 C/C++ Language Reference*.

Using C/C++ variables with Debug Tool

Debug Tool can process all program variables that are valid in C or 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 5 or "Compiling a C++ program with the compile-time TEST option" on page 9 for more details.

Note: There are no suboptions for C++. Symbol information is generated by default when the compile-time TEST option is specified.

Displaying values of C/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 274 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] ); G0;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (G0), stops at line 25, and displays the variable names and their values.

If you want to see the result of their addition, enter:

```
AT 25 LIST ( X + row[X] + col[X] ); G0;
```

Debug Tool sets a breakpoint at line 25 (AT), begins execution of the program (G0), stops at line 25, and displays the result of the expression.

Put commas between the variables when listing more than one. If you do not want to display the variable names when issuing the LIST command, enter LIST UNTITLED.

You can also list variables with the printf function call as follows:

```
printf ("X=%d, row=%d, col=%d\n", X, row[X], col[X]);
```

The output from printf, however, does not appear in the Log window and is not recorded in the log file unless you set INTERCEPT ON FILE stdout.

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 12 on page 250.

To declare a floating-point variable called `maximum`, enter the following C declaration:

```
double maximum;
```

You can only declare scalars, arrays of scalars, structures, and unions in Debug Tool (pointers for the above are allowed as well).

If you declare a 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/C++” on page 158. For more on declarations, see “Declarations (C/C++)” on page 247.

Assigning values to C/C++ variables

To assign a value to a C/C++ variable, you use an assignment expression. See “Expression command (C/C++)” on page 264 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.

Note: Only the assignment operators that work for C will work for C++, that is, there is no support for overloaded operators.

The following example demonstrates how to assign the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

Compound assignment operators perform an operation on both operands and give the result of that operation to the left operand. For example, this expression gives the value of `index` plus 2 to the variable `index`:

```
index += 2
```

Debug Tool supports all C operators except the ternary operator, as well as any other full C language assignments and function calls to user or C library functions. For more on function calls, see “Function calls” on page 147.

Using Debug Tool variables in C/C++

Debug Tool variables, as shown in Table 4, 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 4 (Page 1 of 2). C/C++ Attributes for Debug Tool Variables

Debug Tool Variable	C/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.

Table 4 (Page 2 of 2). C/C++ Attributes for Debug Tool Variables

Debug Tool Variable	C/C++ Attributes	Description
%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.
%CAAADDRESS	void *	Contains the address of the CAA control block associated with the suspended program.
%CONDITION	unsigned char[]	Contains the name (or number) of HLL or Language Environment 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 load module 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. %PLANGUAGE returns "C" for both C and C++.
%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/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.
- 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.
- 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 HLL or Language Environment 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 load module and is used when an unqualified reference to a program or variable is made. If the currently qualified load module 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 load module (or to an asterisk in the initial load module). You can change or override the value of `%LOAD` by using the `SET QUALIFY` command.

For modules to be recognized by Debug Tool, they must be 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** Attention function (*not* ATTENTION condition).
- 1** A block has been entered.
- 2** A block is about to be exited.
- 3** Control has reached a user label.
- 4** Control is being transferred as a result of a function reference. The invoked routine's parameters, if any, have been prepared.
- 5** Control is returning from a function reference. Any return code contained in register 15 has not yet been stored.
- 6** Some logic contained by a conditional `do/while`, `for`, or `while` statement is about to be executed. This can be a single or `Null` statement and not a block statement.
- 7** The logic following an `if(...)` is about to be executed.
- 8** The logic following an `else` is about to be executed.
- 9** The logic following a case within an `switch` is about to be executed.
- 10** The logic following a default within a `switch` is about to be executed.
- 13** The logic following the end of a `switch`, `do`, `while`, `if(...)`, or `for` is about to be executed.
- 17** A `goto`, `break`, `continue`, or `return` is about to be executed.

Values in the range 3–17 can only be assigned to `%PATHCODE` if your program was compiled with an option supporting path hooks.

`%PLANGUAGE`

Indicates the programming language currently in use. `%PLANGUAGE` returns `C` for both `C` and `C++`.

`%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:

LINE
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
IMS
TSO
NONE

Subsystems only occur on MVS; if you list this variable while working with VM, Debug Tool displays NONE.

%SYSTEM

Contains the name of the operating system supporting the program. Possible values are:

MVS
VM

You can access certain variables that have no intrinsic meaning in your operating system or language. For example, when debugging in a VM environment, requesting the value of the variable %SUBSYSTEM does not result in an error. However, subsystems occur only on MVS, so %SUBSYSTEM requested during a debugging session under VM always results in NONE.

C/C++ expressions

Debug Tool allows evaluation of expressions in your test program. All expressions available in C/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/C++ language expressions are arranged in the following groups based on the operators they contain and how you use them:

Primary expression
Unary expression
Binary expression
Conditional expression
Assignment expression
Comma expression

lvalue
Constant

An lvalue is an expression representing a data object that can be examined and altered. For a more detailed description of expressions and operators, see the C and C++ Program Guides.

The semantics for C/C++ operators are the same as in a compiled C or C++ program. Operands can be a mixture of constants (integer, floating-point, character, string, and enumeration), C/C++ variables, Debug Tool variables, or session variables declared during a Debug Tool session. Language constants are specified as described in the C and C++ Language Reference publications.

The Debug Tool command `DESCRIBE ATTRIBUTES` can be used to display the resultant type of an expression, without actually evaluating the expression.

The C/C++ language does not specify the order of evaluation for function call arguments. Consequently, it is possible for an expression to have a different execution sequence in compiled code than within Debug Tool. For example, if you enter the following in an interactive session:

```
int x;  
int y;  
  
x = y = 1;  
  
printf ("%d %d %d%" x, y, x=y=0);
```

the results can differ from results produced by the same statements located in a C or C++ program segment. Any expression containing behavior undefined by ANSI standards can produce different results when evaluated by Debug Tool than when evaluated by the compiler.

For more information about expressions and operators, refer to *OS/390 C/C++ 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/C++ language constants in expressions can be used, as in:

```

pointer_to_c = "abcdef" + 0x2;
*pointer_to_long = 3521L = 0x69a1;
float_val = 3e-11 + 6.6E-10;
char_val = '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 C or `TEST` for C++. For details, see “Compiling a C program with the compile-time `TEST` option” on page 5 or “Compiling a C++ program with the compile-time `TEST` option” on page 9.

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)` for C or with `TEST` for C++.

You can turn off this checking by specifying `SET WARNING OFF`.

Calls can be made to any user functions that have linkage supported by the C or C++ compiler. However, for C++ calls made to any user function, the function must be declared as:

```
extern "C"
```

For example, use this declaration if you want to debug an application signal handler. When a condition occurs, control passes to Debug Tool which then passes control to the signal handler.

Debug Tool attempts linkage checking, and does not perform the function call if it determines there is a linkage mismatch. A linkage mismatch occurs when the target program has one linkage but the source program believes it has a different linkage.

It is important to note the following regarding function calls:

- The evaluation order of function arguments can vary between the C/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 *OS/390 C/C++ Language Reference*.

Using Debug Tool functions with C/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+
```

For the proper syntax of the functions described above, see “Debug Tool's built-in functions” on page 135.

The following are examples of command sequences issued to Debug Tool using C/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));
```

- 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/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/C++ expressions following the rules presented in *OS/390 C/C++ Language Reference*. The result of an expression is equal to the result that would have been produced if the same expression had been part of your compiled program.

Implicit string concatenation is supported. For example, "abc" "def" is accepted for "abcdef" and treated identically. Concatenation of wide string literals to string literals is not accepted. For example, L"abc"L"def" is valid and equivalent to L"abcdef", but "abc" L"def" is not valid.

Expressions you use during your session are evaluated with the same sensitivity to enablement as are compiled expressions. Conditions that are enabled are the same ones that exist for program statements.

During a Debug Tool session, if the current setting for WARNING is ON, the occurrence in your C or C++ program of any one of the conditions listed below causes the display of a diagnostic message. The messages themselves are displayed on your terminal, and are explained in Appendix E, "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} to $2^{31}-1$
- A real constant not having an exponent of 3 or fewer digits
- A float constant not larger than 5.39796053469340278908664699142502496E-79 or smaller than 7.2370055773322622139731865630429929E+75
- A hex escape sequence that does not contain at least one hexadecimal digit
- An octal escape sequence with an integer value of 256 or greater
- An unsigned integer constant greater than the maximum value of 4294967295.

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.

For C++, there is no specific support for intercepting IOStreams. IOStreams is implemented using C I/O which implies that:

- If you intercept I/O for a C standard stream, this implicitly intercepts I/O for the corresponding IOStreams standard stream.
- If you intercept I/O for a file, by name, and define an IOStream object associated with the same file, IOStream I/O to that file will be intercepted.

Note: Although you can intercept IOStreams indirectly via C/370 I/O, the behaviors might be different or undefined in C++.

You can use the following names with the SET INTERCEPT command during a 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 like terminal I/O, with the following considerations:

- Intercepted input behaves as though the terminal 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 '\a' character does not cause a beep when running under VM as it does for terminal output.
- The logical record length of an intercepted stream reflects the logical record length of the real file.

- When an unintercepted memory file is opened, the record format is always fixed and the open mode is always binary. These attributes are reflected in the intercepted stream.
- Files opened to the terminal for write are flushed before an input operation occurs from the terminal. This is not supported for intercepted files.

For more information on the behavior of terminal I/O, see OS/390 C/C++ Programming Guide.

Other characteristics of intercepted files are:

- When an `fclose()` occurs or `INTERCEPT` is set `OFF` for a file that was intercepted, the data is flushed to the session log file before the file is closed or the `SET INTERCEPT OFF` command is processed.
- When an `fopen()` occurs for an intercepted file, an open occurs on the real file before the interception takes effect. If the `fopen()` fails, no interception occurs for that file and any assumptions about the real file, such as the ddname allocation and data set defaults, take effect.
- The behavior of the `ASIS` suboption on the `fopen()` statement is not supported for intercepted files.
- When the `clrmemf()` function is invoked and memory files have been intercepted, the buffers are flushed to the session log file before the files are removed.
- If the `fldata()` function is invoked for an intercepted file, the characteristics of the real file are returned.
- If `stderr` is intercepted, the interception overrides the Language Environment message file (the default destination for `stderr`). A subsequent `SET INTERCEPT OFF` command returns `stderr` to its `MSGFILE` destination.
- If a file is opened with a ddname, interception occurs only if the ddname is specified on the `INTERCEPT` command. Intercepting the underlying file name does not cause interception of the stream.
- When running under VM, if a file mode of `"**"` is specified on the `INTERCEPT` command, all files opened with the specified file name and type are intercepted. If a file mode is not specified, `"**"` is assumed.
- User prefix qualifications are included in MVS data set names entered in the `INTERCEPT` command, using the same rules as defined for the `fopen()` function. For more information, see OS/390 C/C++ Programming Guide.
- 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 `INTERCEPT` 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 `INTERCEPT` command, `stderr` is also intercepted. If `stderr` is the target of the `INTERCEPT` command, `stdout` is not intercepted. When `INTERCEPT` is set OFF for `stderr`, the stream is redirected to `stdout` again.

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 `INTERCEPT` 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 `INTERCEPT` command, both `stderr` and `stdout` are intercepted. If you specify `stdout` or `stderr` on the `INTERCEPT` 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.

Note: The use of an object here is not to be confused with a C++ object. Any reference to C++ will be qualified as such.

In ANSI C, the four kinds of scope are:

- Block
- File
- Function
- Function prototype

For C++, in addition to the scopes defined for C, it also has the class scope.

An object has block scope if its declaration is located inside a block. An object with block scope is visible from the point where it is declared to the closing brace (}) that terminates the block.

An object has file scope if its definition appears outside of any block. Such an object is visible from the point where it is declared to the end of the source file. In Debug Tool, if you are qualified to the compilation unit with the file static variables, file static and global variables are always visible.

The only type of object with function scope is a label name.

An object has function prototype scope if its declaration appears within the list of parameters in a function prototype.

A class member has class scope if its declaration is located inside a class.

You cannot reference objects that are visible at function prototype scope, but you can reference ones that are visible at file or block scope if:

- For C variables and functions, the source file was compiled with `TEST(SYM)` and the object was referenced somewhere within the source.
- For C variables declared in a block that is nested in another block, the source file was compiled with `TEST(SYM, BLOCK)`.
- For line numbers, the source file was compiled with `TEST(LINE) GONUMBER`.
- For labels, the source file was compiled with `TEST(SYM, PATH)`. In some cases (for example, when using `GOTO`), labels can be referenced if the source file was compiled with `TEST(SYM, NOPATH)`.

Debug Tool follows the same scoping rules as ANSI, except that it handles objects at file scope differently. An object at file scope can be referenced from within Debug Tool at any point in the source file, not just from the point in the source file where it is declared. Debug Tool 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 load modules. Multiple load modules are managed through the C library functions `dllload()`, `dllfree()`, `fetch()`, and `release()`. For example, let's assume the program shown in Figure 18 on page 155 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 18 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 18. Program Showing Support for Referencing Variables in Multiple Load Modules

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 18 on page 155, 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;
```

Blocks and block identifiers for C++

Block Identifiers tend to be longer for C++ than C because C++ functions can be overloaded. In order to distinguish one function name from the other, each block identifier is like a prototype. For example, a function named `shapes(int,int)` in C would have a block named `shapes`; however, in C++ the block would be called `shapes(int,int)`.

You must always refer to a C++ block identifier in its entirety, even if the function is not overloaded. That is, you cannot refer to `shapes(int,int)` as `shapes` only.

Note: The block name for `main()` is always `main` (without the qualifying parameters after it) even when compiled with C++ because `main()` has extern C linkage.

Since block names can be quite long, it is not unusual to see the name truncated in the `LOCATION` field on the first line of the screen. If you want to find out where you are, enter:

```
QUERY LOCATION
```

and the name will be shown in its entirety (wrapped) in the session log.

Block identifiers are restricted to a length of 255 characters. Any name longer than 255 characters is truncated.

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` opens a list of open files and conditions being monitored by the run-time environment. For example, if you enter `DESCRIBE ENVIRONMENT` while debugging a C or C++ program, you might get the following output:

```
Currently open files
  stdout
  sysprint
The following conditions are enabled:
  SIGFPE
  SIGILL
  SIGSEGV
  SIGTERM
  SIGINT
  SIGABRT
  SIGUSR1
  SIGUSR2
  SIGABND
```

Using qualification for C/C++

Qualification is a method of:

- Specifying an object through the use of qualifiers
- Changing the point of view.

Qualification is often necessary due to name conflicts, or when a program consists of multiple load modules, compile units, and/or functions.

When program execution is suspended and Debug Tool receives control, the default, or *implicit* qualification is the active block at the point of program suspension. All objects visible to the C or C++ program in this block are also visible to Debug Tool. Such objects can be specified in commands without the use of qualifiers. All others must be specified using *explicit qualification*.

Qualifiers depend, of course, upon the naming convention of the system where you are working. For instance, Figure 19 on page 159, shows a block of code from a C program under VM, and Figure 20 on page 160 shows a block of code from a C program under MVS.

In both examples, when Debug Tool receives control, variables `i`, `j`, `temp`, `table`, and `length` can be specified without qualifiers in a command. If variable `sn` is referenced, Debug Tool uses the variable that is a `float`. However, the names of the blocks and compile units differ, maintaining compatibility with the operating system.

```

LOAD MODULE 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);
    ...
}

LOAD MODULE 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 19. Qualification in VM

```
LOAD MODULE NAME:  MAINMOD
SOURCE FILE  NAME:  MVSID.SORTMAIN.C

short length = 40;
main ()
{
    long *table;
    void (*pf)();

    table = malloc(sizeof(long)*length);
    ...
    pf = fetch("SORTMOD");
    (*pf)(table);
    ...
    release(pf);
    ...
}

LOAD MODULE NAME:  SORTMOD
SOURCE FILE  NAME:  MVSID.SORTSUB.C

short length = 40;
short sn = 3;
void (long table[])
{
    short i;
    for (i = 0; i < length-1; i++) {
        short j;
        for (j = i+1; j < length; j++) {
            float sn = 3.0;
            short temp;
            temp = table[i];
            ...
            >>> Debug Tool is given <<<
            >>> control here.    <<<
            ...
            table[i] = table[j];
            table[j] = temp;
        }
    }
}
```

Figure 20. Qualification in MVS

Using qualifiers

You can precisely specify an object, provided you know the following:

- Load module or DLL name
- Source file (compilation unit) name
- Block name (must include function prototype for C++ block qualification).

These are known as qualifiers and some, or all, might be required when referencing an object in a command. Qualifiers are separated by a combination of greater than signs (>) and colons and precede the object they qualify. For example, the following is a fully qualified object:

```
LOAD_NAME::>CU_NAME:>BLOCK_NAME:>object
```

If required, LOAD_NAME is the name of the load module. It is required only when the program consists of multiple load modules and when you want to change the quali-

fication to other than the current load module. `LOAD_NAME` is enclosed in double quotation marks. If it is not, it must be a valid identifier in the C or C++ programming language. `LOAD_NAME` can also be the Debug Tool variable `%LOAD`.

If required, `CU_NAME` is the name of the compilation unit or source file. The `CU_NAME` must be the fully qualified source file name or an absolute pathname. It is required only when you want to change the qualification to other than the currently qualified compilation unit. It can be the Debug Tool variable `%CU`. If there appears to be an ambiguity between the compilation unit name, and (for example), a block name, you must enclose the compilation unit name in double quotation marks (").

If required, `BLOCK_NAME` is the name of the block. This has the same syntax as described in the section on "Blocks and block identifiers for C" on page 156. `BLOCK_NAME` can be the Debug Tool variable `%BLOCK`.

For VM Only: The following examples are based on Figure 19 on page 159.

- 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 load module and compilation unit, it can also be changed by:

```
length = 20;
```

- Assume Debug Tool gained control as shown in Figure 19 on page 159. You can break whenever the variable `temp` in load module `SORTMOD` changes in any of the following ways:

```
AT CHANGE temp;
AT CHANGE %BLOCK3:>temp;
AT CHANGE sort:>%BLOCK3:>temp;
AT CHANGE %BLOCK:>temp;
AT CHANGE %CU:>sort:>%BLOCK3:>temp;
AT CHANGE "SORTSUB":>sort:>%BLOCK3:>temp;
AT CHANGE "SORTMOD":>"SORTSUB":>sort:>%BLOCK3:>temp;
```

For MVS Only: The following examples are based on Figure 20 on page 160.

- Change the file scope variable `length` defined in the compilation unit `MVSID.SORTSUB.C` in the load module `SORTMOD`:

```
"SORTMOD":>"MVSID.SORTSUB.C":>length = 20;
```

- Assume Debug Tool gained control from `main()`. The following changes the variable `length`:

```
%LOAD:>"MVSID.SORTMAIN.C":>length = 20;
```

Because `length` is in the current load module and compilation unit, it can also be changed by:

```
length = 20;
```

- Assume Debug Tool gained control as shown in Figure 20 on page 160. You can break whenever the variable `temp` in load module `SORTMOD` changes in any of the following ways:

```
AT CHANGE temp;
AT CHANGE %BLOCK3:>temp;
AT CHANGE sort:%BLOCK3:>temp;
AT CHANGE %BLOCK:>temp;
AT CHANGE %CU:>sort:>%BLOCK3:>temp;
AT CHANGE "MVSID.SORTSUB.C":>sort:>%BLOCK3:>temp;
AT CHANGE "SORTMOD":>"MVSID.SORTSUB.C":>sort:>%BLOCK3:>temp;
```

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 load module or DLL, to another compilation unit, to a nested block, or to a block that is not nested. The SET keyword is optional.

The following examples of changing the point of view are based on Figure 19 on page 159:

- 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 load module MAINMOD in the compilation unit SORTMAIN and list the entries of table.

```
QUALIFY BLOCK "MAINMOD":>"SORTMAIN":>main();
LIST table[i];
```

The following examples of changing the point of view are based on Figure 20 on page 160:

- Qualify to the second nested block in the function sort() in sort.

```
SET QUALIFY BLOCK %BLOCK2;
```

You can do this in a number of other ways, including:

```
QUALIFY BLOCK sort:>%BLOCK2;
```

Once the point of view changes, Debug Tool has access to objects accessible from this point of view. You can specify these objects in commands without qualifiers, as in:

```
j = 3;
temp = 4;
```

- Qualify to the function main in the load module MAINMOD in the compilation unit MVSID.SORTMAIN.C and list the entries of table.

```
QUALIFY BLOCK "MAINMOD":>"MVSID.SORTMAIN.C":>main;
LIST table[i];
```

Stepping through C++ programs

You can step through methods as objects are constructed and destructed. In addition, you can step through static constructors and destructors. These are methods of objects that are executed before and after `main()` respectively.

If you are debugging a program that calls a function that resides in a header file, the cursor moves to the applicable header file. You can then view the function source as you step through it. Once the function returns, debugging continues at the line following the original function call.

You can step around a header file function by issuing the `STEP OVER` command. This is useful in stepping over Library functions (e.g., string functions defined in `string.h`) that you cannot debug anyway.

Setting breakpoints in C++

Setting `AT ENTRY/EXIT` and `AT CALL` breakpoints in C++ differs from C. The following sections describe the differences and provides you with examples of these differences:

AT ENTRY/EXIT

`AT ENTRY/EXIT` sets a breakpoint in the specified block. You can set a breakpoint on methods, methods within nested classes, templates, and overloaded operators. An example is given for each below.

A block identifier can be quite long, especially with templates, nested classes, or class with many levels of inheritance. In fact, it might not even be obvious at first as to the block name for a particular function. To set a breakpoint for these non-trivial blocks can be quite cumbersome. Therefore, it is recommended that you make use of `DESCRIBE CU` and retrieve the block identifier from the session log as described in “Retrieving commands from the log and source windows” on page 198.

When you do a `DESCRIBE CU`, the methods are always shown qualified by their class. If a method is unique, you can set a breakpoint by using just the method name. Otherwise, you must qualify the method with its class name. The following two examples are equivalent:

```
AT ENTRY method()
```

```
AT ENTRY classname::method()
```

The following examples are valid:

Using Debug Tool with C/C++ programs

AT ENTRY square(int,int)	'simple' method square
AT ENTRY shapes::square(int)	Method square qualified by its class shapes.
AT EXIT outer::inner::func()	Nested classes. Outer and inner are classes. func() is within class inner.
AT EXIT Stack<int,5>::Stack()	Templates.
AT ENTRY Plus::operator++(int)	Overloaded operator.
AT ENTRY ::fail()	Functions defined at file scope must be referenced by the global scope operator ::

The following examples are invalid:

AT ENTRY shapes	Where shapes is a class. Cannot set breakpoint on a class. (There is no block identifier for a class.)
AT ENTRY shapes::square	Invalid since method square must be followed by its parameter list.
AT ENTRY shapes:>square(int)	Invalid since shapes is a class name, not a block name.

AT CALL

AT CALL gives Debug Tool control when the application code attempts to call the specified entry point. The entry name must be a fully qualified name. That is, the name shown in DESCRIBE CU must be used. Using the example

```
AT ENTRY shapes::square(int)
```

to set a breakpoint on the method square, you must enter:

```
AT CALL shapes::square(int)
```

even if square is uniquely identified.

Examining C++ objects

When displaying an object, only the local member variables are shown. Access types (public, private, protected) are not distinguished among the variables. The member functions are not displayed. If you want to see their attributes, you can display them individually, but not in the context of a class. When displaying a derived class, the base class within it is shown as type class and will not be expanded as the Objects example shows. Here are examples of displaying C++ objects:

```
<< Sample program source >>

class shape {
    . . .
};

class line : public shape {
    member variables of class line...
}

line edge;

<< End >>
```

Objects

To describe attributes for the object EDGE, enter DESCRIBE ATTRIBUTES edge and you get:

```
DESCRIBE ATTRIBUTES edge;
ATTRIBUTES for edge
  Its address is yyyyyyyy and its length is xx
  class line
    class shape
      member variables of class shape....
```

Note that the base class is shown as class shape _shape.

Classes

To display the attributes of class shape, enter DESCRIBE ATTRIBUTES class shape and you get:

```
DESCRIBE ATTRIBUTES class shape ;
ATTRIBUTES for class shape
  const class shape
    . . .
    . . .
```

Static data

If a class contains static data, the static data will be shown as part of the class when displayed. For example:

```
class A {
    int x;
    static int y;
}
```

```
A obj;
```

You can also display the static member by referencing it as A::y since each object of class A has the same value.

Global data

To avoid ambiguity, variables declared at file scope can be referenced using the global scope operator `::`. For example:

```
int x;
class A {
    int x;
    . . .
}
```

If you are within a member function of `A` and want to display the value of `x` at file scope, enter `LIST ::x`. If you do not use `::`, entering `LIST x` will display the value of `x` for the current object (i.e., `this->x`).

Low-level debugging

Debug Tool is not an assembly-level debugger, 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 displays the pseudo assembly code, including Debug Tool hooks. You can watch the hooks that you stop on and watch expected changes in register values step by step in accordance with the pseudo assembly instructions between the hooks. You can also modify the value of machine registers while stepping through your code.

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)
      ST    r1,152(,r13)
* {
      EX    r0,HOOK..PGM-ENTRY
*   return 2*j;
      EX    r0,HOOK..STMT
      L     r15,152(,r13)
      L     r15,0(,r15)
      SLL   r15,1
      B     @5L2
      DC    A@5L2-ep)
      NOPR
@5L1   DS    0D
*   }
@5L2   DS    0D
      EX    r0,HOOK..PGM-EXIT

```

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 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 load modules.

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.

Debugging environment provided for COBOL programs

While Debug Tool allows you to use many commands that are either very similar or equivalent to COBOL commands, Debug Tool does not necessarily interpret these commands as required by the compiler options you chose when compiling your program. This is due to the fact that, in the Debug Tool environment, the following settings are in effect:

```
DYNAM
NOCMPR2
NODBCS
NOWORD
NUMPROC(NOPFD)
QUOTE
TRUNC(BIN)
ZWB
```

For more information on these compile-time options, see the COBOL Language Reference publications.

Debug Tool Subset of COBOL commands

To make testing COBOL programs easier, Debug Tool allows you to write debugging commands that resemble COBOL commands. It does this by providing an *interpretive subset* of COBOL language commands that is recognized by Debug Tool and either closely resembles or duplicates the syntax and action of the appropriate COBOL commands. 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 commands recognized by Debug Tool is shown in Table 23 on page 350. This subset of commands is valid only when the current programming language is COBOL.

For explanations of command usage and keyword meaning, see the *COBOL for MVS & VM 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 COBOL Language Reference publications.

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).
- `COMPUTE` cannot be used to perform a computation with a windowed date field if the arithmetic expression consists of more than one operand.
- Any expanded date field specified as an operand in the arithmetic expression is treated as a nondate field.
- The result of the evaluation of the arithmetic expression is always considered to be a nondate field.
- 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 COBOL Language Reference publications.

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.
- `MOVE` does not support date windowing. Therefore, you cannot use the `MOVE` command to assign the value of a windowed date field to an expanded date field or to a nondate field.
- You cannot use the `MOVE` command to assign the value of one expanded date field to another expanded date field with a different `DATE FORMAT` clause, or

to assign the value of one windowed date field to another windowed date field with a different DATE FORMAT clause.

Table 25 on page 352 shows the permissible moves for the MOVE command.

For more information, see “MOVE command (COBOL)” on page 288 and COBOL Language Reference publications.

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 26 on page 354 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 182.

For more information, see “SET command (COBOL)” on page 328 and COBOL Language Reference publications.

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.
- Comparison combinations with windowed date fields are not supported.
- Comparisons between expanded date fields with different DATE FORMAT clauses are not supported.
- Only the comparisons shown in Table 24 on page 351 are supported.

For more information, see “IF command (COBOL)” on page 271 and COBOL Language Reference publications.

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 24 on page 351 are supported.

For more information, see “EVALUATE command (COBOL)” on page 262 and COBOL Language Reference publications.

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 phrase.
- Windowed date fields are not supported in the VARYING, FROM, or BY phrases.

For more information, see “PERFORM command (COBOL)” on page 293 and COBOL Language Reference publications.

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.
- A windowed date field cannot be specified as either the identifier containing the entry name, or an identifier in the USING phrase.

For more information, see “CALL command” on page 234 and COBOL Language Reference publications.

COBOL command format

When you are entering commands directly at your terminal or workstation, the format is free-form, because you can begin your commands in column 1 and continue long commands using the appropriate method. You can continue on the next line during your Debug Tool session by using an SBCS hyphen (-) as a continuation character.

However, when you use a file as the source of command input, the format for your commands is similar to the source format for the COBOL compiler. The first six positions are ignored, and an SBCS hyphen in column 7 indicates continuation from the previous line. You must start the command text in column 8 or later, and end it in column 72.

The continuation line (with a hyphen in column 7) optionally has one or more blanks following the hyphen, followed by the continuing characters. In the case of the continuation of a literal string, an additional quote is required. When the token being continued is not a literal string, blanks following the last nonblank character on the previous line are ignored, as are blanks following the hyphen.

Using Debug Tool with COBOL programs

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 10 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 21. The examples concerning LIST, found in “Displaying values of COBOL variables” on page 174, 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 21. 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 21 on page 172 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 '20000' 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.

To assign a value to a temporary variable named CMS, TSO, or SYSTEM, about the "=" to the reference as shown in the following example:

```
COMPUTE cms= 5;
```

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 COBOL Language Reference publications. Only elementary variables with the attributes shown in Table 13 on page 252 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 Data pop-up window if you want to update the value.

See COBOL Language Reference publications for discussions of DBCS usage with COBOL.

Using Debug Tool variables in COBOL

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, issue the LIST command.

Table 5 (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 (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.

Table 5 (Page 2 of 2). Descriptions of Debug Tool Variables

Debug Tool Variable	COBOL Attributes	Description
%CONDITION	PICTURE X(j)	Contains the name (or number) of the condition identification when Debug Tool is entered because of an HLL or Language Environment 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 load module of the current program.
%NLANGUAGE	PICTURE X(j)	Contains the national language currently being used.
%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 commands:

```
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. The only possible value is 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 Language Environment 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 load module and is used when an unqualified reference to a program or variable is made. If the currently qualified load module 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 load module (or to an asterisk in the initial load module). You can change or override the value of %LOAD by using the QUALIFY command.

Note: For modules to be recognized by Debug Tool, they must be loaded using Language Environment 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 Attention function (*not* ATTENTION condition).
- 1 A block has been entered.
- 2 A block is about to be exited.

- 3 Control has reached a label coded in the program (a paragraph name or section name).
- 4 Control is being transferred as a result of a CALL or INVOKE. The invoked routine's parameters, if any, have been prepared.
- 5 Control is returning from a CALL or INVOKE. If GPR 15 contains a return code, it has already been stored.
- 6 Some logic contained by an inline PERFORM is about to be executed. (Out-of-line PERFORM ranges must start with a paragraph or section name, and are identified by %PATHCODE = 3.)
- 7 The logic following an IF...THEN is about to be executed.
- 8 The logic following an ELSE is about to be executed.
- 9 The logic following a WHEN within an EVALUATE is about to be executed.
- 10 The logic following a WHEN OTHER within an EVALUATE is about to be executed.
- 11 The logic following a WHEN within a SEARCH is about to be executed.
- 12 The logic following an AT END within a SEARCH is about to be executed.
- 13 The logic following the end of one of the following structures is about to be executed:
 - An IF statement (with or without an ELSE clause)
 - An EVALUATE or SEARCH
 - A PERFORM.
- 14 Control is about to return from a declarative procedure such as USE AFTER ERROR. (Declarative procedures must start with section names, and are identified by %PATHCODE = 3.)
- 15 The logic associated with one of the following phrases is about to be run:
 - [NOT] ON SIZE ERROR
 - [NOT] ON EXCEPTION
 - [NOT] ON OVERFLOW
 - [NOT] AT END (other than SEARCH AT END)
 - [NOT] AT END-OF-PAGE
 - [NOT] INVALID KEY.
- 16 The logic following the end of a statement containing one of the following phrases is about to be run:
 - [NOT] ON SIZE ERROR
 - [NOT] ON EXCEPTION
 - [NOT] ON OVERFLOW
 - [NOT] AT END (other than SEARCH AT END)
 - [NOT] AT END-OF-PAGE
 - [NOT] INVALID KEY.

Note: Values in the range 3–16 can be assigned to %PATHCODE only if your program was compiled with an option supporting path hooks.

%LANGUAGE

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:

LINE
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
IMS
TSO
NONE

Subsystems occur only on MVS; a request for %SUBSYSTEM from a VM host returns NONE.

%SYSTEM

Contains the name of the operating system supporting the program. Possible values are:

MVS
VM

Debug Tool evaluation of COBOL expressions

Debug Tool interprets COBOL expressions according to COBOL rules. Some restrictions do apply. For example, the following restrictions apply when arithmetic expressions are specified:

- Floating-point operands are not supported (COMP-1, COMP-2, external floating point, floating-point literals).
- Only integer exponents are supported.
- Intrinsic functions are not supported.
- Windowed date-field operands are not supported in arithmetic expressions in combination with any other operands.

When arithmetic expressions are used in relation conditions, both comparand attributes are considered. Relation conditions follow the IF rules rather than the EVALUATE rules.

Only simple relation conditions are supported. Sign conditions, class conditions, condition-name conditions, switch-status conditions, complex conditions, and abbreviated conditions are not supported. When either of the comparands in a relation condition is stated in the form of an arithmetic expression (using operators such as plus and minus), the restriction concerning floating-point operands applies to both comparands.

When both comparands are stated as simple references, all combinations listed in Table 24 on page 351 are supported.

Windowed date fields are not supported in relation conditions.

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 COBOL Language Reference publications for discussions of COBOL expression evaluation.

Expressions are evaluated according to COBOL rules applying to the options specified in “Debugging environment provided for COBOL programs” on page 168. 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 COBOL Language Reference publications are valid in Debug Tool, with the following restrictions:

- 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 2222 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 135.

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 load modules, compile units, blocks, section names, or paragraph names punctuated by a combination of greater-than signs (`>`), colons, and the COBOL data qualification notation, `OF` or `IN`, that precede referenced statement numbers or variable names.

When qualifying objects on a block level, use only the COBOL form of data qualification. If data names are unique, or defined as `GLOBAL`, they do not need to be qualified to the block level.

The following is a fully qualified object:

```
LOAD_NAME::>CU_NAME:>BLOCK_NAME:>object;
```

If required, `LOAD_NAME` is the name of the load module. It is required only when the program consists of multiple load modules and you want to change the qualification to other than the current load module. `LOAD_NAME` can also be the Debug Tool variable `%LOAD`.

If required, `CU_NAME` is the name of the 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`.

If required, `BLOCK_NAME` is the name of the block. The `BLOCK_NAME` is required only when you want to change the qualification to other than the currently qualified block. It can be the Debug Tool variable `%BLOCK`. Remember to enclose the block name in double (`"`) or single (`'`) quotes if case sensitive. If the name is not inside quotes, Debug Tool converts the name to upper case.

The following two screens are samples of two similar COBOL programs (blocks):

Using Debug Tool with COBOL programs

```
MAIN
:
  01 VAR1.
    02 VAR2.
      03 VAR3      PIC XX.
  01 VAR4      PIC 99..

*****MOVE commands entered here*****
```

```
SUBPROG
:
  01 VAR1.
    02 VAR2.
      03 VAR3      PIC XX.
  01 VAR4      PIC 99.
  01 VAR5      PIC 99.

*****LIST commands entered here*****
```

You can distinguish between the main and subprog blocks using qualification. If you enter the following MOVE commands when main is the currently executing block:

```
MOVE 8 TO var4;
MOVE 9 TO subprog:>var4;
MOVE 'A' TO var3 OF var2 OF var1;
MOVE 'B' TO subprog:>var3 OF var2 OF var1;
```

and the following LIST commands when subprog is the currently executing block:

```
LIST TITLED var4;
LIST TITLED main:>var4;
LIST TITLED var3 OF var2 OF var1;
LIST TITLED main:>var3 OF var2 OF var1;
```

each LIST command results in the following output (without the commentary) in your Log window:

```
VAR4 = 9;      /* var4 with no qualification refers to a variable */
               /* in the currently executing block (subprog). */
               /* Therefore, the LIST command displays the value of 9.*/

MAIN:>VAR4 = 8  /* var4 is qualified to main. */
               /* Therefore, the LIST command displays 8, */
               /* the value of the variable declared in main. */

VAR3 OF VAR2 OF VAR1 = 'B';
               /* In this example, although the data qualification */
               /* of var3 is OF var2 OF var1, the */
               /* program qualification defaults to the currently */
               /* executing block and the LIST command displays */
               /* 'B', the value declared in subprog. */

VAR3 OF VAR2 OF VAR1 = 'A'
               /* var3 is again qualified to var2 OF var1 */
               /* but further qualified to main. */
               /* Therefore, the LIST command displays */
               /* 'A', the value declared in main. */
```

The above method of qualifying variables is necessary for 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 about the Debug Tool subset of commands for PL/I. It also covers PL/I language statements, conditions, expressions, and functions. Debug Tool also supports PL/I freeform DBCS input, which is covered in this chapter as well.

Debug Tool Subset of PL/I commands

Table 6 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 6. 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

PL/I language statements

PL/I statements are entered as Debug Tool *commands*. Debug Tool makes it possible to issue commands in a manner similar to each language.

The following types of Debug Tool commands will support the syntax of the PL/I statements:

Expression

This command evaluates an expression.

Block

BEGIN/END, DO/END, PROCEDURE/END

These commands provide a means of grouping any number of Debug Tool commands into "one" command.

Conditional

IF/THEN, SELECT/WHEN/END

These commands evaluate an expression and control the flow of execution of Debug Tool commands according to the resulting value.

Declaration

DECLARE or DCL

These commands provide a means for declaring session variables.

Looping

DO/WHILE/UNTIL/END

These commands provide a means to program an iterative or conditional loop as a Debug Tool command.

Transfer of Control

GOTO, ON

These commands provide a means to unconditionally alter the flow of execution of a group of commands.

Table 7 shows the commands that are new or changed for this release of Debug Tool when the current programming language is PL/I.

Table 7. PL/I Language-Oriented Commands

Command	Description or Changes
ANALYZE	Displays the PL/I style of evaluating an expression, and the precision and scale of the final and intermediate results.
ON	Performs as the AT OCCURRENCE command except it takes PL/I conditions as operands.
BEGIN	BEGIN/END blocks of logic.
DECLARE	Session variables can now include COMPLEX (CPLX), POINTER, BIT, BASED, ALIGNED, UNALIGNED, etc. Arrays can be declared to have upper and lower bounds. Variables can have precisions and scales.
DO	The three forms of DO are added; one is an extension of C's do. 1. DO; command(s); END; 2. DO WHILE UNTIL expression; command(s); END; 3. DO reference=specifications; command(s); END;
IF	The IF / ELSE does not require the ENDIF.
SELECT	The SELECT / WHEN / OTHERWISE / END programming structure is added.

Using Debug Tool variables in PL/I

%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:

- | | |
|----------|--|
| 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.

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 289.

When an OCCURRENCE breakpoint is triggered, the Debug Tool %CONDITION variable holds the following values:

Table 8. PL/I Conditions and %CONDITION Values(1)

Triggered Condition	%CONDITION Value
AREA	AREA
ATTENTION	CEE35J
COND (CC#1)	CONDITION
CONVERSION	CONVERSION
ENDFILE (MF)	ENDFILE
ENDPAGE (MF)	ENDPAGE
ERROR	ERROR
FINISH	CEE066
FOFL	CEE348
KEY (MF)	KEY
NAME (MF)	NAME
OVERFLOW	CEE34C
PENDING (MF)	PENDING
RECORD (MF)	RECORD
SIZE	SIZE
STRG	STRINGRANGE
STRINGSIZE	STRINGSIZE
SUBRG	SUBSCRIPTRANGE
TRANSMIT (MF)	TRANSMIT
UNDEFINEDFILE (MF)	UNDEFINEDFILE
UNDERFLOW	CEE34D
ZERODIVIDE	CEE349

(1): The Debug Tool condition ALLOCATE raises the ON ALLOCATE condition when a PL/I program encounters an ALLOCATE statement for a controlled variable.

These PL/I language-oriented commands are only a subset of all the commands that are supported by Debug Tool.

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.

TEST(ERROR, ...)

With the run-time option, TEST(ERROR, ...) only the following can initialize Debug Tool:

- The ERROR condition
- Attention recognition
- CALL PLITEST
- CALL CEETEST

LIST STORAGE

LIST STORAGE address has been enhanced so that the address can be a POINTER, a Px constant, or the ADDR built-in function.

Session variables

PL/I will support all Debug Tool scalar session variables. In addition, arrays and structures can be declared.

Refer to Table 14 on page 255 for variables whose attributes will let them be properly used by other programming languages.

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 13 for more information).

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.

Variable that are based on:

- An OFFSET variable,
- An expression, or
- A pointer that either is based or defined, a parameter, or member of either an array or a structure

must be explicitly qualified when used in expressions. For example, assume you made the following declaration:

```
DECLARE P1 POINTER;  
DECLARE P2 POINTER;  
DECLARE DX FIXED BIN(31) BASED(P2);
```

You would not be able to reference the variable directly by name. You can only reference it by specifying either:

```
P2->DX  
or  
P1->P2->DX
```

The following types of program variables cannot be used with Debug Tool:

- iSUB defined variables
- Variables defined:
 - On a controlled variable
 - On an array with one or more adjustable bounds
 - With a POSITION attributed that specifies something other than a constant
- Variables that are members of a based structure declared with the REFER options.

Structures

You cannot reference elements of arrays of structures. For example, suppose a structure called PAYROLL is declared as follows:

```
Declare 1 Payroll(100),  
       2 Name,  
         4 Last      char(20),  
         4 First     char(15),  
       2 Hours,  
         4 Regular   Fixed Decimal(5,2),  
         4 Overtime  Fixed Decimal(5,2);
```

Given the way PAYROLL is declared, the following examples of commands are **valid** in Debug Tool:

```
LIST ( PAYROLL(1).NAME.LAST, PAYROLL(1).HOURS.REGULAR );
```

```
LIST ( ADDR ( PAYROLL ) ) ;
```

```
LIST STORAGE ( PAYROLL.HOURS, 128 );
```

Given the way PAYROLL is declared, the following examples of commands are **invalid** in Debug Tool:


```
LIST ( PAYROLL(1) );

LIST (ADDR ( PAYROLL(5) ) );

LIST STORAGE ( PAYROLL(15).HOURS, 128 ) );
```

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 192.

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.

PL/I built-in functions

The Debug Tool supports the following PL/I built-in functions:

Table 9. 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:

¹Abbreviation for BINARYVALUE.

²Abbreviation for CURRENTSTORAGE.

³Abbreviation for POINTERADD.

⁴Abbreviation for 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.

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)

Positive identification of a compile unit (CU)

Debug Tool determines if a CU is Language Environment-enabled. If it is, the programming language can be determined. If it is not, the Language Environment utility exit is used to identify the programming language. If the programming language of a CU cannot be identified, Debug Tool ignores the CU.

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 three modes: full-screen, line, 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. The following topics are discussed:

Command format Character set and case Abbreviating or truncating keywords
Continuing multiline commands Significance of blanks Using comments or constants
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.

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).

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 308.

Using C/C++

For both C and C++, Debug Tool set the programming language to C. When the current programming language setting is C:

- All keywords and identifiers must be the correct case. Debug Tool does not do conversion to uppercase.
- DBCS characters are allowed only within comments and literals.
- Either trigraphs or the equivalent special characters can be used. Trigraphs are treated as their equivalents at all times. For example, FIND "??<" would find not only "??<" but also "{".
- The vertical bar (|) can be entered for the following C/C++ operations: bitwise or (|), logical or (||), and bitwise assignment or (|=).
- There are alternate code points for the following C/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 *OS/390 C/C++ User's Guide* for an explanation of the alternate and primary code points in C/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, system keywords (that is, CMS, SYS, SYSTEM, or TSO) or special case keywords such as BEGIN, CALL, COMMENT, COMPUTE, END, FILE (in the SET INTERCEPT and SET LOG commands), GOTO, INPUT, LISTINGS (in the SET DEFAULT LISTINGS command), or USE. In addition, PROCEDURE can only be abbreviated as PROC.

The system keywords, and COMMENT, INPUT, and USE keywords, take precedence over other keywords and identifiers. If one of these keywords is followed by a blank, it is always parsed as the corresponding command. Hence, if you want to assign the value 2 to a variable named *CMS* and the current programming language setting is C, the "=" must be abutted to the reference, as in "CMS<no space>= 2;" not "CMS<space>= 2;". If you want to define a procedure named USE, you must enter "USE<no space>: procedure;" not "USE<space>:: procedure;".

When you truncate, you need only enter enough characters of the command to distinguish the command from all other valid Debug Tool commands. You should *not* use truncations in a commands file or compile them into programs because they might become ambiguous in a subsequent release. The following shows examples of Debug Tool command truncations:

If you enter the following command...	It will be interpreted as...
A 3	AT 3
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 and line mode)

If you need to use more than one line when entering a command, you must use a continuation character.

When you are entering a command in interactive mode, the continuation character must be the last nonblank character in each line that is to be continued. In the following example:

```
LIST (" this is a very very very vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -
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/C++, if you want to enter `"i—"`, you could enter `"(i—)"` or `"i—;"`. When the current programming language setting is C/C++, the back slash character (`\`) can also be used.

When Debug Tool is awaiting the continuation of a command in full-screen mode, you receive a continuation prompt of `"MORE..."` until the command is completely entered and processed. When continuation is indicated in line mode, you receive a continuation prompt of `"PENDING..."` until the command is completely entered and processed.

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/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 vvvvvvvvvvvvvvvvvvvvvvvvvvvvv"
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 C++ only: Comments in the form `"/"` are not processed by Debug Tool in C++.

- For all supported programming languages, comments can be entered by:
 - Enclosing the text in comment brackets `"/"` and `"/"`. Comments can occur anywhere a blank can occur between keywords, identifiers, and numeric constants. Comments entered in this manner do not appear in the session log.
 - Using the `COMMENT` command to insert commentary text in the session log. Comments entered in this manner cannot contain embedded semicolons.
- When the current programming language setting is COBOL, comments can also be entered by using an asterisk (`*`) in column 7. This is valid for file input only.

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/C++ expressions” on page 145 or “Using constants in expressions” on page 182 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 nonnumeric literals, such as `MOVE X'C1C2C3C4' TO NON-PTR-VAR`, should be used for all other situations where a hexadecimal value is needed.

The PL/I `PX` constant is a hexadecimal value, delimited by single quotes (`'`) and followed by `PX`. The value is right-justified and can be used in any context in which a pointer value is allowed. For example, to display the contents at a given address in hexadecimal format, specify:

```
LIST STORAGE (H'20CD0');
```

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 300.

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 can be one of:
;          DECLARE      IF          SCROLL
reference  DESCRIBE     INPUT       SELECT
register   DISABLE      LIST        SET
ANALYZE    DO           MONITOR     STEP
AT         ENABLE       PANEL       SYSTEM
BEGIN      END          PROCEDURE   name TRIGGER
CALL       END          QUALIFY     TSO
CLEAR      FIND         QUERY       USE
COMMENT    GO           QUIT        WINDOW
CURSOR     GO TO       RETRIEVE
DCL        GOTO        RUN
```

The above output sample is meant to illustrate a point and might not appear exactly as shown.

Note: DECLARE is not a command but a method of making an interactive variable or tag declaration.

Also, 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 can be one of:
BOTTOM      RIGHT
DOWN        TO
LEFT        TOP
NEXT        UP

```

The Debug Tool CMS, SYSTEM, and TS0 commands followed by ? do not invoke the syntax help; instead the ? is sent to the host as part of the system command. The COMMENT command followed by ? also does not invoke the syntax help.

Common syntax elements

Several syntax elements are used in many Debug Tool commands. To reduce the size of this document, 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/C++ function or a block statement
- A COBOL nested program or method contained within a complete COBOL program
- A PL/I block

The current block qualification can be changed using the SET QUALIFY BLOCK command.

For C++ Only:

Include full declaration in block qualification.

For COBOL Only:

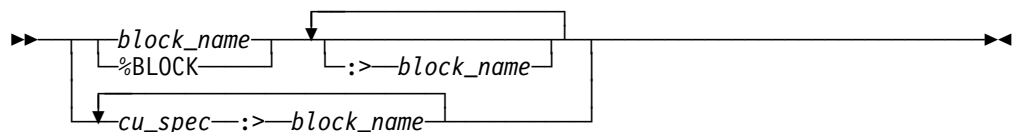
Enclose the block name in double (") or single (') quotes if it is case sensitive. If the name is not inside quotes, Debug Tool will convert the name to upper case.

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.

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 3 on page 127.

cu_spec

A valid compile unit specification; see “CU_Spec” on page 201.

You can only use *block_name* for blocks known in the current enclave.

For C++ Only:

Block_spec must include the formal parameters for the function. The correct block qualification is:

```
int function(int, int) is function(int, int)
```

Use Describe CUS to determine correct *block_spec* for blocks known in the current enclave.

Compile_Unit_Name

A *compile_unit_name* identifies:

- A C/C++ source file
- A COBOL program or class
- The external procedure name of a PL/I program.

For C/C++ Only:

The compile unit name must be enclosed in double quotes (") when there is any chance of ambiguity between a block name and a compile unit name. For example:

```
LIST CU2:>CU2:>var1
```

is ambiguous because the compile unit and a function in that compile unit has same name.

To avoid the ambiguity, use:

```
LIST "CU2":>CU2:>var1
```

to correctly list the value of the variable *var1* scoped to the function CU2.

Escape sequences in compile unit names that are specified as strings are not processed if the string is part of a qualification statement.

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.

For PL/I only:

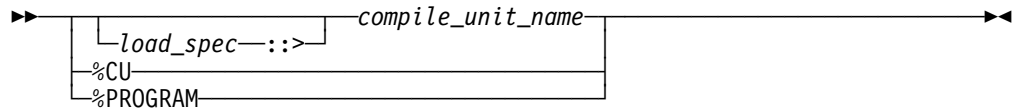
For consistency, the compile unit name can optionally be enclosed in single quotes (').

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.



load_spec

A valid load module specification; see “Load_Spec” on page 202. If omitted, the current load module qualification is used.

compile_unit_name

A valid compile unit name; see “Compile_Unit_Name” on page 200.

%CU

Represents the currently qualified compile unit. See Table 3 on page 127.

%CU is equivalent to %PROGRAM.

%PROGRAM

Is equivalent to %CU. See Table 3 on page 127.

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 202 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/C++ programs” on page 138, “Debug Tool evaluation of COBOL expressions” on page 181, or “PL/I expressions” on page 191.

You can only use expressions for variables contained in the current enclave.

Load_Module_Name

A *load_module_name* is the name of a file, object, or Dynamic Link Library (DLL) that has been loaded by a supported HLL load service, or a subsystem. For example, an enclave can contain load modules, which in turn contain compile units.

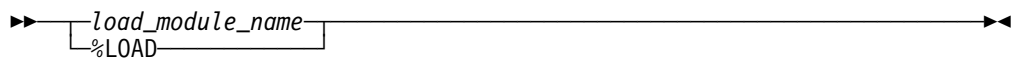
For C, escape sequences in load module names that are specified as strings are not processed if the string is part of a qualification statement.

If omitted from a name that allows it as a qualifier, the current load module qualification is assumed. It can be changed using the SET QUALIFY LOAD command.

If two enclaves contain duplicate modules, references to compile units in the modules will be ambiguous, and will be flagged as errors. However, if the compile unit is in the currently executing load module, that load module is assumed and no check for ambiguity will be performed. Therefore, for Debug Tool, load module names must be unique.

Load_Spec

A *load_spec* identifies a load module in the program being debugged.



load_module_name

A valid load module name; see “Load_Module_Name.” 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 load module. See Table 3 on page 127.

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 load module name qualifier. For example, when the current programming language setting is C, `mod::>cu:>proc:>struc1.struc2.array[23]`.

When the current programming language setting is C/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

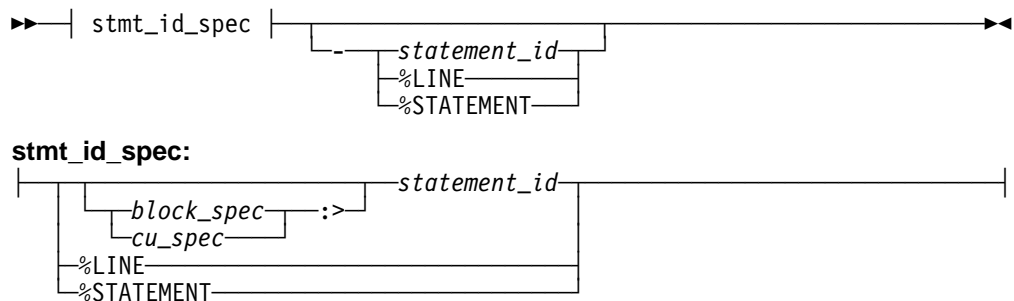
A *statement_id* identifies an executable statement in a manner appropriate for the current programming language. This can be a statement number, sequence number, or source line number. The statement id is an *integer* or *integer.integer* (where the first *integer* is the line number and the second *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/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.



block_spec

A valid block specification; see “Block_Spec” on page 199. 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.”

`%LINE`

Represents the currently suspended source statement or line. See Table 3 on page 127. `%LINE` is equivalent to `%STATEMENT`.

%STATEMENT

Is equivalent to %LINE. See Table 3 on page 127.

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/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 load module name qualifier. For example:

```
mod::>proc1:>proc2:>block1:>start
```

The form of a label depends on the current programming language:

- In C/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 xv 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 might be raised when evaluating the expression. See “SET WARNING (C/C++ and PL/I)” on page 327 for specific information.
- Although the PL/I compiler supports the concatenation of GRAPHIC strings, Debug Tool does not.

Examples:

- 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 202.

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 in the AT command, 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 10.

Table 10 (Page 1 of 2). Summary of AT Commands

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: <ul style="list-style-type: none"> • 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 DATE	For COBOL, gives Debug Tool control for each date processing statement within the specified block.
AT DELETE	Gives Debug Tool control when a load module 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 load module is loaded.
AT OCCURRENCE	Gives Debug Tool control on a language or Language Environment condition or exception.
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.

Table 10 (Page 2 of 2). Summary of AT Commands

AT TERMINATION	Gives Debug Tool control when the application program is terminated.
----------------	--

Usage Notes:

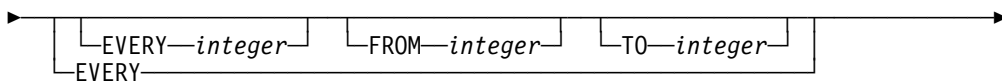
- To set breakpoints at specific locations in a program, Debug Tool depends on that program being loaded into storage. If you issue an AT command for a specific ENTRY, EXIT, LABEL, LINE, or STATEMENT breakpoint and the program is not known by Debug Tool, a warning message is issued and the breakpoint 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.
- If you want breakpoints to only stop your program under certain conditions, you can use a combination of the AT command and the IF command to establish a conditional breakpoint. For more information, see the following sections titled “Stopping on a line only if a condition is true”: for C programs, page 51; for C++ programs, page 60; for COBOL programs, page 71; and for PL/I programs, page 80.

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.

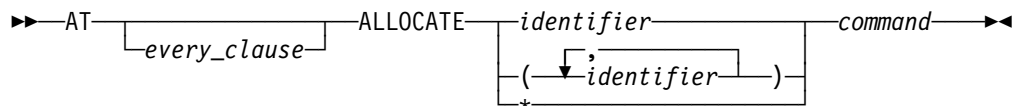
```
AT EVERY 3 FROM 48 TO 59 STATEMENT 50;
```

- 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 208.

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');
```

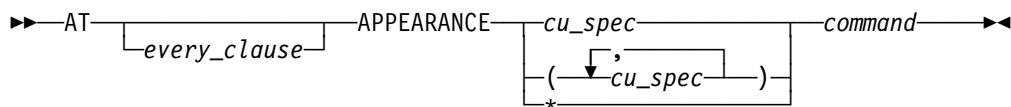
```
AT ALLOCATE temp;
  BEGIN;
    AT CHANGE temp;
    BEGIN;
      LIST (temp);
    GO;
  END;
GO;
END;
GO;
```

```
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 load module being loaded. However, for modules 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 208.

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 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.
- 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 load module name, the AT APPEARANCE breakpoint will only be recognized for the compile unit that is contained in the specified load module. For example, if a compile unit *cux* which is in load module *loady* appears, the breakpoint AT APPEARANCE *loadx* :>*cux* will not be TRIGGERed.
- If the compile unit is *not* qualified with a load module name, the current load module qualification is not 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/C++, AT APPEARANCE is not triggered for compile units that reside in a loaded module since the compile units are known at the time of the load.
- For C/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.

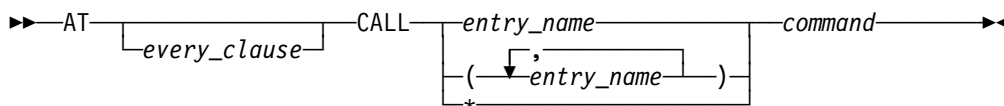
```
AT APPEARANCE cuy PERFORM
  AT EXIT cuy:>blocky LIST ('Exiting blocky.');
```

```
  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 208.

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 5, “Compiling a COBOL program with the compile-time TEST option” on page 10, or “Compiling a PL/I program with the compile-time TEST option” on page 13.

- A breakpoint set with AT CALL for a call to a C, 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, entry_name can refer to a method as well as a procedure.
- 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/C++ program has unresolved entry points or entry variables, issue AT CALL 0.
- To be able to set CALL breakpoints in C++, you must compile your program with the compile-time TEST option.
- 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.

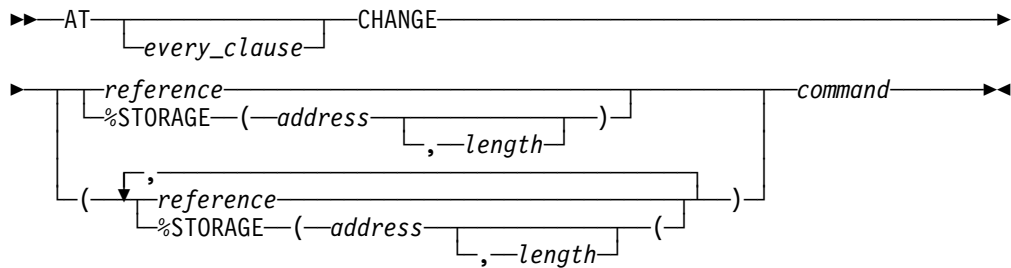
```
AT CALL *;
```

- If the program invokes function `badsubr`, intercept the call, set variable `varb1` to 50, and then bypass the target function. The current programming language setting is C.

```
AT CALL badsubr {
    varb1 = 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 208.

reference

A valid Debug Tool reference in the current programming language; see “References” on page 202.

%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 entered. 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 entered first, but the CHANGE x breakpoint was TRIGGERed first (because 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 on Debug Tool, the CHANGE %STORAGE 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.
- 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 module 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/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 module 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 182.
 - 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 or double 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 varb11 or varb12 is found to have a changed value. The current programming language setting is COBOL.

```
AT CHANGE (varb11, varb12) 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.

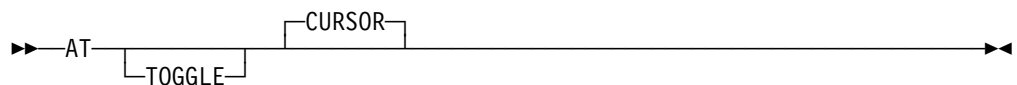
```
AT CHANGE %STORAGE (0x000022222, 8)
    LIST "Storage has changed at hex address 22222";
```

- 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).

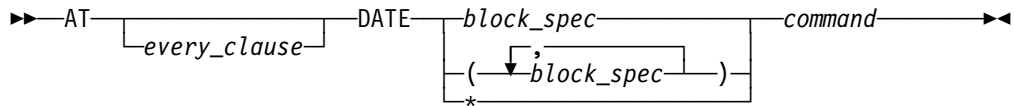
Example:

Define a PF key to toggle the breakpoint setting at the cursor position.

```
SET PF10 = AT TOGGLE CURSOR;
```

AT DATE (COBOL)

Gives Debug Tool control for each date processing statement within the specified block. A date processing statement is a statement that references a date field, or an EVALUATE or SEARCH statement WHEN phrase that references a date field.



every_clause

As described under “Every_clause” on page 208.

block_spec

A valid block specification; see “Block_Spec” on page 199.

- * Sets a breakpoint at every date processing statement.

command

A valid Debug Tool command.

Usage Note:

When AT DATE is set, execution is halted only for COBOL compile units compiled with the DATEPROC compiler option.

Examples:

- Each time a date processing statement is encountered in the nested subprogram subrx, display the location of the statement.

```
AT DATE subrx QUERY LOCATION;
```

- Each time a date processing statement is encountered in the compile unit, display the name of the compile unit.

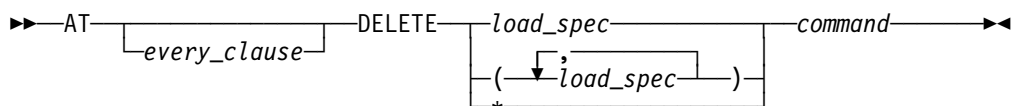
```
AT DATE * LIST %CU;
```

- Each time a date processing statement is encountered in the compile unit, display the location of the statement, list a specific variable, and resume running the program.

```
AT DATE * PERFORM
    QUERY LOCATION;
    LIST DATE-FIELD
    GO;
END-PERFORM;
```

AT DELETE

Gives Debug Tool control when a load module is removed from storage by a Language Environment 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 208.

load_spec

A valid load module specification; see “Load_Spec” on page 202.

- * Sets a breakpoint at every DELETE of any load module.

command

A valid Debug Tool command.

Usage Notes:

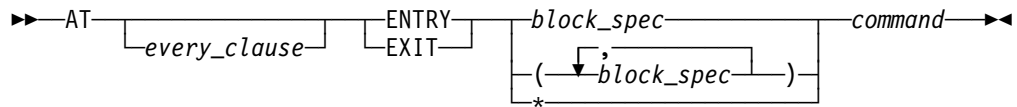
- Debug Tool gains control for deletes that are affected by the Language Environment delete service. If a load module is deleted using the OS DELETE macro, Debug Tool is not informed. This can cause errors if Debug Tool attempts to operate on any part of the deleted load module.
- AT DELETE cannot specify the initial load module.
- 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.

Examples:

- Each time a load module is deleted, request input from the terminal.
AT DELETE *;
- Stop watching variable var1:>x when load module mymod is deleted.
AT DELETE mymod 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 208.

block_spec

A valid block specification; see “Block_Spec” on page 199.

- * 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 210.

- 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 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.
- 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.

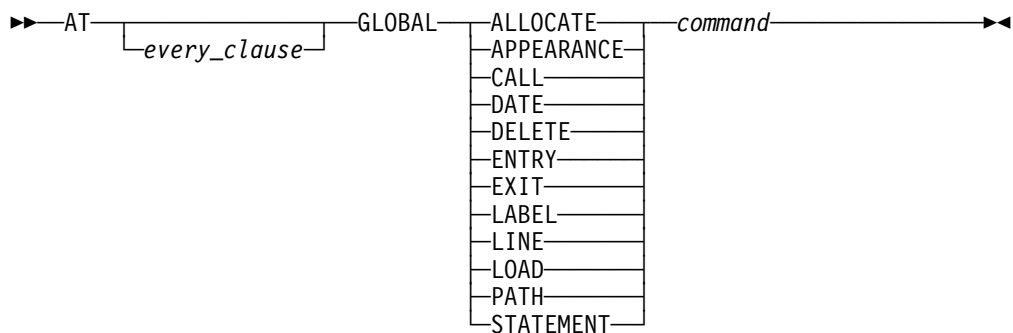
```
AT ENTRY subrx PERFORM
    SET ix TO 5;
    GO;
END-PERFORM;
```

- 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 208.

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;
```

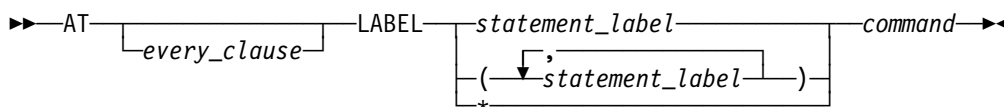
only benefits and scale are listed when your program reaches the exit point of block `table1`. The second `AT EXIT` replaces the first because the breakpoints are defined for the same location. However, if you define the following GLOBAL breakpoint:

```
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 208.

statement_label

A valid source label constant; see “Statement_Label” on page 204.

* Sets a breakpoint at every LABEL.

command

A valid Debug Tool command.

Usage Notes:

- 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 load module qualifiers.
- For C/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 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.
- You cannot set LABEL breakpoints at, for example, PL/I label variables.

- LABEL breakpoints for label constants in a fetched, loaded program or DLL 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.

- For COBOL, a reference to a label or a label constant can take either of the following forms:

- *name*

This form is used to refer to a section name or the name of a paragraph contained in not more than one section of the block.

- *name1 OF name2* or *name1 IN name2*

This form is used to refer to a paragraph contained within a section if the paragraph name exists in other sections in the same block. You can use either OF or IN, but Debug Tool only uses OF for output to the log file.

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 */
```

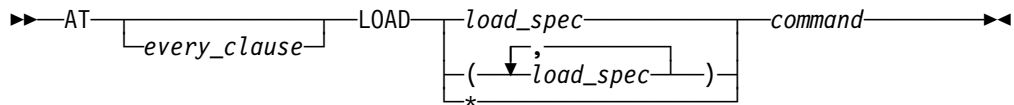
AT LINE

See “AT STATEMENT” on page 230.

AT LOAD

Gives Debug Tool control when the specified load module 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 load module, it is not raised again unless either the load module is released and fetched again or another load module 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 208.

load_spec

A valid load module specification; see “Load_Spec” on page 202.

- * Sets a breakpoint at every LOAD of any load module.

command

A valid Debug Tool command.

Usage Notes:

- Debug Tool gains control for loads that are affected by the Language Environment load service. If a load module is loaded using the OS LOAD macro or EXEC CICS LOAD, Debug Tool is not informed.
- AT LOAD can be used to detect the loading of specific language library load modules; however, the loading of language library load modules does not TRIGGER an AT GLOBAL LOAD or AT LOAD *.
- AT LOAD cannot specify the initial load module 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 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.
- AT LOAD on an implicitly or explicitly loaded DLL is not supported by Debug Tool.
- Debug Tool recognizes an implicitly loaded DLL only after a compile unit in that DLL is called. For example, if LIST NAMES CUS is issued after an implicit load of a DLL, there will be no entry in the output of the command of the DLL.
- Depending on the version of the C/C++ compiler used, Debug Tool might recognize a compile unit in a DLL only after it has had a function in it called. For example, if a DLL contains a function fn1 in CU file1 and it contains a function fn2 in CU file2, a call to fn1 **will not** enable

Debug Tool to recognize file2, only file1. Similarly, a call to fn2 **will not** enable Debug Tool to recognize file1.

- At the triggering of a LOAD breakpoint for C/C++ and PL/I, Debug Tool has enough information about the loaded module to set breakpoints and examine variables of static and extern storage classes.
- At the triggering of a LOAD breakpoint for COBOL and C/C++ DLL's, Debug Tool does not have enough information about the loaded module to set breakpoints in blocks contained within the module. At the triggering of an APPEARANCE breakpoint, however, you can set such breakpoints.

Examples:

- Print a message when load module mymod is loaded. The current programming language setting is either C/C++ or COBOL.

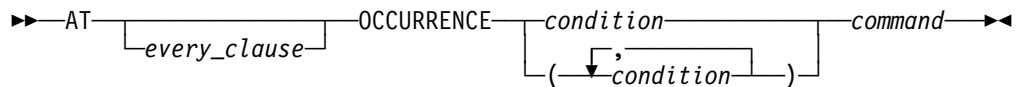
```
AT LOAD mymod LIST ("Load module mymod has been loaded");
```

- Establish an entry breakpoint when load module 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 Language Environment condition or exception.



every_clause

As described under “Every_clause” on page 208.

condition

A valid condition or exception. This can be either an Language Environment symbolic feedback code, or a language-oriented keyword or code, depending on the current programming language setting.

Following are the C/C++ condition constants; they must be uppercase and not abbreviated:

SIGABND	SIGILL	SIGTERM
SIGABRT	SIGINT	SIGUSR1
SIGFPE	SIGIOERR	SIGUSR2
	SIGSEGV	THROWOBJ

When a C++ user specifies AT CONDITION THROWOBJ, Debug Tool transfers control to the user at the point of the throw in C++ code.

PL/I condition constants can be used as well. See “ON command (PL/I)” on page 289 for information about valid condition names.

There are no COBOL condition constants. Instead, an Language Environment symbolic feedback code must be used, for example, CEE347. For symbolic

feedback codes for Language Environment callable services, see *OS/390 Language Environment Programming Guide*.

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/C++:

When a C/C++ or an Language Environment condition occurs during your session, the following series of events takes place:

1. Debug Tool is invoked before any C/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/C++ signals and Language Environment 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/C++ and Language Environment designations, the Language Environment breakpoint is the only breakpoint triggered. Any command list associated with the C condition is not executed. Table 22 on page 348 lists the Language Environment conditions and their C/C++ equivalents. Also see *OS/390 Language Environment 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 module has been initialized.
- For C/C++ and PL/I, certain Language Environment conditions map to C/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 Language Environment condition name and the other could be entered with the C/C++ SIGxxx condition constant. In this case, the AT OCCURRENCE breakpoint for the

Language Environment condition name is TRIGGERed and the AT OCCURRENCE breakpoint for the C/C++ condition constant is *not*. However, if an AT OCCURRENCE breakpoint for the Language Environment condition name is not defined, the corresponding mapped C/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 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, Debug Tool detects Language Environment conditions. If a Language Environment 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 *OS/390 Language Environment Debugging Guide and Run-Time Messages* for a list of Language Environment conditions.

- For PL/I, Debug Tool detects Language Environment 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 289.
- 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 *PL/I for MVS &*

VM Language Reference. Also see OS/390 Language Environment 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 Language Environment 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.

```
▶▶—AT—┬──────────┬──PATH—command──────────▶▶
      |every_clause|
```

every_clause

As described under “Every_clause” on page 208.

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/C++” on page 140 “Using Debug Tool variables in COBOL” on page 175, or “Using Debug Tool variables in PL/I” on page 187.
- 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 C, COBOL and PL/I, you can set PATH breakpoints if you compiled with the PATH suboption. For more information, see:
 - “Compiling a C program with the compile-time TEST option” on page 5

- “Compiling a COBOL program with the compile-time TEST option” on page 10
- “Compiling a PL/I program with the compile-time TEST option” on page 13
- For C++, you can set PATH breakpoints at any time. For more information, see “Compiling a C++ program with the compile-time TEST option” on page 9.
- 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 10 or “Compiling a PL/I program with the compile-time TEST option” on page 13.

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.

►► AT integer ◄◄

integer

Selects a relative statement (for C/C++ and PL/I) or a relative verb (for COBOL) within the line. The default value is 1.

Example:

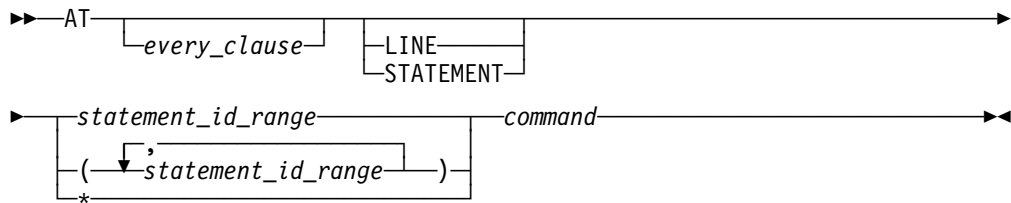
Set a breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
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 208.

statement_id_range

A valid statement id or statement id range; see “Statement_Id_Range and Stmt_Id_Spec” on page 203.

* 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 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.
- 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/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 load modules 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

BEGIN

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 reenables or reset it.
- 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.

Examples:

- When the program ends, check the Debug Tool environment to see what files have not been closed.

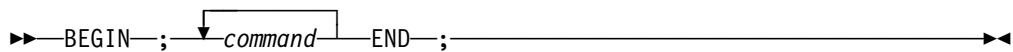
```
AT TERMINATION DESCRIBE ENVIRONMENT;
```

- 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.



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 `D0`.

Examples:

- Set a breakpoint at statement 320 listing the value of variable `x` and assigning the value of 2 to variable `a`.

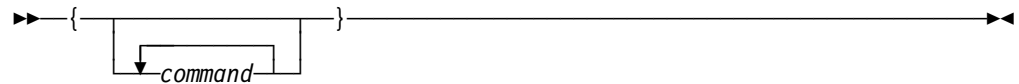
```
AT 320 BEGIN;  
  LIST (x);  
  a = 2;  
END;
```

- 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/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 load module `a` is fetched.

```
AT LOAD a {
  AT ENTRY a;
  GO;
}
```

break command (C/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.

```
►►—break—;—————►◄
```

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 Language Environment run-time dump be produced. The C/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 C++, calls can be made to any user function as long as the function is declared as:

```
extern "C"
```

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 11.

Table 11. Summary of CALL Commands

CALL %DUMP	Invokes the Language Environment dump service to obtain a formatted dump.
CALL entry_name (COBOL)	Invokes an entry name in the application program (COBOL).
CALL procedure	Invokes a procedure that has been defined with the PROCEDURE command.

CALL %DUMP

Invokes the Language Environment 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:

THREAD(ALL|CURRENT)

Dumps the current thread or all threads associated with the current enclave. The default is to dump only the current thread. Only one thread is supported in Language Environment. For enclaves that consist of a single thread, THREAD(ALL) and THREAD(CURRENT) are equivalent.

THREAD can be abbreviated as THR.

CURRENT can be abbreviated as CUR.

TRACEBACK

Requests a traceback of active procedures, blocks, condition handlers, and library modules 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)
C++	TEST
COBOL	TEST or TEST(h,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 Language Environment 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.

FNAME(s)

Specifies the ddname of the file where the dump report is written.

The default ddname CEEDUMP is used if this option is not specified.

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 Language Environment 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 Language Environment 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:

- 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 Language Environment 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 *OS/390 Language Environment Programming Guide* for additional details on the CEE3DMP dump options.

- When you use CALL %DUMP, one of the following ddnames must be allocated for you to receive a formatted dump:
 - CEEDUMP (default)
 - SYSPRINT.

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 *OS/390 Language Environment 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 modules. Identify the dump as "Dump after read".

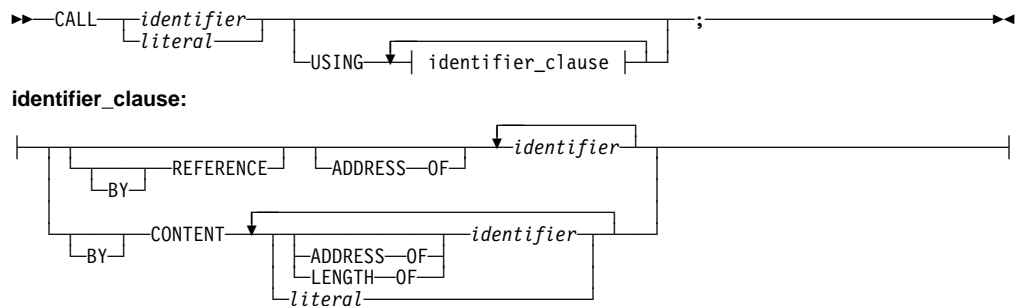
```
CALL %DUMP ("TRACEBACK", "Dump after read");
```

- Call the Language Environment 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 *COBOL Language Reference publications* for an explanation of the following COBOL keywords: ADDRESS, BY, CONTENT, LENGTH, OF, REFERENCE, USING.
- An entry_name cannot refer to a method.

CLEAR

- A windowed date field cannot be specified as either the identifier containing the entry name, or an identifier in the USING phrase.

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:

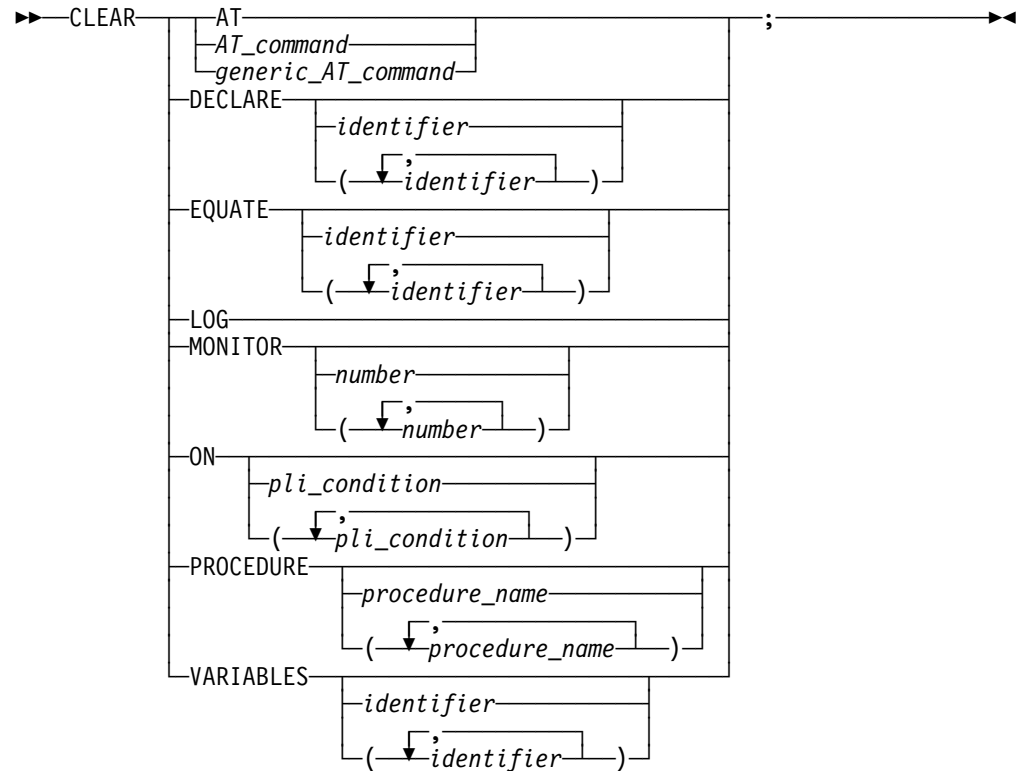
- Since the Debug Tool procedure names are always uppercase, the procedure name is converted to uppercase even for programming languages that have mixed-case symbols.
- The CALL keyword is required even for programming languages that do not use CALL for subroutine invocations.
- The CALL command is restricted 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 10 on page 207 for a list of valid AT commands. The AT command must be complete except that the *every_clause* and *command* are omitted.

generic_AT_command

A valid AT command without operands. It can be one of the following: ALLOCATE, APPEARANCE, CALL, CHANGE, CURSOR, DATE, DELETE, ENTRY, EXIT, LABEL, LOAD, OCCURRENCE, PATH, STATEMENT (the LINE keyword can be used in place of STATEMENTS), or TERMINATION.

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.

CLEAR

LOG

Erases the log file and clears out the data being retained for scrolling. In line mode, CLEAR LOG clears only the log file.

For MVS Only: If the log file is directed to a SYSOUT type file, CLEAR LOG will not clear the log contents in the file.

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).

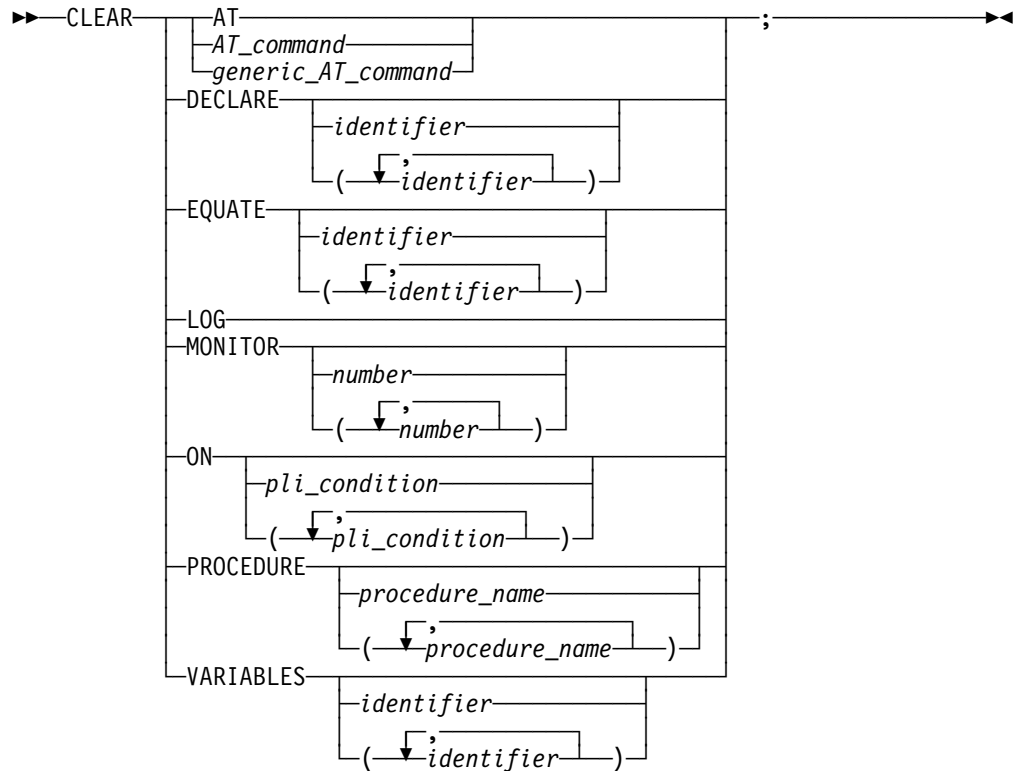
```
AT 50;
AT 56;
AT 55 LIST (r, c);
LIST AT;
CLEAR AT 50;
LIST AT;
CLEAR AT;
LIST AT;
```

- 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:

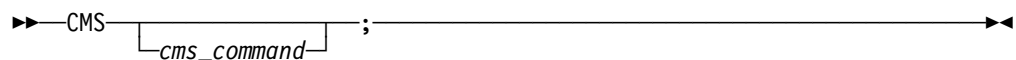
Clear a breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

```
CLEAR 3
```

No space is needed as a delimiter between the keyword and the integer; hence, CLEAR 3 is equivalent to CLEAR3.

CMS command (VM)

The CMS command lets you issue certain CMS subset commands during a Debug Tool session. The CMS keyword cannot be abbreviated.

*cms_command*

A CMS system command that can be issued while in the CMS editor. If omitted, CMS subset mode is entered.

Usage Notes:

- When not operating interactively, a *cms_command* must be supplied.

- When operating interactively, if no *cms_command* is specified, CMS subset mode is entered. While in CMS subset mode, a subset of CMS commands (that is, CMS system commands that can be issued while in the CMS editor) can be performed repeatedly. To return to Debug Tool, type RETURN.
- See also “SYSTEM command” on page 333.

Example:

- List all the files that are named free on the a disk.
CMS LIST free * a;
- Copy the contents of myprog script a into ourprog script a.
CMS COPYFILE myprog script a ourprog script a;

COMMENT command

The COMMENT command can be used to insert commentary in to the session log. The COMMENT keyword cannot be abbreviated.

►► COMMENT ————— ; ◀◀
 └── commentary ─┘

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 288).
- 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.
- COMPUTE cannot be used to perform a computation with a windowed date field if the *expression* consists of more than one operand.
- Any expanded date field specified as an operand in the *expression* is treated as a nondate field.
- The result of the evaluation of the *expression* is always considered to be a nondate field.
- 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.

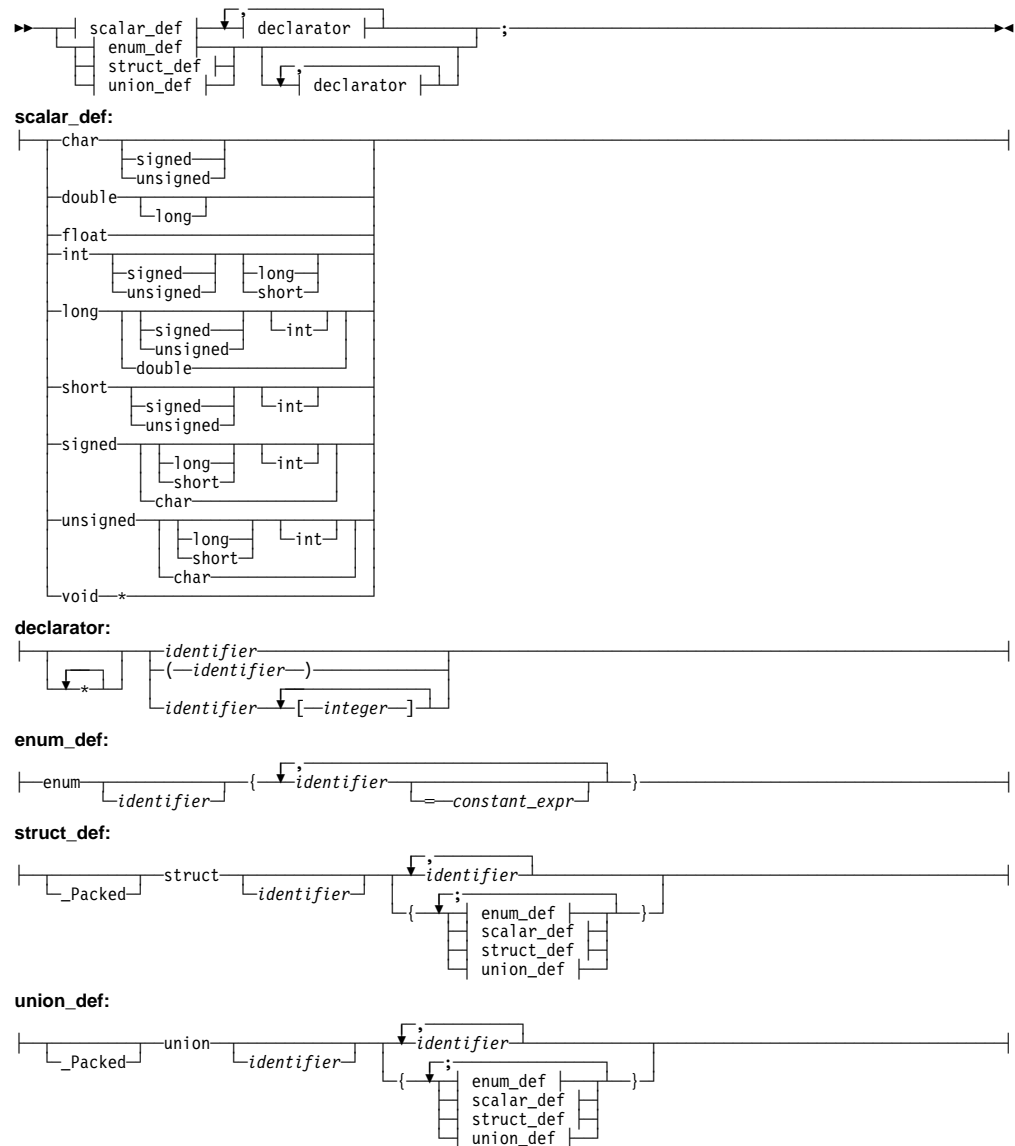
```
CURSOR;
```

Declarations (C/C++)

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.

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.

Declarations



* 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/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 ccx., 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 12 on page 250 table can be declared.
- See the C and C++ Language References for an explanation the following keywords:

<code>char</code>	<code>short</code>
<code>double</code>	<code>signed</code>
<code>enum</code>	<code>struct</code>
<code>float</code>	<code>union</code>
<code>int</code>	<code>unsigned</code>
<code>long</code>	<code>void</code>
<code>_Packed(1)</code>	

(1) `_Packed` is not supported in C++.

C/C++ compatible attributes

Debug Tool allows you, while working in one language, to declare session variables you can continue to use after calling in a load module of a different language.

Table 12 on page 250 shows how session data attributes are mapped across programming languages. Attributes not shown in the table cannot be mapped to other programming languages.

Remember that, when declaring session variables, C variables are case-sensitive. This means that only C session variables whose names are all uppercase can continue to be accessed when the programming language setting is changed to COBOL or PL/I. Those declared when the programming language setting is COBOL or PL/I are converted to uppercase and are accessible when the programming language is C.

Variables with incompatible attributes cannot be accessed from the other programming languages, but replace variables with the same names in the other languages (if uppercase). For example, COBOL has no equivalent to C's long double. If a COBOL session variable `x` is declared, it is converted to `X`; if the current programming language setting is changed to C and a C session variable `X` is declared as a long double, C's variable `X` replaces COBOL's variable `X`. If `x` is declared instead of `X` when the programming language setting is changed to C, COBOL's variable `X` will not be replaced. There exists no COBOL counterpart to the C variable.

Table 12. C/C++ Attribute Mappings

Machine Value	C/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).

Examples:

- Define two C integers.
`int myvar, hisvar;`
- Define an enumeration variable status that represents the following values:

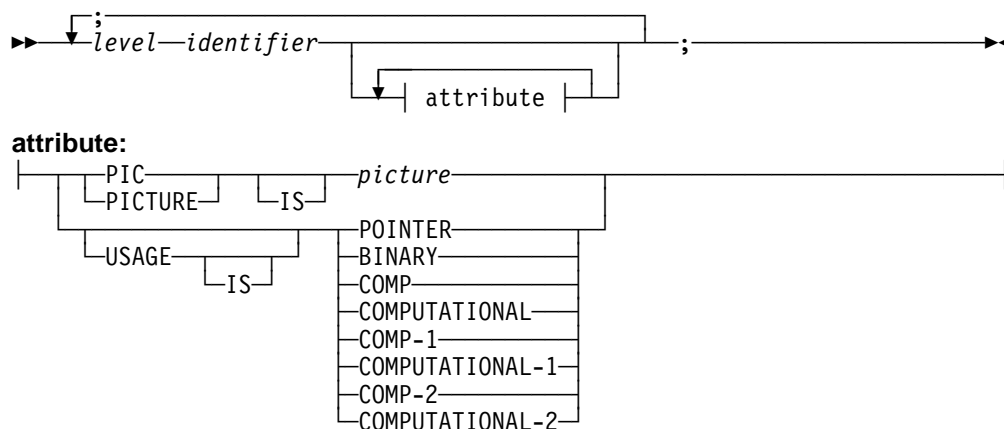
Enumeration Constant	Integer Representation
run	0
create	1
delete	5
suspend	6

```
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)

Use declarations to declare temporary variables effective during a Debug Tool session. Session variables remain in effect for the entire debug session, or process in which they were declared. Variables declared with declarations can be used in other Debug Tool commands as if they were declared to the compiler. Declared variables are removed when your Debug Tool session ends or when the CLEAR command is used to remove them. The keywords cannot be abbreviated.



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 13 on page 252 can be declared.
- See *COBOL Language Reference publications* for an explanation of the following COBOL keywords:

```

  BINARY
  COMP
  COMPUTATIONAL
  IS
  PIC
  PICTURE
  POINTER
  USAGE
  
```

COBOL compatible attributes

You can declare session variables, while working in one language, that you can continue to use after calling in a load module of a different language. Table 13 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/C++ variables are case-sensitive. This means that only C/C++ session variables whose names are all uppercase can continue to be accessed when the programming language setting is changed to COBOL. Session variables declared when the programming language setting is COBOL are converted to uppercase and are accessible when the programming language is C/C++.

Variables with incompatible attributes cannot be accessed from the other programming languages, but replace variables with the same names in the other languages (if uppercase). For example, COBOL has no equivalent to C's long double. If a COBOL session variable `x` is declared, it is converted to `X`; if the current programming language setting is changed to C and a C session variable `X` is declared as a long double, C's variable `X` replaces COBOL's variable `X`. If `x` is declared instead of `X` when the programming language setting is changed to C, COBOL's variable `X` will not be replaced. There exists no COBOL counterpart to the C variable.

Table 13. COBOL Attribute Mappings

Machine Value	COBOL Value	C/C++ Value	PL/I Value
byte	PICTURE X	unsigned char	CHAR(1)
byte string	PICTURE X(j)	unsigned char[j]	CHAR(j)
halfword	PICTURE S9(j<=4) USAGE BINARY	signed short int	FIXED BIN(15,0)
fullword PICTURE S9(j<=4)	PICTURE S9(4<j<=9) USAGE BINARY	signed long int	FIXED BIN(31,0)
floating point	USAGE COMP-1	float	FLOAT BIN(21) or FLOAT DEC(6)
long floating point	USAGE COMP-2	double	FLOAT BIN(53) or FLOAT DEC(16)
extended floating point	n/a	long double	FLOAT BIN(109) or FLOAT DEC(33)
fullword pointer	USAGE POINTER	void *	POINTER

Note:

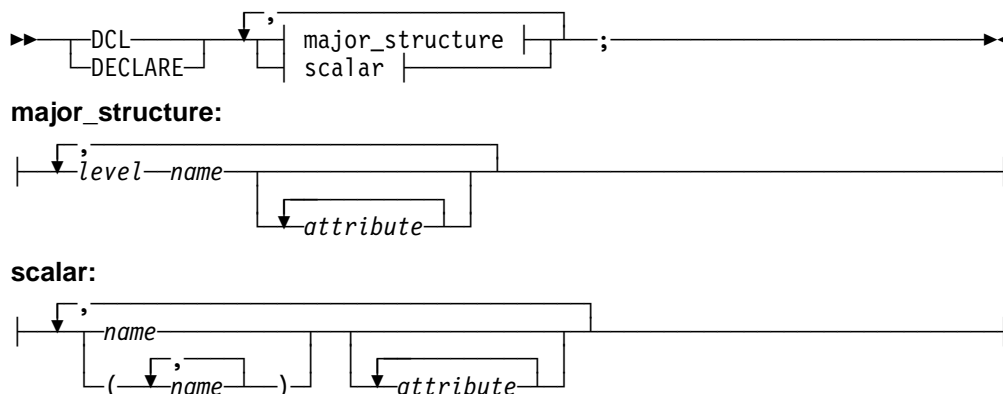
When registering session variables in PL/I, the DECIMAL type is always the default. For example, if C declares a float, PL/I registers the variable as a FLOAT DEC(6) rather than a FLOAT BIN(21).

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 DECLARE command declares temporary variables effective during a Debug Tool session. Variables declared this way can be used in other Debug Tool commands as if they were declared to the compiler. They are removed with the CLEAR command or when your Debug Tool session ends. The keywords cannot be abbreviated.



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 include:

BINARY	CPLX	FIXED	LABEL	PTR
BIT	DECIMAL	FLOAT	OFFSET	REAL
CHARACTERS	EVENT	GRAPHIC	POINTER	VARYING
COMPLEX				

Acceptable PL/I storage attributes include:

BASED ALIGNED UNALIGNED

Pointers cannot be specified with the BASED option.

Only simple factoring of attributes is allowed. DECLAREs such as the following are not allowed:

DECLARE

```
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 *PL/I 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 14 on page 255 can be shared.
- See *PL/I 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	

PL/I compatible attributes

While working in one language, you can declare session variables that you can continue to use after calling in a load module of a different language. Table 14 on page 255 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/C++ variable names are case-sensitive. When the current programming language is C/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/C++, within C/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/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 14. PL/I Attribute Mappings

Machine Value	PL/I Value	C/C++ Value	COBOL Value
byte	CHAR(1)	unsigned char	PICTURE X
byte string	CHAR(j)	unsigned char[j]	PICTURE X(j)
halfword	FIXED BIN(15,0)	signed short int	PICTURE S9(j≤4) USAGE BINARY
fullword	FIXED BIN(31,0)	signed long int	PICTURE S9(4<j≤9) USAGE BINARY
floating point	FLOAT BIN(21) or FLOAT DEC(6)	float	USAGE COMP-1
long floating point	FLOAT BIN(53) or FLOAT DEC(16)	double	USAGE COMP-2
extended floating point	FLOAT BIN(109) or FLOAT DEC(33)	long double	n/a
fullword pointer	POINTER	*	USAGE POINTER

Note:

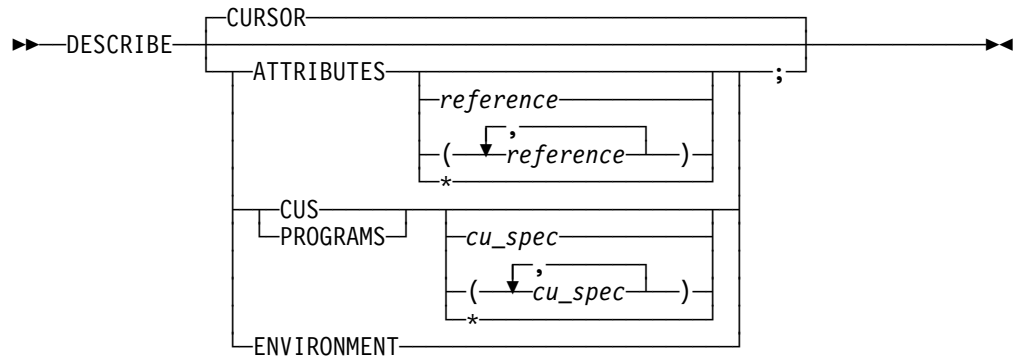
When registering session variables in PL/I, the DECIMAL type is always the default. For example, if C declares a float, PL/I registers the variable as a FLOAT DEC(6) rather than a FLOAT BIN(21).

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/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 the following points:

In C/C++, this can be a valid expression, enumeration tag, structure tag, or union tag identifier. For a C/C++ expression, the type is the only attribute displayed. You must use enum, struct, or union when referencing the C/C++ tag; see “Declarations (C/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 per their definitions (that is, if they are defined as alphanumeric data or as arrays).

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 202.

*

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 closed 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/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 COBOL, if DESCRIBE ATTRIBUTES * is specified and your compile unit is large, you might receive an *out of storage* error message.
- 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 AT command must be complete except that the *every_clause* and *command* are omitted. Valid forms are the same as those allowed with CLEAR AT.

DISABLE

Usage Notes:

- To reenable a disabled AT command, use the `ENABLE` command.
- Disabling an AT command does not affect 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/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 13 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/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/C++ expression.

The body of the loop is performed before the while clause (the controlling part) 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:

The following command prompts you to enter a 1. If you enter a 1, the command ends execution. Otherwise, the command displays another prompt.

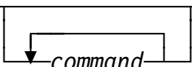
```
int reply1;

do {
    printf("Enter a 1.\n");
    scanf("%d", &reply1);
} while (reply1 != 1);
```

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.

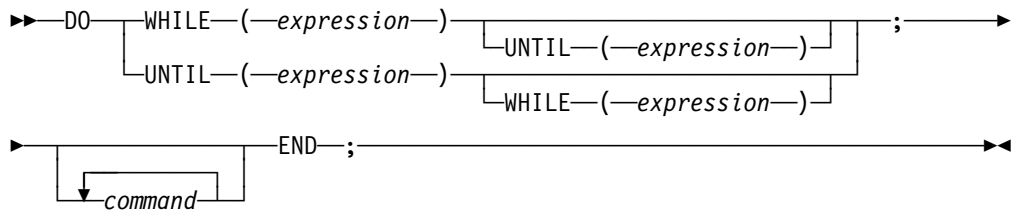
Simple

►►DO—;◄◄
◄◄

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

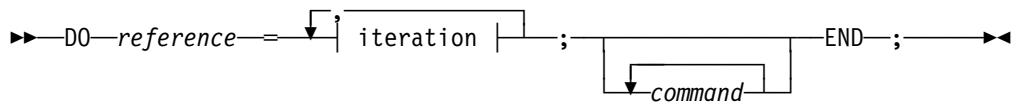
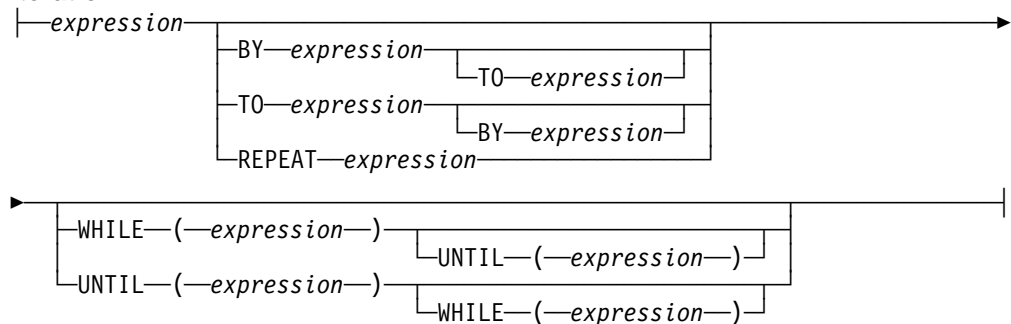
A valid Debug Tool PL/I Boolean expression.

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 T0 *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.

T0 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 T0 *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 T0 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 AT command must be complete except that the *every_clause* and *command* 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:

Reenable 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/C++ or PL/I) or a relative verb (for COBOL) within the line. The default value is 1.

Example:

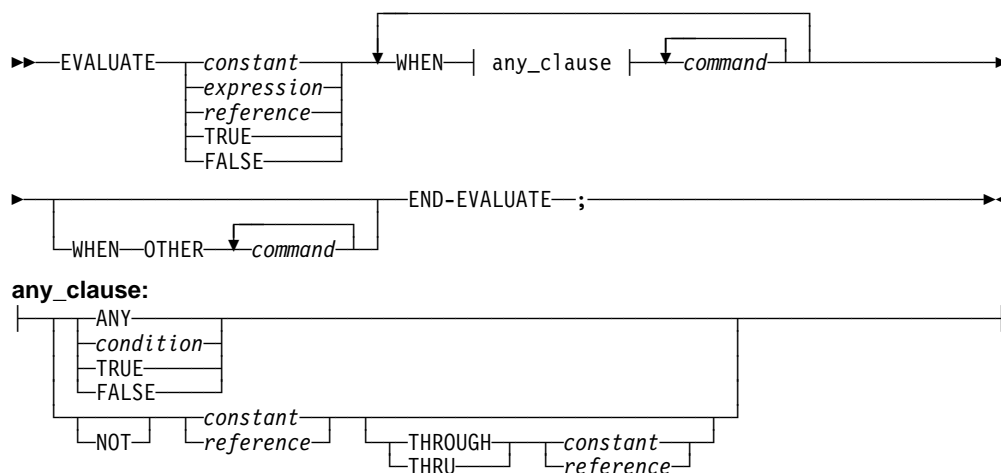
Enable the breakpoint at the third statement or verb in the line (typed in the prefix area of the line where the statement is found).

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 *COBOL Language Reference publications* for an explanation of the following COBOL keywords:

ANY
FALSE
NOT
OTHER
THROUGH
THRU
TRUE
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/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/C++ expression. Assignment is affected by including one of the C/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:

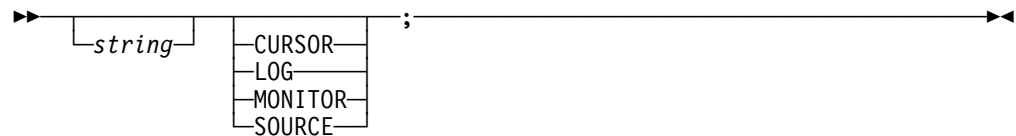
- Initialize the variables x, y, z and note that function invocations are supported.

```
x = 3 + 4/5;
y = 7;
z = 8 * func(x, y);
```

- Increment y and assign the remainder of the integer division of ω by 4 to α .
 $\alpha = (y++, \omega \% 4);$

FIND command

The FIND command provides full-screen, line, 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
"ABC"

C++
"IntLink::~"

COBOL
"A5"
'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 listing window.

Usage Notes:

- Window defaulting can be controlled by the SET DEFAULT WINDOW command.
- If the current programming language setting is C/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 line or 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 line and batch mode.

Example:

Indicate that you want to search the monitor window for the name `myvar`. The current programming language setting is either C/C++ or COBOL.

```
FIND "myvar" MONITOR;
```

for command (C/C++)

The `for` command provides iterative looping similar to the C/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.

```
►► for ( expression ; expression ; expression ) ►
► command ►►
```

expression

A valid Debug Tool C/C++ expression.

command

A valid Debug Tool command.

Debug Tool evaluates the first *expression* 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 the second *expression* 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 the second expression, 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 the third *expression* 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 this expression.

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

HLL or Language Environment condition

Usage Notes:

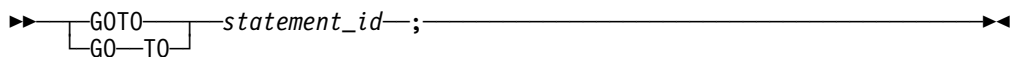
- If G0 is specified in a command list (for example, as the subject of an IF command or WHEN clause), all subsequent commands in the list are ignored.
- If G0 is specified within the body of a loop, it causes the execution of the loop to end.
- To suppress the logging of G0 commands, use the SET ECHO command.
- G0 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 208 for an explanation on processing multiple AT-conditions.

Examples:

- Resume execution.
G0;
- Resume execution and bypass user and system actions for the AT-condition that caused the breakpoint.
G0 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 G0 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.

►►  *statement_id*—; ►►

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 might be needed.
- Statement GOTO's are not restricted if the program is compiled with minimum optimization.
- Because statements can 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 131.

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 ─┴─┘
      └─ LABEL ─┘

```

statement_label

A valid statement label within the currently executing program or, in PL/I, a label variable. See “Statement_Label” on page 204.

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 131.
- 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).
- 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 load module 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 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. The label itself can take either of two forms:
 - name, where name is a section name, or the name of a paragraph not within a section or in only one section of the block.
 - name1 OF name2 or name1 IN name2, where name1 is duplicated by one or more other paragraphs in one or more other sections in the block. You can use either OF or IN, but Debug Tool always logs OF.
- 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/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/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.

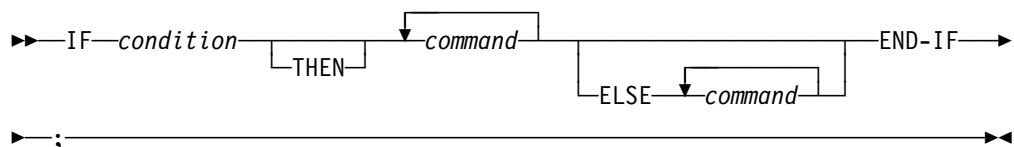
```
if (score >= 90)
    grade = "A";
```

- 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.
- Comparison combinations with windowed date fields are not supported.
- Comparisons between expanded date fields with different DATE FORMAT clauses are not supported.
- Only the comparison combinations listed in “Allowable comparisons for the Debug Tool IF command” on page 350, are supported.

Example:

To substitute the input that would have come from the ddname specified by the SET INTERCEPT ON command with your desired input, enter:

```
INPUT text you want to input ;
```

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.

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.

```

▶▶ IF expression THEN command
    └── ELSE command ───▶
  
```

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 G0;
```

- 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 G0;
```

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.

►►—IMMEDIATE—*command*—————◄◄

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/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/C++ and COBOL)” on page 314 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;
```

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 15.

Table 15 (Page 1 of 2). Summary of LIST Commands

LIST (blank)	Displays Source Identification panel
LIST AT	Lists the currently defined breakpoints.
LIST CALLS	Displays the dynamic chain of active blocks.
LIST CURSOR	Displays the variable pointed to by the cursor.

Table 15 (Page 2 of 2). Summary of LIST Commands

LIST expression	Displays values of expressions.
LIST FREQUENCY	Lists statement execution counts.
LIST LAST	Displays 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	Displays the current register contents.
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 291 for additional information.

LIST AT

Lists the currently defined breakpoints, including the action taken when the specified breakpoint is activated.

```

graph LR
    LIST --> AT_command
    LIST --> Semicolon[";"]
    AT_command --> AT
    AT_command --> Bar["|"]
    AT --> ENABLED
    AT --> DISABLED
    Bar --> ALLOCATE1[ALLOCATE]
    Bar --> APPEARANCE1[APPEARANCE]
    Bar --> CALL1[CALL]
    Bar --> CHANGE[CHANGE]
    Bar --> DATE1[DATE]
    Bar --> DELETE1[DELETE]
    Bar --> ENTRY1[ENTRY]
    Bar --> EXIT1[EXIT]
    Bar --> GLOBAL
    Bar --> LABEL1[LABEL]
    Bar --> LINE1[LINE]
    Bar --> LOAD1[LOAD]
    Bar --> OCCURRENCE[OCCURRENCE]
    Bar --> PATH1[PATH]
    Bar --> STATEMENT1[STATEMENT]
    Bar --> TERMINATION[TERMINATION]
    GLOBAL --> ALLOCATE2[ALLOCATE]
    GLOBAL --> APPEARANCE2[APPEARANCE]
    GLOBAL --> CALL2[CALL]
    GLOBAL --> DATE2[DATE]
    GLOBAL --> DELETE2[DELETE]
    GLOBAL --> ENTRY2[ENTRY]
    GLOBAL --> EXIT2[EXIT]
    GLOBAL --> LABEL2[LABEL]
    GLOBAL --> LINE2[LINE]
    GLOBAL --> LOAD2[LOAD]
    GLOBAL --> PATH2[PATH]
    GLOBAL --> STATEMENT2[STATEMENT]
  
```

- A valid AT command that includes at least one operand. See Table 10 on page 207 for a list of valid AT commands. The AT command must be complete except that the *every* clause and *command* are omitted.

Restricts the list to enabled breakpoints. The default is to list both enabled and disabled breakpoints.

Restricts the list to disabled breakpoints. The default is to list both enabled and disabled breakpoints.

Lists currently defined AT `ALLOCATE` breakpoints.

Lists currently defined AT APPEARANCE breakpoints.

Lists currently defined AT CALL breakpoints.

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).

Lists currently defined AT DATE breakpoints.

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 information about global DATE breakpoint entries.

```
LIST AT DATE *;
```

- 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.

```
LIST AT LINE 22;
```

or

```
LIST AT OCCURRENCE SIGSEGV;
```

or

```
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. Under MVS batch and MVS with TSO, LIST CALLS lists the call chain of every active enclave in the process.

```
►►—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.
- This command also lists compile units in parent enclaves under CMS if the enclave was created using SVC LINK. If the enclave was created with the system() function or the CMSCALL macro, 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.

```
►►—LIST—CURSOR—————►►
```

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.

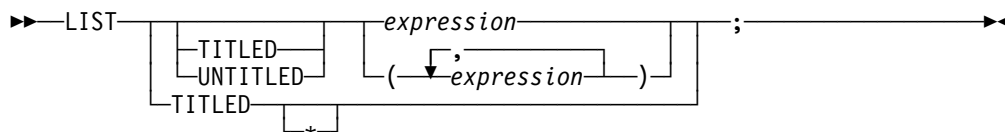
Example:

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/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/C++ and COBOL)

Lists all variables in the currently qualified compile unit.

UNTITLED

Lists expression values without displaying the expressions themselves. For C/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 with the exceptions described in “Expression” on page 201. 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 as in 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) as in example LIST TITLED GROUP y;.

Usage Notes:

- If LIST TITLED * is specified and your compile unit is large, slow performance might result.
- For COBOL, if LIST TITLED * is specified and your compile unit is large, you might receive an *out of storage* error message.
- 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 (i.e., 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);

LIST

- Display the COBOL references as if they were elementary items. The current programming language setting is COBOL.

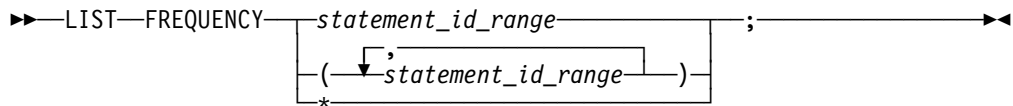
LIST (GROUP x OF z(1,2), GROUP a, w);

- Display the value of the Debug Tool variable %ADDRESS.

LIST %ADDRESS;

LIST FREQUENCY

Lists statement execution counts.



statement_id_range

A valid statement id or statement id range; see “Statement_Id_Range and Stmt_Id_Spec” on page 203.

- * 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.

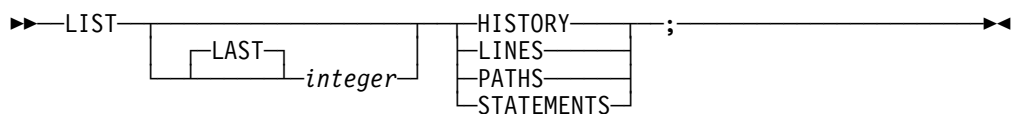
Note: See also “SET FREQUENCY” on page 312.

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 313.

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 284.

LIST LINES

See “**LIST STATEMENTS**” on page 285.

LIST MONITOR

Lists all or selected members of the current set of **MONITOR** commands.

►►—LIST—MONITOR—integer—- integer—;————►►

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.

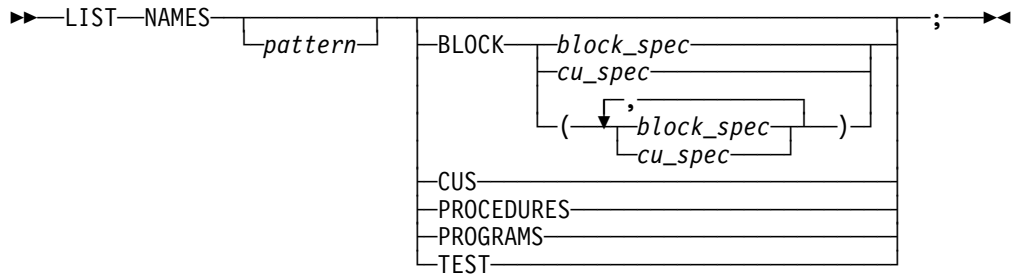
Example:

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"	'MY'
	'A5'	

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 199.

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 289.

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

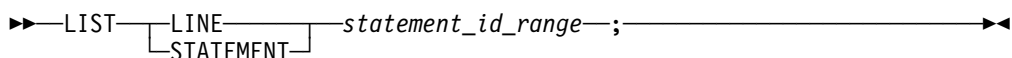
Displays the long-precision floating-point registers.

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 199. 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 203.

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 file. It is primarily intended for viewing portions of the source listing or source file in line mode, but can also be used in full-screen mode to copy a portion of a source listing or source file to the log.

►►—LIST—LINE
STATEMENT—*statement_id_range*—;—————◄◄

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 203.

Usage Notes:

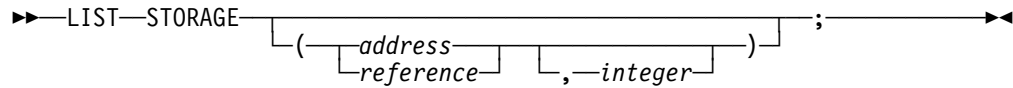
- The specified lines are displayed in the same format as they would appear in the full-screen source window, except that wide lines are truncated.
- You might need to specify a range of line numbers to ensure that continued statements are completely displayed.
- This command is not to be confused with 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.



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 202.

For C/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:

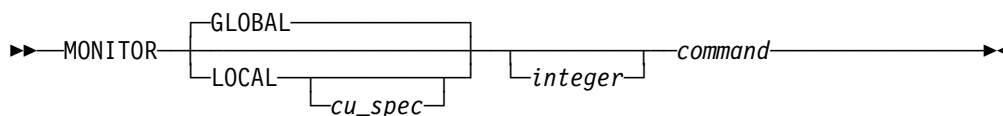
- Using Debug Tool, cursor pointing can be used by typing the LIST STORAGE command on the command line and moving the cursor to a variable in the source window before pressing ENTER, or by moving the cursor and pressing a PF key with the LIST STORAGE command assigned to it.
- When using the LIST STORAGE 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), terminal output (line 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, terminal output, 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 POPUP, TITLED, and UNTITLED options.

- When using the MONITOR LIST command, simple references (or C1 values) 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
literal TO reference—; —————►►

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.
- MOVE does not support date windowing. Therefore, you cannot use the MOVE command to assign the value of a windowed date field to an expanded date field or to a nondate field.
- You cannot use the MOVE command to assign the value of one expanded date field to another expanded date field with a different DATE FORMAT clause, or to assign the value of one windowed date field to another windowed date field with a different DATE FORMAT clause.

- Only the sender/receiver combinations listed in Appendix D, “Using COBOL Reference Information with Debug Tool” on page 350, 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;
- To assign a new value to a DBCS variable when the current programming language is COBOL, enter MOVE G"D B C S V A L U E" in the Command/Log window:

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.

►► ; ◀◀

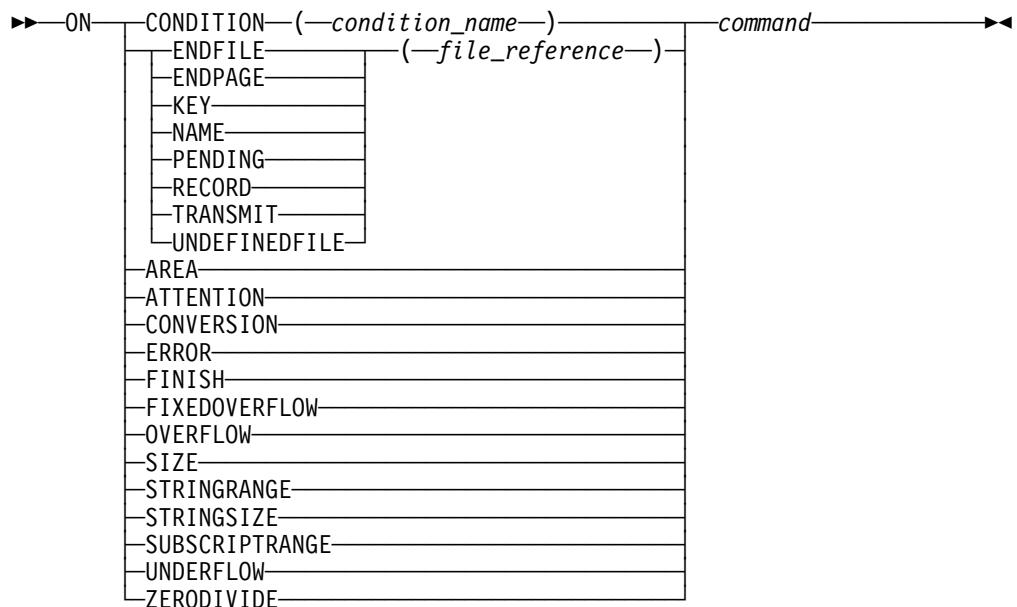
Example:

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 *PL/I 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:

```
ATTENTION or ATTN
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. For compatibility with PLITEST and INSPECT, however, it 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;

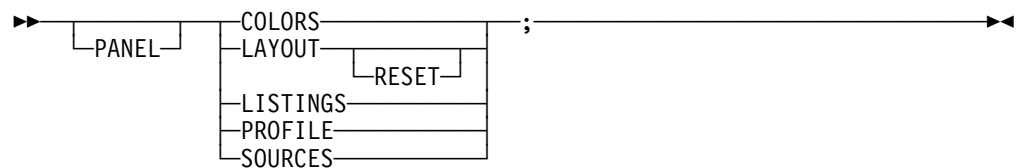
```

'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 22 on page 292 is an example of the panel.

Source Identification Panel		
Command ==>		
Compile Unit	Listings/Source File	Display
DBKP515	TS64081.TEST.LISTING(IBM73)	Y

Enter QUIT to return with current settings saved.		
CANCEL to return without current settings saved.		
UP/DOWN to scroll up and down.		

Figure 22. 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 LISTING *` (COBOL and PL/I), where `pgmname` is the name of your program. For TSO, the default file specification is `userid.pgmname.C` (C/C++), `userid.pgmname.list` (COBOL), or `userid.pgmname.list` (PL/I) for sequential data sets and `userid.dsname.C(membername)` (C/C++), `userid.dsname.Listing(membername)` (COBOL), or `userid.dsname.List(membername)` (PL/I) for partitioned data sets.

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:

- Compile the program with the proper option to generate a source or source listing file.
- Make sure the file is available and accessible on your host operating system.
- 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 the Debug Tool-defined file INSPSAFE. 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.

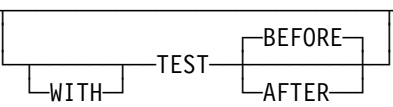

Simple:

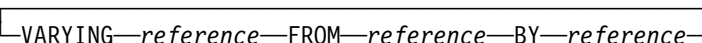

►►—PERFORM——END—PERFORM—;—◄◄



command

A valid Debug Tool command.

Repeating:

►►—PERFORM——◄◄

►——UNTIL—◄◄

►—*condition*——END—PERFORM—;—◄◄

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.
- Windowed date fields cannot be used as the VARYING *reference*, the FROM *reference*, or the BY *reference*.
- See *COBOL Language Reference publications* 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 16.

Table 16. 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 G0 or STEP command is issued within a procedure or a nested procedure, any statements following the G0 or STEP in that procedure and the containing procedure are ignored. If control returns to Debug Tool,

QUERY

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.

QUERY	CHANGE	
	COLORS	
	COUNTRY	
	DBCS	
	DEFAULT LISTINGS	
	DEFAULT SCROLL	
	DEFAULT WINDOW	
	ECHO	
	EQUATES	
	EXECUTE	
	FREQUENCY	
	HISTORY	
	INTERCEPT	
	KEYS	
	LOCATION	
	LOG	
	LOG NUMBERS	
	MONITOR NUMBERS	
	MSGID	
	LANGUAGE	
	NATIONAL	
	PACE	
	PFKEYS	
	PROGRAMMING LANGUAGE	
	PROMPT	
	QUALIFY	
	REFRESH	
	REWRITE	
	SCREEN	
	SCROLL DISPLAY	
	SETS	
	SOURCE	
	SUFFIX	
	TEST	
	WARNING	

CHANGE

Displays the current CHANGE setting.

COLORS (Full-Screen and Line Mode)

Displays the current COLOR setting.

COUNTRY

Displays the current COUNTRY setting.

DBCS

Displays the current DBCS setting.

DEFAULT LISTINGS (MVS)

Displays the current DEFAULT LISTINGS setting.

DEFAULT SCROLL (Full-Screen and Line Mode)

Displays the current DEFAULT SCROLL setting.

DEFAULT WINDOW (Full-Screen and Line 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 and Line 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 and Line Mode)**

Displays the current LOG NUMBERS setting.

MONITOR NUMBERS **(Full-Screen and Line Mode)**

Displays the current MONITOR NUMBERS setting.

MSGID

Displays the current MSGID setting.

NATIONAL LANGUAGE

Displays the current NATIONAL LANGUAGE setting.

PACE

Displays the current PACE setting. This setting is not supported in batch mode.

PFKEYS

Displays the current PFKEY definitions. This setting is not supported in batch mode.

PROGRAMMING LANGUAGE

Displays the current PROGRAMMING LANGUAGE setting. Debug Tool does not differentiate between C and C++, use this option for C++ as well as C programs.

PROMPT **(Full-Screen and Line Mode)**

Displays the current PROMPT setting.

QUALIFY

Displays the current QUALIFY BLOCK setting.

REFRESH **(Full-Screen and Line Mode)**

Displays the current REFRESH setting.

REWRITE

Displays the current REWRITE setting. This setting is not supported in batch mode.

SCREEN (Full-Screen and Line Mode)

Displays the current SCREEN setting.

SCROLL DISPLAY (Full-Screen and Line Mode)

Displays the current SCROLL DISPLAY setting.

SETS

Displays all current settings.

SOURCE

Displays the current SOURCE setting.

SUFFIX (Full-Screen and Line 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 if an expression is specified, sets the return code. It also invokes a prompt panel (full-screen) that asks if you really want to quit the debug session. In line and 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 201.

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 except where it appears in a command list. 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 files, primary commands files, 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 might be only the last line of the command.

```

▶▶ RETRIEVE [ COMMAND ] ;

```

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;

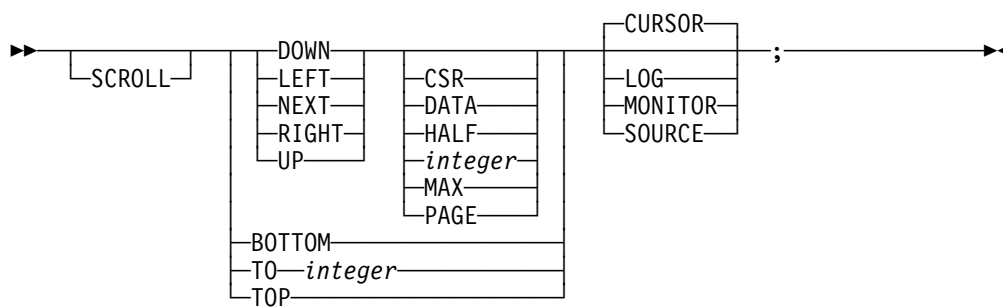
RUN command

The RUN command is synonymous to the G0 command. See “GO command” on page 267.

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).

SCROLL

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.

T0 *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 T0 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 309.

Examples:

- Scroll one page down in the window containing the cursor.


```
SCROLL DOWN PAGE CURSOR;
```

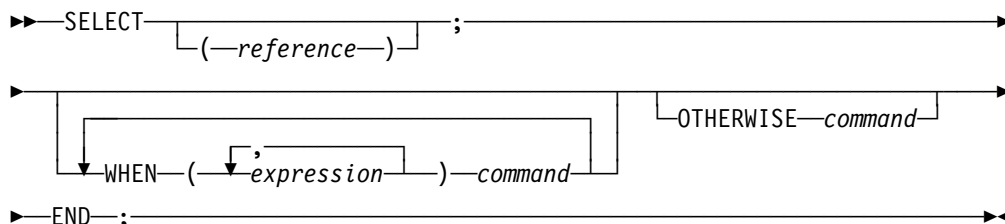
- Scroll the monitor window 12 columns to the left.

```
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 17.

Table 17 (Page 1 of 2). Summary of SET Commands

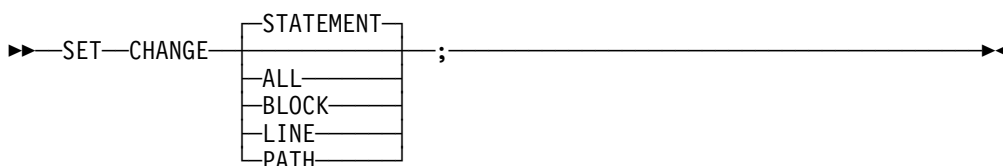
SET CHANGE	Controls the frequency of checking the AT CHANGE break-points.
SET COLOR	Provides control of the color, highlighting, and intensity attributes.
SET COUNTRY	Changes the current national country setting.
SET DBCS	Controls whether DBCS shift-in and shift-out codes are recognized.
SET DEFAULT LISTINGS	Defines a default partitioned data set (PDS) ddname or dsname 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/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.
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.

Table 17 (Page 2 of 2). Summary of SET Commands

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

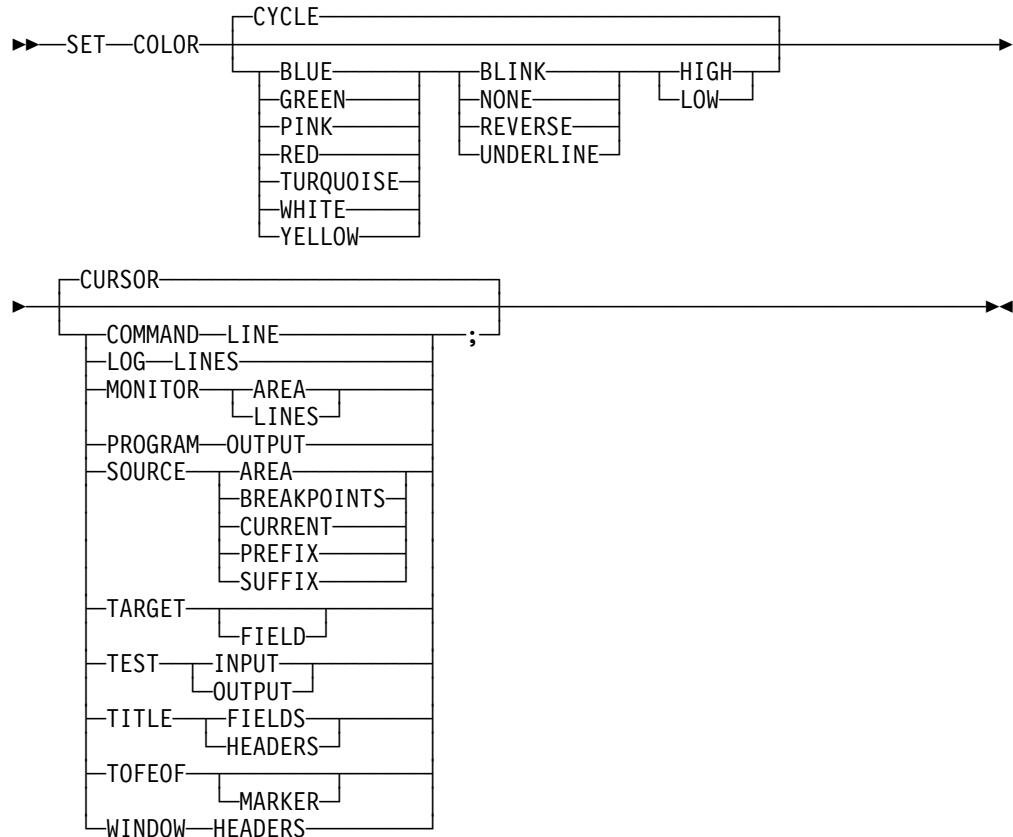
Specifies that the AT CHANGE breakpoints are checked at all path points.

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 and line 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.

TOEOF 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

```
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 Language Environment. 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 Language Environment 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. The initial setting is ON for C and PL/I and OFF for COBOL.

```
►►SET—DBCS—

|     |
|-----|
| ON  |
| 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;
```

SET DEFAULT LISTINGS (MVS)

Defines a default partitioned data set (PDS) ddname or dsname searched for program source listings or source files. The LISTINGS keyword cannot be abbreviated.

►►—SET—DEFAULT—LISTINGS—*listings_file*—;—————►►

listings_file

Specifies a ddname (a valid ddname in MVS) or a fully-qualified MVS data set name (for TSO and CICS) 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.
- For C/C++ compile units, Debug Tool requires a file containing the source code. By default, when Debug Tool encounters a new C/C++ compile unit, it looks for the source code in a file whose name is the one that was used on the compile step.

For VS COBOL II and PL/I compile units, Debug Tool requires a file containing the compiler listing. By default, when Debug Tool encounters a new VS COBOL II or PL/I compile unit, it looks for the listing in a file named *hlq.cuname.LIST*.

For COBOL/370, COBOL for MVS, and COBOL for OS/390, Debug Tool looks for the listing in a partitioned data set member named *cuname*.

Example:

Indicate that the default listings file is allocated to dsname SVTRSAMP.TS99992.MYPROG.

```
SET DEFAULT LISTINGS SVTRSAMP.TS99992.MYPROG;
```

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.

►►—SET—DEFAULT—SCROLL—

CSR
DATA
HALF
<i>integer</i>
MAX
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).

SET

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.

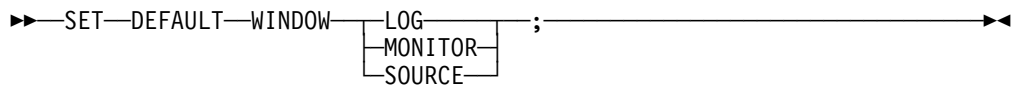
Example:

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.

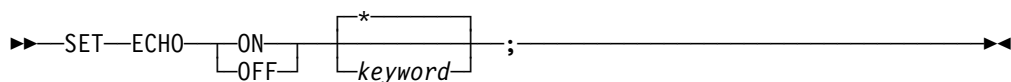
Example:

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.



ON Shows given commands in the log window.

OFF

Suppresses given commands in the log window.

keyword

Can be G0 (with no operand) or STEP.

- * Specifies that the command is applied to the G0 and STEP commands. This is the default.

Examples:

- Specify that the display of G0 and STEP commands is suppressed.

```
SET ECHO OFF;
```

- Specify that G0 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 (that is, CMS fileid, TSO dsname, and so forth) and are not scanned for EQUATED symbol substitution:

```
CMS
COMMENT
INPUT
SET DEFAULT LISTINGS
SET INTERCEPT ON/OFF FILE
SET LOG ON FILE
SET SOURCE (cu_spec)
SYSTEM/SYS
TSO
USE
```

SET

- 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.

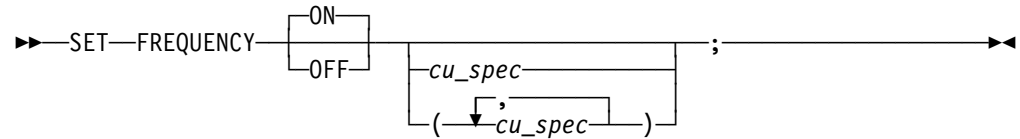
Example:

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 280.

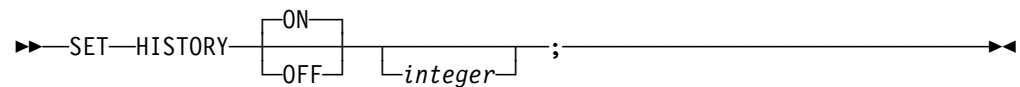
Example:

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 280.

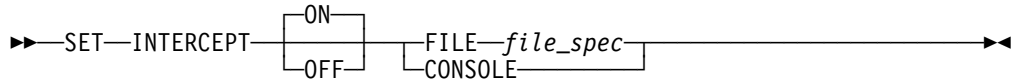
Examples:

- Adjust the history table size to 50 lines.
SET HISTORY 50;
- Turn off history recording.
SET HISTORY OFF;

SET INTERCEPT (C/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/C++ and COBOL)” on page 274.

OFF

Turns off I/O interception for the specified file.

FILE *file_spec*

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.

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:

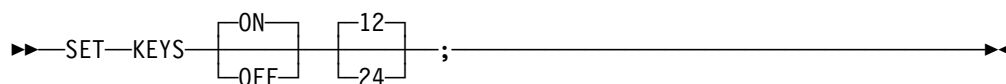
- COBOL supports only the CONSOLE command.
- For C, intercepted streams or files cannot be part of any C I/O redirection during the execution of a nested enclave.
- See also “SET REFRESH (full-screen mode)” on page 322 for debugging MVS/TSO applications that do line-input I/O.
- For PL/I, SET INTERCEPT 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 and line mode)

Controls whether PF key definitions are displayed when the SCREEN setting is ON.
The initial setting is ON.



0N 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 318.

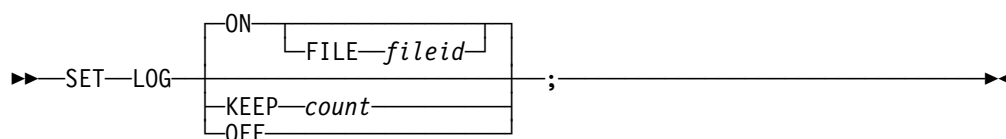
Example:

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 INSPLOG. This is a valid ddname in MVS or FILEDEF name in CMS.



ON Specifies that commands and output are written to the log file.

FILE fileid

Identifies the log file used. The FILE keyword cannot be abbreviated.

In TSO, *fileid* is a ddname or a qualified data set name. Partitioned data sets should *not* be used.

In CICS, fileid is a qualified data set name. Partitioned data sets should *not* be used.

In CMS, *fileid* is a FILEDEF name or a CMS fileid (*filename filetype filemode*). If *filemode* is omitted, the CMS search sequence is used.

If *fileid* has the form of a ddname, Debug Tool checks to see if the file is allocated (FILEDEFed in CMS).

KEEP count

Specifies the number of lines of log output retained for display. The initial setting is 1000; count cannot equal zero (0).

SET

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.
- For attributes of the log file, see “Creating the Log file” on page 88.

Examples:

- Specify that commands and output are written to the log file named mainprog.

```
SET LOG ON FILE mainprog;
```

Another example using the data set name thing.


```
SET LOG ON FILE userid.thing.log
```

- Indicate that 500 lines of log output are retained for display.

```
SET LOG KEEP 500;
```

SET LOG NUMBERS (full-screen and line mode)

Controls whether line numbers are shown in the log window. The initial setting is ON.

►► SET LOG NUMBERS  ; ◀◀

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 and line mode)

Controls whether line numbers are shown in the monitor window. The initial setting is ON.

►► SET MONITOR NUMBERS  ; ◀◀

ON Shows line numbers in the monitor window.

OFF

Suppresses line numbers in the monitor window.

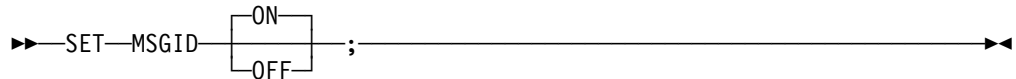
Example:

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.



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.

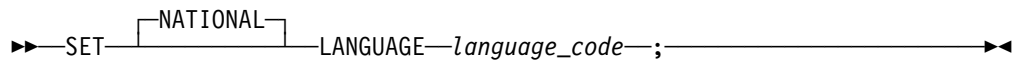
Example:

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 Language Environment, according to the setting in the current enclave.



language_code

A valid three-letter set that identifies the language used or (for compatibility) one of the two-letter language codes that was accepted in the previous release of INSPECT for C/370 and PL/I. The language code can have one of the following values:

United States English: ENU

United States English (Uppercase): UEN

Japanese: JPN

For compatibility with the previous release of INSPECT for C/370 and PL/I:

EN or ENGLISH is mapped to ENU

UE or UENGLISH is mapped to UEN

JA, JAPANESE, NI, or NIHONGO is mapped to JPN

SET

Usage Notes:

- This setting affects both your application and Debug Tool.
- At the beginning of an enclave, the settings are those provided by Language Environment or your operating system. For nested enclaves, the parent's settings are restored upon return from a child enclave.

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 PF*n* *string* = *command* ; ◀◀

PF*n*

A valid program function key specification (PF1 - PF24).

string

The label shown in the PF key display (if the KEYS setting is ON) that is entered as a string constant. The string 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:

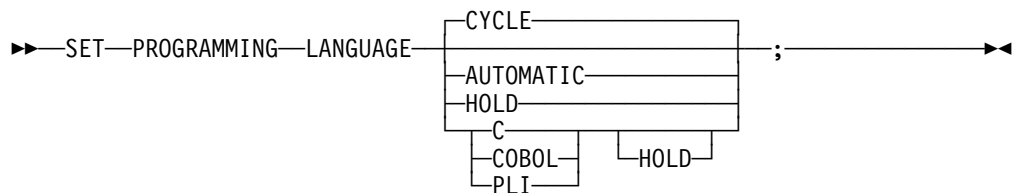
Define PF key 5 to scroll the cursor-selected screen forward. The current programming language setting is COBOL.

```
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 load module 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 load module 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.

SET

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. Debug Tool does not differentiate between C and C++, use this option for C++ as well as C programs.

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 (that is, HOLD is not 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/C++ is the current programming language.

```
SET PROGRAMMING LANGUAGE C;
```

SET PROMPT (full-screen and line mode)

Controls whether the current program location is automatically shown as part of the prompt message in line mode. It has no effect in full-screen mode, because the current location is always shown in the panel header in that case. The initial setting is LONG.

```
►►—SET—PROMPT—┐LONG┐;—————►►
                  └SHORT┘
```

LONG	Uses long form of prompt message.
SHORT	Uses short form of prompt message.

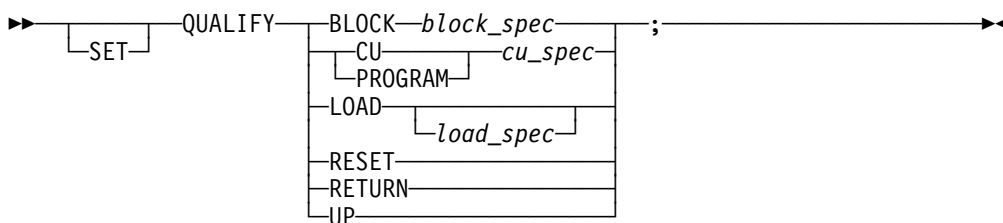
Example:

Specify that the long form of prompt message is used.

```
SET PROMPT LONG;
```

SET QUALIFY

Simplifies the identification of references and statement numbers by resetting the point of view to a new block, compile unit, or load module. 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 199.

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 load module.

load_spec
A valid load module specification; see “Load_Spec” on page 202. If omitted, the initial (primary) load module 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 load modules, 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 load module statmod should be used when no load module is specified.
SET QUALIFY LOAD statmod;
- Set the qualification back to the point of the suspended program.
SET QUALIFY RESET;
- Set the block qualification to blockx. As a result, the load module qualification and compile unit qualification will be updated to the load module 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.



ON Clears the screen before each rewrite. This is a required setting if your application handles line mode I/O.

OFF

Rewrites without clear.

Note: SET REFRESH ON is needed for applications that also make use of the screen , for example application using ISPF services to display panels.

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 EVERY number ;

```

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 and line mode)

Controls how information is displayed on the screen. The initial setting for a supported full-screen terminal is ON.

```

>> SET SCREEN 
  ON
  
    CYCLE
    integer
  
  
    LOG
    MONITOR
    SOURCE
  
  OFF
 ;

```

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.

OFF

Activates line mode. This mode is forced if the terminal is not a supported full-screen device.

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.

Examples:

- Indicate that the Debug Tool full-screen services are used.

```
SET SCREEN ON;
```

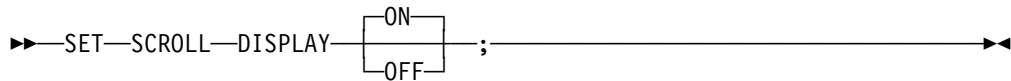
SET

- 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 95.

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.

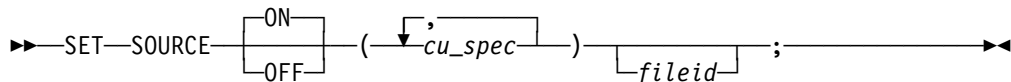
Example:

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.

fileid

Identifies the compile unit source file used. It is used in place of the default *fileid* for the compile unit. The file that you specify must be of fixed block format.

In MVS, *fileid* is a ddname, a qualified partitioned data set name (followed by a parenthesized member name if a PDS member is desired), or an HFS pathname.

In CICS, *fileid* is a fully-qualified data set name.

In CMS, *fileid* is a FILEDEF name or a CMS *fileid* (*filename filetype filemode*). If *filemode* is omitted, the CMS search sequence is used.

If *fileid* has the form of a ddname, Debug Tool checks to see if it is allocated (FILEDEFed in CMS). If not allocated, it is taken as a partitioned data set name or CMS fileid.

Fileid specifies a file identifier used in place of the default file identifier for the compile unit. A valid fileid is required unless it is already known to Debug Tool (via a previous SET SOURCE) or the default fileid 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 might not exist and the error (for non-existent 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 might 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 sequential data set userid.pgmname.LIST. If the Debug Tool window comes up empty, use the following command:

```
SET SOURCE ON (PGMNAME) userid.source.listings(cu_name) ;
```

This specifies the actual location of the listing file, in this example a partitioned data set with the program name differing from the CU name.

Examples:

- Indicate that the source associated with compile unit prog1 is found in ddname mainprog.

```
SET SOURCE ON (prog1) mainprog;
```
- Indicate that the source file associated with compile unit "/u/userid/code/oefun.c" is found in the HFS under the absolute pathname "/u/userid/code/oefun.c".

```
SET SOURCE ON ("/u/userid/code/oefun.c") /u/userid/code/oefun.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.

```
►► SET SUFFIX ON  
OFF ; ◄◄
```

ON Displays the suffix column.

OFF

Suppresses the suffix column.

Example:

SET

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.

```
►►—SET—TEST—test_level—;—————►►  
          └─(—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 Language Environment 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:

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

For more information about attention interrupts, see “Requesting an attention interrupt during interactive sessions” on page 135.

ERROR

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

- For C:
 - An attention interrupt
 - Program termination
 - A predefined Language Environment condition of Severity 2 or above
 - Any C condition other than SIGUSR1, SIGUSR2, SIGINT or SIGTERM.
- For COBOL:
 - An attention interrupt
 - Program termination
 - A predefined Language Environment condition of Severity 2 or above.
- For PL/I:
 - An attention interrupt, directed at either PL/I or Debug Tool
 - Program termination

value of one of the enumeration constants of the enumeration data type.

See “C/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 192 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 354 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.

►► SHOW *integer* ◄◄

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:

- User attention interrupt
- A breakpoint is encountered
- Normal or unusual termination of the program

►► STEP *integer* * INTO OVER RETURN ; ◄◄

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 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 load module where the source listing or source file is not displayed.

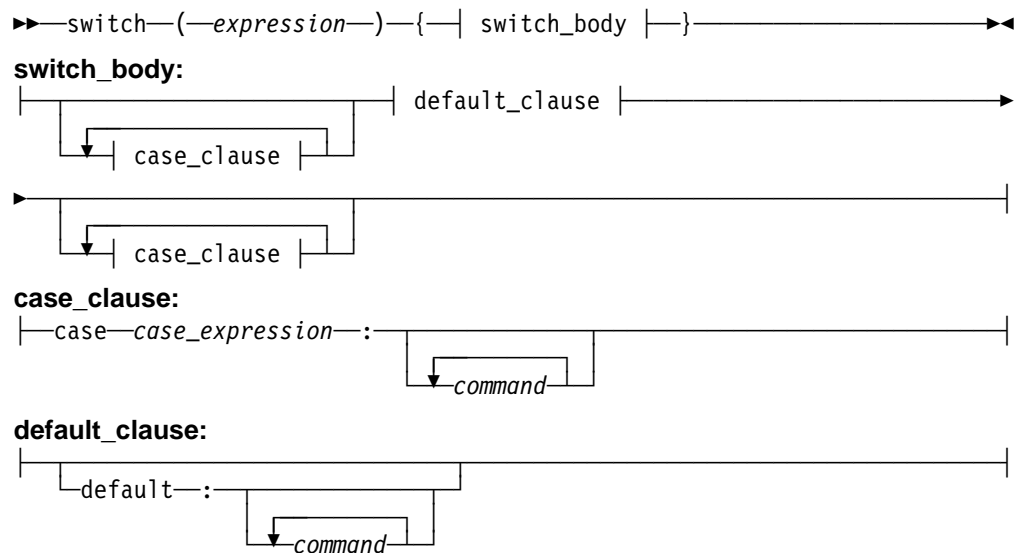
Examples:

- 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/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++;
    }
}
```

SYSTEM command

The SYSTEM command lets you issue system (CMS or TSO) commands during a Debug Tool session. The SYSTEM keyword can only be abbreviated as SYS.

```
►► [SYS] [SYSTEM] [system_command] ; ◄◄
```

system_command

A valid system command in the current operating system environment; however, the specified system command must be appropriate for the environment. For example, when operating in TSO, *system_command* can be a valid TSO system command or CLIST name.

Usage Notes:

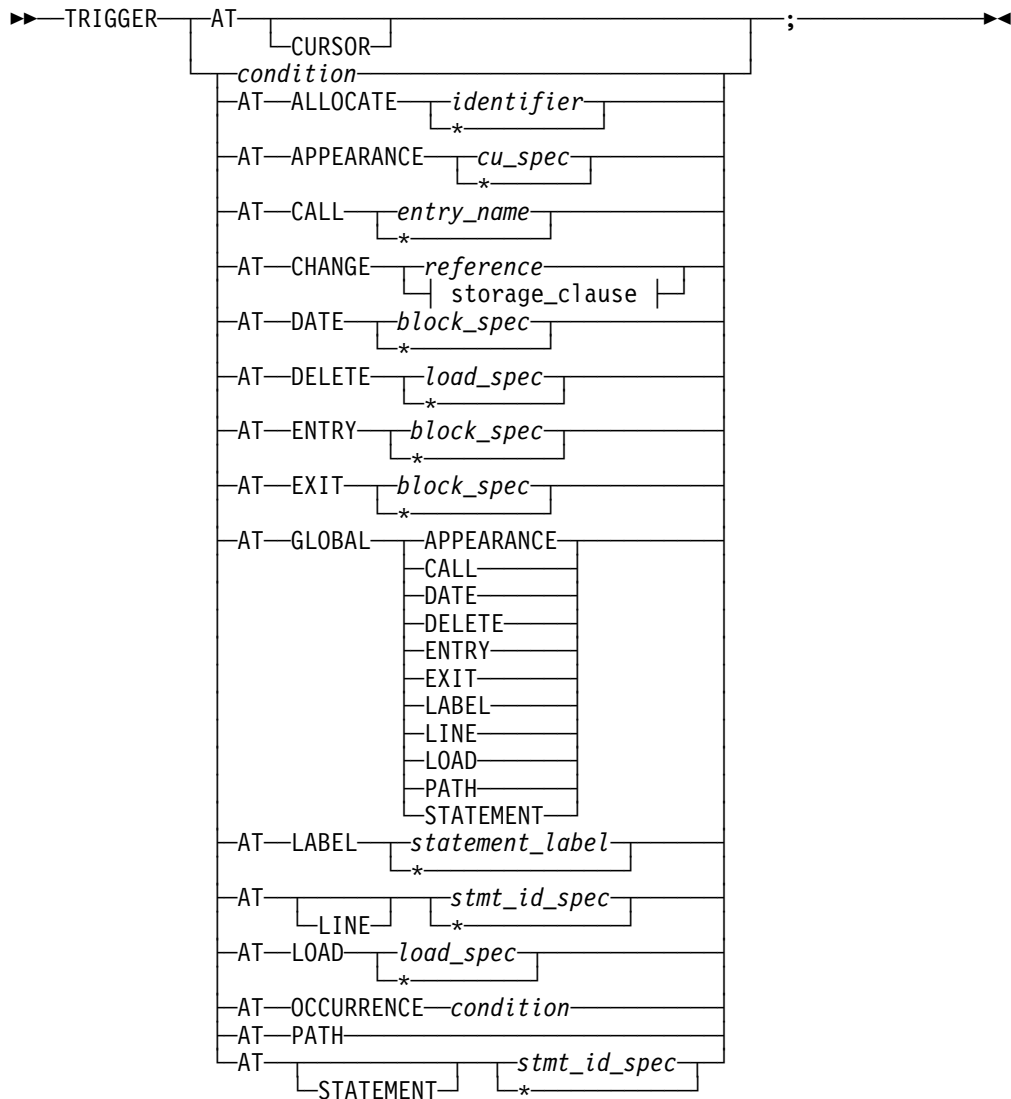
- You cannot introduce a new interactive debugging session with the SYSTEM command. For example, you cannot invoke a REXX program that would invoke a new Debug Tool instance, using SYSTEM REXXINVK.
- When operating interactively in CMS, if no CMS system command is specified, CMS subset mode is entered. While in CMS subset mode, a subset of CMS commands (that is, CMS system commands that can be issued while in the CMS editor) can be performed repeatedly. To return to Debug Tool, type RETURN.
- While in CMS subset mode, caution should be taken that your application program does not conflict with the memory or other resources of Debug Tool.
- When operating in TSO, a *system_command* must be supplied.
- When operating in TSO, no parameters can be specified as part of the system command or CLIST invocation. To execute noninteractively when parameters are required, you must enter the complete invocation in a CLIST and then use a TSO or SYSTEM command to invoke that CLIST (without parameters).
- You cannot introduce a new Debug Tool session using the SYSTEM command.
- When operating interactively in TSO, there is no provision for entering a mode where commands are accepted repeatedly; however, it is possible to write your own such iterative sequence in a CLIST.
- See also “CMS command (VM)” on page 244.
- See also “TSO command (MVS)” on page 336.
- You cannot issue CICS commands using SYSTEM.

Examples:

- List all the data sets in the user catalog. The operating system is TSO.
SYSTEM LISTCAT;
- List all the files that are named run on the a disk. The operating system is CMS.
SYSTEM LISTFILE run * a;

TRIGGER command

The TRIGGER command raises the specified AT-condition in Debug Tool, or it raises the specified programming language condition in your program.



storage_clause:

```

%STORAGE (address , length)
  
```

condition

A valid condition or exception. This can be either a Language Environment 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/C++ Reference Information with Debug Tool" on page 347 for a list of C conditions and their Language Environment equivalents.

SIGABND	SIGILL	SIGTERM
SIGABRT	SIGINT	SIGUSR1
SIGFPE	SIGIOERR	SIGUSR2
	SIGSEGV	

There are no COBOL condition constants. Instead, an Language Environment symbolic feedback code must be used, for example, CEE347. See *OS/390 Language Environment Programming Guide* for more details about language condition handling interactions.

PL/I condition constants can be used, for syntax and acceptable abbreviations see “ON command (PL/I)” on page 289.

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 202.

%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 load module specification; see “Load_Spec” on page 202.

block_spec

A valid block specification; see “Block_Spec” on page 199.

statement_label

A valid source label constant; see “Statement_Label” on page 204.

stmt_id_spec

A valid statement id specification; see “Statement_Id_Range and Stmt_Id_Spec” on page 203.

Usage Note:

- AT TERMINATION cannot be raised by TRIGGER.

Examples:

In the following examples, note the difference between triggering a break-point, which performs Debug Tool commands associated with the break-point, 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.

TSO command (MVS)

The TSO command lets you issue TSO commands during a Debug Tool session and is valid only in a TSO environment. The TSO keyword cannot be abbreviated.

►►—TSO—*tso_command*—;—————◄◄

tso_command

A valid TSO system command or CLIST name that does not require a parameter.

Usage Note:

- TSO is synonymous to SYSTEM. See “SYSTEM command” on page 333.

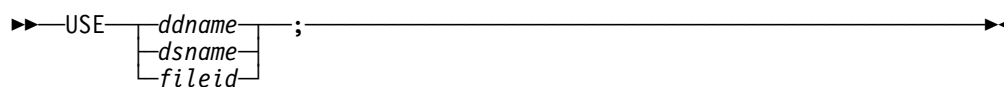
Example:

List all the data sets in the user catalog.

```
TSO LISTCAT;
```

USE command

The USE command causes the Debug Tool commands in the specified file or data set to be either performed or syntax checked. This file can be a log file from a previous session. The specified file or data set 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.

**ddname**

A valid ddname in MVS or FILEDEF name in CMS.

dsname

An MVS data set containing the Debug Tool commands to be performed.

fileid

A CMS fileid (*filename filetype filemode*) containing the Debug Tool commands to be performed. If *filemode* is omitted, the CMS search sequence is used.

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 G0) is performed from a USE file, the action taken (as far as closing the USE file) depends on certain things:
 - If the USE file was invoked directly or indirectly from the primary commands file or preferences file, it has the same characteristics as the primary commands file or preferences 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.
 - 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 G0, 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 G0, GOTO, or STEP causes any remaining commands in the USE file to be discarded.

Examples:

while

- On VM, perform the Debug Tool commands in the file pointed to by the ddname duse300 in the following filedef statement.
CMS filedef duse300 disk my300 exec o (recfm f lrecl 80 blksize 80;
USE duse300;
 - On VM, perform the Debug Tool commands in the file duse200 commands a.
USE duse200 commands a;
 - Perform the Debug Tool commands in the MVS data set mylog. The data set must first be allocated with, for example, ALLOC FI(MYLOG) DA("TS64081.USE.FILE").
USE MYLOG;
 - For CICS, perform Debug Tool commands in the fully-qualified data set TS64081.USE.FILE.
USE TS64081.USE.FILE;
- In addition to using sequential files, you can perform Debug Tool commands using partitioned data sets.
- USE userid.thing.file(usefile)

while command (C/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 18.

Table 18. Summary of WINDOW Commands

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.
WINDOW ZOOM	Expands the indicated window to fill the entire screen.

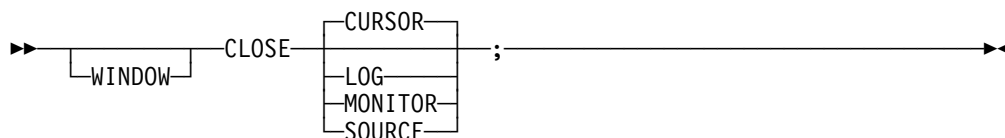
Usage Notes:

- 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.

WINDOW

MONITOR

Selects the monitor window.

SOURCE

Selects the source listing window.

Example:

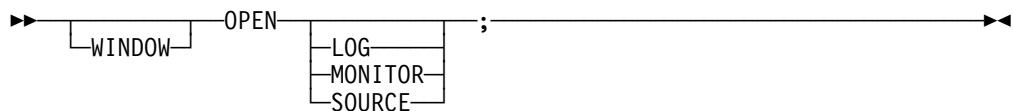
Close the window containing the cursor.

```
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.

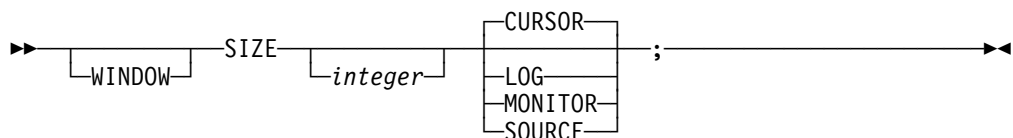
Example:

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:

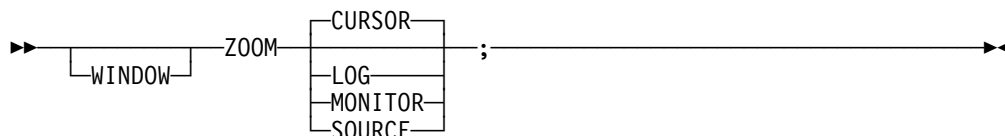
- You cannot use WINDOW SIZE if a window is ZOOMed or if there is only one window open.
- Each window in any configuration has only one adjustable dimension:
 - If one or more windows are as wide as the screen:
 - The number of rows is adjustable for each window as wide as the screen
 - The number of 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.

WINDOW

MONITOR

Selects the monitor window.

SOURCE

Selects the source listing window.

If the selected window is currently ZOOMed, the zoom mode is toggled. That is, the currently defined window configuration is restored.

Example:

Expand the log window.

```
WINDOW ZOOM LOG;
```

Appendix A. Coexistence

This appendix discusses Debug Tool's level of coexistence with other HLL debuggers, and the amount of debugging support you can expect for previous versions of debuggable languages.

Coexistence with other debuggers

Coexistence of Debug Tool with low-level debugging facilities (such as TSO TEST) is maintained. However, coexistence with other HLL debuggers cannot be guaranteed.

C/C++, COBOL, and PL/I are dependent upon Language Environment to provide debugging information.

Another debugger might provide limited services for an HLL not yet supported by Debug Tool, but conditions such as attention interrupts and exceptions cause Language Environment to pass control to an installed Language Environment debugger.

Coexistence with unsupported HLL modules

Compile units or program units written in unsupported high- or low-level languages, or in older releases of HLLs, are tolerated. See *Using CODE/370 with VS COBOL II and OS PL/I* for information about two unsupported HLLs which can be used with Debug Tool.

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. In such a case you can request an interrupt three times. After the third time, Debug Tool is able to stop program execution and prompt you to enter `QUIT` or `G0`. If you enter `QUIT`, your Debug Tool session ends. If you enter `G0`, control is returned to your application.

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 tradeoffs 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 `__ctest()`, `CEETEST`, or `PLITEST` is executed.

When Debug Tool receives control in this limited environment, it does not know what statement is in error (no statement table), nor can it locate variables (no symbol table). Thus, you must use addresses and interpret hexadecimal data values to examine variables. In this limited environment, you can:

- Determine the block that is in control:

```
list (%LOAD, %CU, %BLOCK);  
or  
list (%LOAD, %PROGRAM, %BLOCK);
```

- Determine the address of the error and of the 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/C++ program by entering:

```
LIST STORAGE (0x20058);
```

To display the contents at address 20058 in a COBOL or PL/I program, you would enter:

```
LIST STORAGE (X'20058');
```

- Display registers:

```
LIST REGISTERS;
```

- Display program characteristics:

```
DESCRIBE CU; (for C)
```

```
DESCRIBE PROGRAM; (for COBOL)
```

- Display the dynamic block chain:

```
LIST CALLS;
```

- Request assistance from your operating system:

```
SYSTEM ...;
```

- Continue your program processing:

```
GO;
```

- End your program processing:

```
QUIT;
```

If your program does not contain a statement or symbol table, you can use 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 var-

table 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/C++ Reference Information with Debug Tool

This appendix contains reference information for use when debugging C/C++ programs with Debug Tool.

All references in this section refer to C and C++ unless otherwise stipulated.

Table 19 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/C++.

Table 19. Debug Tool Interpretive Subset of C/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 20 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 20. 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 21 on page 348 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 21 on page 348 lists the C operators and their orders of precedence.

Table 21. 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	,

Language Environment conditions and their C/C++ equivalents

Language Environment condition names (the symbolic feedback codes CEExxx) can be used interchangeably with the equivalent C/C++ conditions listed in Table 22. 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 22 (Page 1 of 2). Language Environment Conditions and Their C/C++ Equivalents

Language Environment Condition	Description	Equivalent C/C++ Condition
CEE341	Operation exception	SIGILL
CEE342	Privileged operation exception	SIGILL
CEE343	Execute exception	SIGILL
CEE344	Protection exception	SIGSEGV
CEE345	Addressing exception	SIGSEGV
CEE346	Specification exception	SIGILL
CEE347	Data exception	SIGFPE
CEE348	Fixed point overflow exception	SIGFPE
CEE349	Fixed point divide exception	SIGFPE
CEE34A	Decimal overflow exception	SIGFPE
CEE34B	Decimal divide exception	SIGFPE
CEE34C	Exponent overflow exception	SIGFPE
CEE34D	Exponent underflow exception	SIGFPE

Table 22 (Page 2 of 2). Language Environment Conditions and Their C/C++ Equivalents

Language Environment Condition	Description	Equivalent C/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.

COBOL listing files

Debug Tool does not display a COBOL listing file associated with any compile unit (CU) unless the file has a fixed blocked format.

Debug Tool interpretive subset of COBOL commands

Table 23 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 23. 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 24 on page 351 shows the allowable comparisons for the Debug Tool IF command. A description of the codes follows the table.

Table 24. 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) ⁸	NN		NN												
EXTERNAL DECIMAL (ED) ⁸	NN			NU											
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) ⁸	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:

- FIGCON includes all figurative constants except ZERO and ALL.
- A nonnumeric literal must be enclosed in quotation marks, and the quotation marks are not valid characters in the literal.
- Must contain only alphabetic characters.
- Index name converted to subscript value before compare.
- Only comparison for equal and not equal can be made.
- Must be hexadecimal characters only, delimited by either double (") or single (') quotation marks and preceded by H.
- Constants and literals can also be compared against constants and literals of the same type.
- Comparisons using windowed date fields are not supported.

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 25 shows the allowable moves for the Debug Tool MOVE command.

<i>Table 25 (Page 1 of 2). Allowable Moves for the Debug Tool MOVE Command</i>											
Receiving Field	GR	AL	AN	ED	BI	NE	ANE	ID	IF	EF	D1
Source Field											
GROUP (GR)	Y	Y	Y	Y1	Y1	Y1	Y1	Y1	Y1	Y1	
ALPHABETIC (AL)	Y	Y									
ALPHANUMERIC (AN) ^{4,5}	Y		Y								
EXTERNAL DECIMAL (ED) ^{4,5}	Y1			Y							
BINARY (BI)	Y1				Y						
NUMERIC EDITED (NE)	Y										
ALPHANUMERIC EDITED (ANE)	Y						Y				
FIGCON ZERO	Y		Y	Y2	Y2		Y	Y2	Y	Y	
SPACES (AL)	Y	Y	Y				Y				
HIGH-VALUE, LOW-VALUE, QUOTES	Y		Y				Y				
NUMERIC LITERAL	Y1			Y	Y			Y	Y	Y	
NONNUMERIC LITERAL	Y	Y	Y			Y1	Y				
INTERNAL DECIMAL (ID) ^{4,5}	Y1							Y			
FLOATING POINT LITERAL	Y1								Y	Y	
INTERNAL FLOATING POINT (IF)	Y1								Y	Y	
EXTERNAL FLOATING POINT (EF)	Y1								Y	Y3	
DBCS DATA ITEM (D1)											Y

Table 25 (Page 2 of 2). Allowable Moves for the Debug Tool MOVE Command

Receiving Field											
Source Field	GR	AL	AN	ED	BI	NE	ANE	ID	IF	EF	D1
DBCS LITERAL											Y
Notes: <ol style="list-style-type: none"> 1 Move without conversion (like AN to AN) 2 Numeric move 3 Decimal-aligned and truncated, if necessary 4 MOVE does not support date windowing. For example, the MOVE statement cannot be used to move a windowed date field to an expanded date field, or to a nondate field. 5 The MOVE command cannot be used to move one windowed date field to another windowed date field with a different DATE FORMAT clause, or to move one expanded date field to another expanded date field with a different DATE FORMAT clause. 											

For more information, see the “MOVE command (COBOL)” on page 288.

Allowable moves for the Debug Tool SET command

Table 26 shows the allowable moves for the Debug Tool SET command.

<i>Table 26. Allowable Moves for the Debug Tool SET Command</i>							
Receiving Field	IN	IDI	PTR	ED	BI	ID	OR
Source Field							
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 <i>H</i> . 2 Index name is converted to index value.							

Appendix E. Debug Tool Messages

All messages 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.

Symbols in messages

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 italicized symbols, as in the following:

EQA1046I The *breakpoint-id* breakpoint is replaced.

The portion of Debug Tool located on the host notifies you of errors associated with debugging functions carried out by the host.

EQA0320E Host server not active

Explanation: The host server specified in the debug options has not been started. Debug Tool cannot initialize.

EQA0321E Host server not available

Explanation: Debug Tool cannot establish communications with the host server specified in the debug options. Debug Tool cannot initialize.

EQA0322E Invalid host server name

Explanation: The host server specified in the debug options cannot be found. Debug Tool cannot initialize.

EQA0323I Host server busy. Action will complete when server is available.

Explanation: The host server is busy processing a request from CODE. Debug Tool cannot proceed until the previous request completed.

EQA0324E Fatal communications error. Debug Tool cannot continue.

Explanation: Debug Tool cannot send/receive messages from the host server; Debug Tool cannot continue and will terminate abnormally. Diagnostic information is included in the EVFERROR.LOG file, in the path specified by the CODETMPDIR variable in your CONFIG.SYS file.

EQA1000I TEST (*cu_name* initialization):

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs after Debug Tool initialization and before any program hooks are reached.

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.

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.

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 might 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 might be compiled without the TEST option.

EQA1116I Common Language Environment math library is being used

Explanation: This is the response for the DESCRIBE ENVIRONMENT command when the Language Environment 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 *Load Module name*:

Explanation: This title line precedes a list of compile unit names for noninitial load modules in a LIST NAMES CUS command.

EQA1156I The following CUs with pattern *pattern* are known in *Load Module name*

Explanation: This title line precedes a list of compile unit names for noninitial load modules that satisfy the pattern in a LIST NAMES CUS command.

EQA1157I There are no CUs with pattern *pattern* in *Load Module name*.

Explanation: This line appears when no compile unit satisfied the pattern in a LIST NAMES CUS command for noninitial load modules.

EQA1158I The following CUs have pattern *pattern*

Explanation: This title line precedes a list of compile unit names for an initial load module 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 load module.

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 *ddname*

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.

EQA1188I (*history number*) DATE hook at statement *cu_name* :> *statement_id*

Explanation: This is a LIST HISTORY message.

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* :> *state-
ment 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 PFKEYS.

EQA1247I colored area *color highlight 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 name 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.

EQA1279I TEST (*cu_name* **>** *statement_id* **):**

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a statement and a statement table is available.

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.

EQA1280I TEST (*cu_name* **ENTRY):**

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a compile unit entry.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information will be displayed identifying the commands you are allowed to enter.

EQA1281I TEST (*cu_name* **>** *block name* **ENTRY):**

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a block entry.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information will be displayed identifying the commands you are allowed to enter.

EQA1282I TEST (*cu_name* **EXIT):**

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a compile unit exit.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information will be displayed identifying the commands you are allowed to enter.

EQA1283I TEST (*cu_name* **>** *block name* **EXIT):**

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a block exit.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information will be displayed identifying the commands you are allowed to enter.

EQA1284I TEST (Application program has terminated):

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at the termination of the application program.

Programmer Response: Enter a command. If you are not sure what you can enter, enter HELP or ?. Information will be displayed identifying the commands you are allowed to enter.

EQA1285I TEST (*label-name* LABEL);

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in line mode when an initial prompt occurs at a label.

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.

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 *ddname*: program output

Explanation: Displays program output with the *ddname* preceding the output.

EQA1290I The program is waiting for input from *ddname*

Explanation: Debug Tool has gained control because the program is waiting for input.

EQA1291I Use the INPUT command to enter *resize* 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 *resize* characters for the intercepted variable-format file.

Explanation: Prompt user for intercepted input of variable-formatted file.

EQA1293I TEST (*cu-name*):

Explanation: Debug Tool is ready to accept a command from the terminal. This message is used in linemode when an initial prompt occurs at a statement and a statement table is not available.

Programmer Response: Enter a command. If you are not sure of what you can enter, enter HELP or ?. Information is displayed identifying the available commands you are allowed to enter.

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 *OS/390 Language Environment 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. The line on which the string was found is displayed just above this message when operating in line mode.

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 IBM Debug Tool Version 1 Release 2 time stamp 5688-194 (C) Copyright IBM Corp. 1995

Explanation: This message is used to place the Debug Tool logo, a timestamp, and copyright at the beginning of the log. This is for Language Environment.

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: There 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 load module.

Explanation: A SET PROGRAMMING LANGUAGE command was issued, but the initial load module 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, or incompatible or unsupported date fields. If you are using COBOL, see Table 24 on page 351 for allowable comparisons for the Debug Tool IF command, and Table 25 on page 352 for allowable moves for the Debug Tool MOVE command.

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. Examples of when this message is issued when using COBOL include:

- An attempt to perform arithmetic with a nonnumeric data item
- An attempt to perform arithmetic with a windowed date field

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 for items in the LINKAGE SECTION that are not based on a valid address.

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 nonnumeric 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 might 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 might 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 might have been deleted or canceled.

EQA1466E *data item* is not a valid subscript or index.

Explanation: The subscript or index might be out of range or an ODO variable might 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 might 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 region 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 3 and the compile-time TEST or FDUMP option, or AD/Cycle COBOL and the compile-time TEST option.

EQA1486E *variable name* is presently not in accessible storage.

Explanation: The variable might 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 struc-

ture, 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 "\neq" 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 might be uninitialized or might 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 might be uninitialized or it might 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 can 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 = +1i or z = -1i.*

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	$ABS(X) \leq (2^{**18}) * \pi$
TAN	$ABS(X) \leq (2^{**18}) * \pi$ if x is real and $ABS(REAL(X)) \leq (2^{**17}) * \pi$ if x is complex
TANH	$ABS(IMAG(X)) \leq (2^{**17}) * \pi$ if x is complex
COSH, EXP and SINH	$ABS(IMAG(X)) \leq (2^{**18}) * \pi$ if x is complex
COSD, SIND and TAND	$ABS(X) \leq (2^{**18}) * 180$

For long floating-point arguments, the limits are:

COS and SIN	$ABS(X) \leq (2^{**50}) * \pi$
TAN	$ABS(X) \leq (2^{**50}) * \pi$ if x is real and $ABS(REAL(X)) \leq (2^{**49}) * \pi$ if x is complex
TANH	$ABS(IMAG(X)) \leq (2^{**49}) * \pi$ if x is complex
COSH, EXP and SINH	$ABS(IMAG(X)) \leq (2^{**50}) * \pi$ if x is complex
COSD, SIND and TAND	$ABS(X) \leq (2^{**50}) * 180$
For extended floating-point arguments, the limits are:	
COS and SIN	$ABS(X) \leq (2^{**106}) * \pi$
TAN	$ABS(X) \leq (2^{**106}) * \pi$ if x is real and $ABS(REAL(X)) \leq (2^{**105}) * \pi$ if x is complex
TANH	$ABS(IMAG(X)) \leq (2^{**105}) * \pi$ if x is complex
COSH, EXP and SINH	$ABS(IMAG(X)) \leq (2^{**106}) * \pi$ if x is complex
COSD, SIND and TAND	$ABS(X) \leq (2^{**106}) * 180$

EQA1631E *The subject of the SUBSTR pseudovisible (character string) is not a string.*

Explanation: You are trying to get a substring from something other than a string.

EQA1632E *Argument to pseudovisible must be complex numeric*

Explanation: This message applies to PL/I constants.

EQA1633E *The first argument to a pseudovisible must refer to a variable, not an expression or a pseudovisible.*

Explanation: The arguments that accompany a pseudovisible 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 dDCL 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 built-in function chapter for the description of when STRINGRANGE is raised. See the Language Reference condition chapter for the values of the revised SUBSTR reference.

EQA1645E The subject of the *pseudovalue name* pseudovalue must have data type: *data type*.

Explanation: This message applies to various pseudovariables. By pseudovalue, 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.

EQA1683E Since the **cu** *cu_name* does not have hooks at statements with modified behavior due to the Millennium Language Extensions, all **AT DATE** commands will be rejected for the **cu**.

Explanation: A compile unit must have been compiled with the **DATEPROC** option and either **TEST(STMT)** or **TEST(ALL)** for hooks to be present at statements affected by the Millennium Language Extensions.

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 Language Environment **CEE3DMP** 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 is 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 and C identifiers in 255 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'. For ASCII, the leading byte is X'82' and the trailing byte is in the range X'40' to X'7E'.

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. This message is for the System/370 environment.

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. This message is for the System/370 environment.

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. This message is for the System/370 environment.

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 7.23700557733226221397318656304298E+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 5.3976053469340278908664699142502496E-79 and less than 7.2370055773322622139731865630429929E+75.

Explanation: This is the range of values allowed by C.

EQA1872E An error occurred while opening file: *file name*.

Explanation: An error during the initial processing (OPEN) of the file occurred.

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 line mode or in a 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: Under MVS, data definition names are limited to 8 characters and data set names are limited to 44 characters. If a partitioned data set is named, the member name must be specified (with up to 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 GOS/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.

EQA1903E An attention request has been issued. Enter QUIT to terminate Debug Tool or GO or RUN to resume execution.

Explanation: The attention key was pressed three times because the application was looping either in system code or application code without debugging hooks. Only the GO/RUN and QUIT commands are valid at this point.

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 load module 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 load module 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 load module named *load module name*.

Explanation: Load module qualification is referring to a load module that cannot be located.

Programmer Response: The load module might be missing or it might have been loaded before Debug Tool was first used. On the System/370, Debug Tool is aware of additional load modules **only** if they were FETCHed after Debug Tool got control for the first time.

EQA1926S There is no cu named *cu_name* within load module *load module name*.

Explanation: The compilation unit might be misspelled or missing.

EQA1927S There are *number* CUs named *cu_name*, but neither belongs to the current load module.

Explanation: The compilation unit you named is not unique.

Programmer Response: Add further qualification so that the correct load module 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 load module 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 load module *load module 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 load module 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.

EQA1958E **The requested SYSTEM command could not be run.**

Explanation: A SYSTEM command was issued. The underlying operating system received it but did not process it successfully.

EQA1959E **The requested SYSTEM command was not recognized.**

Explanation: The underlying operating system was passed a command that was not recognized. The system could not process the command.

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* pseudovisible (*character string*) must be complex numeric.

Explanation: You are trying to get apply the PL/I IMAG or REAL pseudovisible 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 GOTOs 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. Upper-case 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 cannot 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.

EQA1980E Invalid *symbolic_destination_name* - *symbolic_destination_name*.

Explanation: Conversation initialization failed due to an invalid *symbolic_destination_name* in the Session Parameter. The *symbolic_destination_name* was either not found in the APPC/MVS side information file, or it is longer than 8 characters.

Programmer Response: If the length of the *symbolic_destination_name* is valid, contact your APPC/MVS system administrator to verify its existence in the side information file. For a description of the Session Parameter and its contents, see the CODE/370 Debug Tool manual.

**EQA1981E Invalid mode name, transaction program name, or partner LU name associated with *symbolic_destination_name*.
Mode_name= *mode_name* and
partner_LU_name= *partner_LU_name***

Explanation: A conversation allocation request failed due to invalid conversation characteristics obtained from the APPC/MVS side information file. There could be several reasons for this:

1. The *mode_name* characteristic specifies a mode name that is either not recognized by the LU as valid or is reserved for SNA service transaction programs.
2. The *TP_name* characteristic specifies an SNA service transaction program name.
3. The *partner_LU_name* characteristic specifies a partner LU name that is not recognized by the LU as being valid.

Programmer Response: Contact your APPC/MVS system administrator to modify the characteristics associated with the given *symbolic_destination_name* in the side information file. For information about the recommended values for *mode_name* and *TP_name*, see the CODE/370 Installation manual.

The OS/2 system error log can contain valuable diagnostic information. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

**EQA1982E Permanent conversation allocation failure for *symbolic_destination_name*.
Partner_LU_name= *partner_LU_name* and
mode_name= *mode_name***

Explanation: The conversation cannot be allocated because of a condition that is not temporary. There could be several reasons for this:

1. The workstation where the *partner_LU_name* is defined is turned off or Communications Manager/2 is not started.
2. The *partner_LU_name* has not been defined.
3. The current session limit for the specified *partner_LU_name* and *mode_name* pair is zero.
4. A system definition error or a session-activation protocol error has occurred.

Programmer Response: Ensure that you specified the correct *symbolic_destination_name* or contact your APPC/MVS system administrator to correct the condition.

The OS/2 system error log can contain valuable diagnostic information. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

**EQA1983E Temporary conversation allocation failure for *symbolic_destination_name*.
Partner_LU_name= *partner_LU_name* and
mode_name= *mode_name*.**

Explanation: The conversation cannot be allocated because of a condition that might be temporary. There could be several reasons for this:

1. Undefined *mode_name* (not temporary)
2. Temporary lack of resources at the host LU or workstation LU

Programmer Response: Verify that *mode_name* is defined on the target workstation using the CM/2 Communication Manager Setup panels. If *mode_name* is defined on the workstation, contact your MVS/ESA system programmer to ensure that *mode_name* is also defined on the MVS system.

The OS/2 system error log can contain valuable diagnostic information. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

**EQA1984E The workstation transaction program is permanently unavailable at *symbolic_destination_name*.
Partner_LU_name= *partner_LU_name*.**

Explanation: *Partner_LU_name* rejected the allocation request because the host program specified a workstation program that *partner_LU_name* recognizes but it cannot start. There could be several reasons for this:

1. Missing transaction program definition on the workstation.
2. Invalid OS/2 program path and file name specified in the transaction program definition.

Programmer Response: Define the transaction program on the workstation or ensure that the transaction program definition is correct. The *symbolic_destination_name* can be used to obtain the workstation transaction program name from the APPC/MVS side information table. For information about the recommended values for *TP_name*, see the CODE/370 Installation manual.

The OS/2 system error log can contain valuable diagnostic information. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

EQA1985E Unrecognized transaction program name
 at *symbolic_destination_name*.
Partner_LU_name= *partner_LU_name*.

Explanation: *Partner_LU_name* rejected the allocation request because the host program specified a workstation *TP_name* that *partner_LU_name* does not recognize. The transaction program definition is missing on the workstation.

Programmer Response: Define the transaction program on the workstation. The *symbolic_destination_name* can be used to obtain the workstation transaction program name from the APPC/MVS side information table. For information about the recommended values for *TP_name*, see the CODE/370 Installation manual.

The OS/2 system error log can contain valuable diagnostic information. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

EQA1986E Unexpected LU 6.2 error. Module=
module_name, **Location=** *location_id*,
CPI-C call= *call_type*, **return_code=** *rc*.

Explanation: The host communications code received an unexpected return code from a CPI-C call. The information displayed is for diagnostic purposes.

- *module_name* is the name of the communications module issuing the CPI-C call
- *location_id* is an internal 3 digit identifier for the CPI-C call within the module
- *call_type* is the CPI-C call type (for example, CMINIT or CMALL)
- *rc* is the unexpected *return_code* which is displayed in decimal.

Programmer Response: Forward a copy of this message to your APPC/MVS system administrator. Diagnostic information was recorded in either the EVFERROR.LOG or the EQALU62.LOG. The path where these logs are stored is in the CODETMPDIR environment variable in CONFIG.SYS.

The OS/2 system error log can contain valuable diagnostic information for your IBM service representative. To access the system error log, select **System Error Log** from the FFST/2 folder or type SYSLOG at the OS/2 command line.

EQA1987E Debugger terminated, execution continues.

Explanation: The initialization of the LU 6.2 conversation between the host and the workstation (in a batch process) has failed. The debugger is terminated and the execution of the batch application continues. Note the accompanying messages as to possible causes.

EQA1988E Severe internal error. PWS Debug Tool terminated.

Explanation: PWS Debug Tool detected a severe internal error. It has been shutdown.

Programmer Response: Diagnostic information was recorded in either the EVFERROR.LOG or the EQALU62.LOG. The path where these logs are stored is in the CODETMPDIR environment variable in CONFIG.SYS.

EQA1989E Invalid session ID - *session_ID*

Explanation: Conversation initialization failed due to an invalid session ID in the Session Parameter. There could be several reasons for this,

1. The session ID is longer than 8 characters or contains invalid characters. Valid session IDs consist of 1-8 alphanumeric characters.
2. There is already another PWS Debug Tool session with the given session ID.

Programmer Response: Diagnostic information is recorded in either the EVFERROR.LOG or the EQALU62.LOG. The path where these logs are stored is in the CODETMPDIR environment variable in CONFIG.SYS.

If there is already an existing PWS Debug Tool session with the given session ID then a different session ID must be provided for concurrent debug sessions on the same workstation. If a session ID is not specified, it defaults to CODEDT. For a description of the Session Parameter and its contents, see the CODE/370 Debug Tool manual.

EQA1990E Invalid session parameter - *session_parameter*

Explanation: Conversation initialization failed. A batch program, attempting to start an LU 6.2 debug session, has passed an invalid Session Parameter. For example, LU2 or MFI has been specified for session type or a session ID longer than eight characters has been specified. For a description of the Session Parameter and its contents, see the CODE/370 Debug Tool manual.

Programmer Response: Correct the Session Parameter and invoke the batch application again.

EQA2001E Ambiguous conversion between "&1" and "&2".

Problem Determination: (*where* &1 is a C/C++ type &2 is a C/C++ type)

Explanation: The debugger was not able to find a single type common to the two specified types and was therefore unable to convert from one to the other.

Programmer Response: Explicitly cast the type to an intermediate type and then convert to requested type.

EQA2002E A return value is not allowed for this function.

Explanation: A function with a return type of "void" cannot return a value.

Programmer Response: Remove the value or expression from the return statement, remove the return statement, or change the return type of the function.

EQA2003E Identifier "&1" is undefined.

Problem Determination: (*where* &1 is a C/C++ name)

Explanation: The specified identifier is used but has not been defined.

Programmer Response: Define the identifier before using it. Check its spelling. If the identifier has been defined in a header file, check that any required macros have been defined.

EQA2004E &1 member "&2" cannot be accessed.

Problem Determination: (*where* &1 is the keyword "private" or "protected" &2 is a class member name)

Explanation: The specified member is private, protected, or is a member of a private base class and cannot be accessed from the current scope.

Programmer Response: Check the access specification rules for the member function and change the access specifier if necessary. If the member function belongs to a base class, check the access specifier of the base class where the current class is defined.

EQA2005E Return value of type "&1" is expected.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: No value is returned from the current function, but the function is expecting a nonvoid return value. The function was declared with a return type but the debugger did not detect a return statement. Only functions with a void return type can have no return statement or have a return statement with no return value.

Programmer Response: Return a value from the function or change the function's return type to void.

EQA2006E "&1" cannot be made a &2 member.

Problem Determination: (*where* &1 is a class member name &2 is the keyword "public", "protected" or "private")

Explanation: An attempt is made to give private access to a base class member or to give an access that is different from the access the member was

declared with. A derived class can only change the access of a base class member to public or protected if the access of that member was not private in the base class.

Programmer Response: Remove the invalid access statement or change the access specifier in the base class.

EQA2007E The array boundary in "&1" is missing.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: An array must be defined with at least one element. Use a pointer if you want to dynamically allocate memory for the array.

Programmer Response: Add an array bound.

EQA2008E The bit-field length must be an integral constant expression.

Explanation: The bit-field length, which is the value to the right of the colon, must be an integer. A constant expression has a value that can be determined during compilation and does not change during execution.

Programmer Response: Change the bit-field length to an integral constant expression.

EQA2009E "&1" is not a base class of "&2".

Problem Determination: (*where* &1 is a class name &2 is a class name)

Explanation: A derived class attempted to access elements of a class it did not inherit from. A derived class can only access elements of its base class or base classes.

Programmer Response: Ensure the class names are correct and the classes are derived properly.

EQA2010E The array bound must be a positive integral constant expression.

Explanation: The debugger detected an array declaration that did not have a constant that is greater than 0 for the array bounds. Use pointers if you want to dynamically allocate storage for arrays.

Programmer Response: Change the array bound to an integral constant expression or change it to a pointer. A constant expression has a value that can be determined during compilation and does not change during execution.

EQA2011E "&1" has the same name as its containing class.

Problem Determination: (*where* &1 is a C++ name)

Explanation: The debugger has detected conflicting names for objects within a class declaration. Nested class declarations must have different names.

Programmer Response: Change the name of the conflicting class.

EQA2012E A destructor can only be used in a function declaration or in a function call.

Explanation: The debugger has detected an incorrect destructor call.

Programmer Response: Check the call to the destructor to ensure no braces are missing. If the braces are correct, remove the destructor call.

EQA2013E An initializer is not allowed for "&1".

Problem Determination: (*where* &1 is a C/C++ name or keyword)

Explanation: The debugger detected an initializer where one is not allowed. For example, a class member declarator cannot contain an initializer.

Programmer Response: Remove the initializer.

EQA2014E The string must be terminated before the end of the line.

Explanation: The debugger detected a string that was not terminated before an end-of-line character was found.

Programmer Response: End the string before the end of the line, or use "\" to continue the string on the next line. The "\" must be the last character on the line.

EQA2015E An expression of type "&1" cannot be followed by the function call operator ().

Explanation: The debugger detected an expression followed by the function call operator. The expression must be of type function, pointer to function, or reference to function.

Programmer Response: Change the type of expression or remove the function call operator.

EQA2016E The "this" keyword is only valid in class scope.

Explanation: An attempt to use the C++ keyword `this` was detected outside class scope. The keyword `this` cannot be used outside a class member function body.

Programmer Response: Remove or move the `this` keyword.

EQA2017E A destructor cannot have arguments.

Programmer Response: Remove the arguments from the destructor.

EQA2018E A declaration has been made without a type specification.

Explanation: The debugger detected a typedef specification that did not have a type associated with it.

Programmer Response: Add a type specification to the declaration.

EQA2019E Class qualification for "&1" is not allowed.

Problem Determination: (*where* &1 is a C++ name)

Explanation: Explicit class qualification is not allowed in this context.

Programmer Response: Remove the class qualification.

EQA2020E The "&1" operator is not allowed between "&2" and "&3".

Problem Determination: (*where* &1 is a C/C++ operator &2 is a C/C++ type &3 is a C/C++ type)

Explanation: The debugger detected an illegal operator between two operands. For user-defined types, you must overload the operator to accept the user-defined types.

Programmer Response: Change the operator or change the operands.

EQA2021E "&1" cannot be converted to "&2".

Problem Determination: (*where* &1 is a C/C++ type &2 is a C/C++ type)

Explanation: The type conversion cannot be performed because there is no conversion between the types. This can occur in an initialization, assignment, or expression statement.

Programmer Response: Change one of the types or overload the operator.

EQA2022E Operand for "&1" must be a pointer or an array.

Problem Determination: (*where* &1 is a C/C++ operator)

Explanation: The specified operator must have an operand which is a pointer or an array.

Programmer Response: Change the operand to either a pointer or an array.

EQA2023E Syntax error - "&1" is not a class name.

Problem Determination: (*where* &1 is a C++ name)

Explanation: A class name must be specified in this context.

Programmer Response: Specify a class name.
Check the spelling.

EQA2024E Operand of "&1" operator must be an lvalue.

Problem Determination: (*where* &1 is a C/C++ operator)

Explanation: The debugger detected an operand that is not an lvalue. An lvalue is an expression that represents an object. For example, the left hand side of an assignment statement must be an lvalue.

Programmer Response: Change the operand to an lvalue.

EQA2025E const expression cannot be modified.

Explanation: You can initialize a const object, but its value cannot change afterwards.

Programmer Response: Eliminate the const type qualifier from the expression or do not use it with the increment/decrement operators.

EQA2026E An expression of type "&1" is not allowed on the left side of "&2&3".

Problem Determination: (*where* &1 is a C/C++ type &2 is a C/C++ operator &3 is a C/C++ name)

Explanation: The debugger detected a mismatch between the operands of an operator.

Programmer Response: Change the operand type or use a different operator.

EQA2027E "&1" is neither an immediate base class nor a nonstatic data member of class "&2".

Problem Determination: (*where* &1 is a C++ name)

Explanation: The debugger has detected an element of the initializer list that is not an element of the member list. In the constructor initializer list, you can only initialize immediate base classes and data members not inherited from a base class.

Programmer Response: Change the constructor initializer list.

EQA2028E Constructor initializer list is not allowed for nonconstructor function.

Explanation: An attempt is being made to give a constructor initializer list to a nonconstructor function. A constructor initializer list is only allowed for a constructor function.

Programmer Response: Remove the constructor initializer list.

EQA2029E Variable "&1" is not allowed in an argument initializer.

Problem Determination: (*where* &1 is a C++ name)

Explanation: The debugger has detected a default argument initialized by a parameter.

Programmer Response: Remove the parameter from the default argument initialization.

EQA2030E There are too many initializers in the initializer list.

Explanation: The debugger detected more initializers than were present in the function declaration.

Programmer Response: Remove one or more initializers from the initializer list. Make sure the number of initializers in the initializer list corresponds to the number of arguments in the function declaration.

EQA2031E An initializer is not allowed for an array allocated by "new".

Programmer Response: Remove the initializer or remove the "new" allocation.

EQA2032E The bit-field length must not be more than &1.

Problem Determination: (*where* &1 is a number)

Explanation: The bit-field length must not exceed the maximum bit size of the bit-field type.

Programmer Response: Reduce the bit-field length.

EQA2033E The type of "&1" cannot be "&2".

Problem Determination: (*where* &1 is a C++ construct &2 is a C++ type)

Explanation: The debugger detected a conflict in a type declaration.

Programmer Response: Change the type.

EQA2034E Function overloading conflict between "&1" and "&2".

Problem Determination: (*where* &1 is a function type &2 is a function type)

Explanation: The debugger detected function argument types that did not match.

Programmer Response: Change the argument declarations of the functions.

EQA2035E Declarations of the same &1 must not specify default initializers for the same argument.

Problem Determination: (*where* &1 is the word "function" or the keyword "template")

Explanation: The debugger has detected a duplicate default initializer value for the same argument in both overloaded functions or in both templates.

Programmer Response: Ensure that you wanted to declare the same function or template. If that is the case, remove one of the default initializers. Otherwise, remove one of the declarations or overload the function.

EQA2036E Call does not match any argument list for "&1".

Problem Determination: (*where* &1 is a function name)

Explanation: No variant of the overloaded function matches the argument list. The argument mismatch could be by type or number of arguments.

Programmer Response: Change the argument list on the call to the overloaded function or change the argument list on one of the overloaded function variants so that a match is found.

EQA2037E Call to "&1" matches more than one function.

Problem Determination: (*where* &1 is a function name)

Explanation: More than one variant of the overloaded function matches equally well with the argument list specified on the call.

Programmer Response: Change the argument list on the call to the overloaded function or change the argument list on one of the overloaded function variants so that only one match is found.

EQA2038E The "operator" declaration must declare a function.

Explanation: The keyword "operator" can only be used to declare an operator function.

Programmer Response: Check the declaration of the operator and make sure the function declarator () appears after it. Use the "operator" keyword to declare an operator function or remove it.

EQA2039E Operand for "&1" is of type "&2" which is not of type pointer to member.

Problem Determination: (*where* &2 is a C++ type)

Explanation: The specified operator must have an operand which is of type pointer to member.

Programmer Response: Change the operand to type pointer to member.

EQA2040E "&1" is not allowed as a function return type.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: You cannot declare a function with a function or an array as its return type.

Programmer Response: Declare the function to return a pointer to the function or the array element type.

EQA2041E "&1" is not allowed as an array element type.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: The declaration of an array of functions or references, or an array of type void is not valid.

Programmer Response: Remove the declaration or change the declaration so that it is an array of pointer to functions, pointers to references, or pointers to void.

EQA2042E const variable "&1" does not have an initializer.

Problem Determination: (*where* &1 is a variable name)

Explanation: You can only assign a value to a const variable using an initializer. This variable has no initializer, so it can never be given a value.

Programmer Response: Initialize the variable or remove the const keyword.

EQA2043E Nonstatic member "&1" must be associated with an object or a pointer to an object.

Problem Determination: (*where* &1 is a class member name)

Explanation: The debugger detected a nonstatic member making a reference to an object that has not been instantiated. You can reference only static members without associating them with an instance of the containing class.

Programmer Response: Check the spelling and the class definition. Change the name of the class or function, or define the function as static in that class.

EQA2044E "&1" is not a member of "&2".

Problem Determination: (*where* &1 is a C++ name &2 is a class name)

Explanation: The class is used explicitly as the scope qualifier of the member name, but the class does not contain a member of that name.

Programmer Response: Check the spelling of the scope qualifier. Change the scope qualifier to the class containing that member, or remove it.

EQA2045E Wrong number of arguments for "&1".

Problem Determination: (*where* &1 is a function or type name)

Explanation: A function or an explicit cast has been specified with the wrong number of arguments.

Programmer Response: Use the correct number of arguments. Ensure that overloaded functions have the correct number and type of arguments.

EQA2046E "&1" must be a class member.

Problem Determination: (*where* &1 is a C++ name)

Explanation: Conversion functions and certain operator functions must be class members. They cannot be defined globally.

Programmer Response: Remove the global definition or make the function a class member.

EQA2047E An argument type of "&1" is not allowed for "&2".

Problem Determination: (*where* &1 is a C/C++ type &2 is a function name)

Explanation: The function being declared has restrictions on what types its arguments can have. The specified type is not allowed for this argument.

Programmer Response: Change the argument type.

EQA2048E "&2" cannot have a return type of "&1".

Problem Determination: (*where* &1 is a C++ type &2 is an operator function)

Explanation: The specified operator function has the wrong return type.

Programmer Response: Change the return type.

EQA2049E The array operator must have one operand of pointer type and one of integral type.

Explanation: This error can result from the incorrect use of the array operator.

Programmer Response: Change the operands of the array operator.

EQA2050E Wrong number of arguments specified in the function call.

Explanation: The number of arguments in the function call does not match the number of arguments in the function declaration.

Programmer Response: Ensure the function declaration and function call specify the same number of arguments.

EQA2051E "&1" operator is not allowed for type "&2".

Problem Determination: (*where* &1 is a C/C++ operator &2 is a C/C++ type)

Explanation: The specified operator cannot be used with operands of this type.

Programmer Response: Change either the operator or the operands.

EQA2052E Syntax error - expected "&1" and found "&2".

Problem Determination: (*where* &1 is a C++ token &2 is a C++ token)

Explanation: A syntax error was found while parsing the expression. The message identifies what the debugger expected and what it actually found. Often the source of the error is an unmatched parenthesis or a missing semicolon.

Programmer Response: Correct the syntax.

EQA2053E "&1" is not allowed for &2.

Problem Determination: (*where* &1 is a keyword &2 is a C++ construct)

Explanation: The attribute or name cannot be specified in the given context. The debugger detected incompatible names that conflict with the language definition.

Programmer Response: Remove the attribute or name.

EQA2054E "&1" conflicts with previous "&2" declaration.

Problem Determination: (*where* &1 is a keyword &2 is a keyword)

Explanation: The declaration conflicts with a previous declaration of the same symbol.

Programmer Response: Remove one of the declarations or make them identical.

EQA2055E The "operator->" function must return a class type that contains an "operator->" function.

Explanation: The "operator->" function must return either a class type, a reference to a class type, or a pointer to class type, and the class type must itself have an "operator->" function.

Programmer Response: Change the return value of the "operator->" function.

EQA2056E Unused "&1" definition.

Problem Determination: (*where* &1 is the keyword struct or class)

Explanation: An unnamed class or struct definition was found that has no object associated with it. The definition can never be referenced. A class can be unnamed, but it cannot be passed as an argument or returned as a value. An unnamed class cannot have any constructors or destructors.

Programmer Response: Create an object for the class or struct, or remove the definition.

EQA2057E Internal debugger error at line &1 in module "&2": &3.

Explanation: The debugger detected an error within itself from which it cannot recover. The error was found within the debugger itself.

Programmer Response: Note the line and module references in this message. Contact your IBM Representative Debug Tool support.

EQA2058E Reference to member "&1" of undefined class "&2".

Problem Determination: (*where* &1 is a member name &2 is a class name)

Explanation: The member has been explicitly given the specified class as a scope qualifier but the class (and hence the member) has not been defined.

Programmer Response: Check the spelling of the scope qualifier. Change the scope qualifier to the class containing that member, or remove it.

EQA2059E Pointer conversion may be wrong if the classes are related in a multiple inheritance hierarchy.

Explanation: The relationship between the classes in a pointer conversion is not known. If the target class is later defined as a base class of the source class in a multiple inheritance, this conversion will be wrong if the value of the pointer should have been modified by the conversion.

Programmer Response: Change the ambiguous reference in the conversion.

EQA2060E The reference variable "&1" is uninitialized.

Problem Determination: (*where* &1 is a variable name)

Explanation: Reference variables must be initialized.

Programmer Response: Initialize the reference variable or remove it.

EQA2061E "&1" must already be declared.

Problem Determination: (*where* &1 is a class or enum name)

Explanation: The specified class or enum name must have been declared before this use of the name.

Programmer Response: Declare the class or enum name before you use it. Check the correct spelling of the name.

EQA2062E Unrecognized source character "&1", code point &2.

Problem Determination: (*where* &1 is a character &2 is an integer)

Explanation: The specified character is not a valid character in a C/C++ expression. The code point displayed represents its hexadecimal value.

Programmer Response: Remove the character.

EQA2063E A local class cannot have a non-inline member function "&1".

Problem Determination: (*where* &1 is a function name)

Explanation: A class declared within a function must have all of its member functions defined inline, because the class will be out of scope before non-inline functions can be defined.

Programmer Response: Define the functions inline, or move the class definition out of the scope of the function.

EQA2064E The size of "&1" is unknown in "&2" expression.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: The operation cannot be performed because the size of the specified type is not known.

Programmer Response: Ensure the size of the type is known before this expression.

EQA2065E Assignment in logical expression.

Explanation: The logical expression contains an assignment (=). An equality comparison (==) might have been intended.

Programmer Response: Change the operator or the expression.

EQA2066E Conversion from "&1" to "&2" may cause truncation.

Problem Determination: (*where* &1 is a C/C++ type &2 is a C/C++ type)

Explanation: The specified conversion from a wider to a narrower type might cause the loss of significant data.

Programmer Response: Remove the conversion from a wider to a narrower type.

EQA2067E "goto &1" bypasses initialization of "&2".

Problem Determination: (*where* &1 is the C/C++ label used with the goto keyword &2 is the variable being initialized)

Explanation: Jumping past a declaration with an explicit or implicit initializer is not valid unless the declaration is in an inner block or unless the jump is from a point where the variable has already been initialized.

Programmer Response: Enclose the initialization in a block statement.

EQA2068E References to "&1" may be ambiguous. The name is declared in base classes "&2" and "&3".

Problem Determination: (*where* &3 is a C++ class name)

Explanation: The debugger detected the base classes of a derived class have members with the same names. This will cause ambiguity when the member name is used. This is only an informational message because the declaration of a member with an ambiguous name in a derived class is not an error. The ambiguity is only flagged as an error if you use the ambiguous member name.

Programmer Response: Change one of the names, or always fully qualify the name.

EQA2069E Ambiguous reference to "&1", declared in base classes "&2" and "&3".

Problem Determination: (*where* &3 is a C++ class name)

Explanation: The derived class made a reference to a member that is declared in more than one of its base classes and the debugger cannot determine which base class member to choose.

Programmer Response: Change one of the names, or always fully qualify the name.

EQA2070E Conversion from "&1" to "&2" is ambiguous.

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: There is more than one way to perform the specified conversion. This ambiguity can be caused by an overloaded function.

Programmer Response: Change or remove the conversion.

EQA2071E "&1" is only valid for non-static member functions.

Problem Determination: (*where* &1 is the keyword const or volatile)

Explanation: const and volatile are only significant for nonstatic member functions, since they are applied to the "this" pointer.

Programmer Response: Remove const and volatile from all static members.

EQA2072E Character literal is null.

Explanation: An empty character literal has been specified. A string literal might have been intended.

Programmer Response: Remove the character literal, change it to a string literal, or give it a value.

EQA2073E "&1" has more than one base class "&2".

Problem Determination: (*where* &1 is a class name &2 is a class name)

Explanation: A derived class has inherited the same base class in more than one path and the debugger cannot determine which one to choose.

Programmer Response: Remove one of the inheritances.

EQA2074E "&1" is a &2 base class of "&3".

Problem Determination: (*where* &1 is a class name &2 is the keyword private or protected &3 is a class name)

Explanation: An attempt is being made to convert a pointer to a derived class into a pointer to a private or protected base class.

Programmer Response: Remove the pointer conversion.

EQA2075E &1 "&2" is not allowed in a union.

Problem Determination: (*where* &1 is a C++ construct &2 is a C++ name)

Explanation: Unions must not be declared with base classes, virtual functions, static data members, members with constructors, members with destructors, or members with class copying assignment operators.

Programmer Response: Remove any such members from the union declaration.

EQA2076E union "&1" cannot be used as a base class.

Problem Determination: (*where* &1 is a union name)

Explanation: Unions cannot be used as base classes for other class declarations.

Programmer Response: Remove the union as a base class for other class declarations.

EQA2077E Local variable "&1" is inaccessible from "&2".

Problem Determination: (*where* &1 is a variable name &2 is a class name)

Explanation: An automatic variable within a function is not accessible from local classes declared within the function.

Programmer Response: Remove the reference to the local variable, or move the variable to a different scope.

EQA2078E Value of enumerator "&1" is too large.

Problem Determination: (*where* &1 is an enumerator name)

Explanation: The value of an enumerator must be a constant expression that is promotable to a signed integer value.

Programmer Response: Reduce the value of the enumerator.

EQA2079E A constant is being used as a conditional expression.

Explanation: The condition to an if, for, or switch is constant and therefore, that condition will always hold.

Programmer Response: Remove the constant or ignore this message.

EQA2080E The argument to a not (!) operator is constant.

Explanation: The debugger has detected a constant after the ! operator which might be a coding error.

Programmer Response: Remove the constant or ignore this message.

EQA2081E There is more than one character in a character constant.

Explanation: Using more than one character in a character constant (for example, 'ab') might not be portable across machines.

Programmer Response: Remove the extra character(s) or change the character constant to a string constant.

EQA2082E Possible pointer alignment problem with the "&1" operator.

Problem Determination: (*where* &1 is a C/C++ operator)

Explanation: A pointer that points to a type with less strict alignment requirements is being assigned, cast, returned or passed as a parameter to a pointer that is a more strictly aligned type. This is a potential portability problem.

Programmer Response: Remove the pointer reference or change the alignment.

EQA2083E A constant expression is being cast to a pointer.

Explanation: Casting a constant value to a pointer is not portable to other platforms.

Programmer Response: Remove the constant expression from the cast expression.

EQA2084E Precision will be lost in assignment to (possibly sign-extended) bit-field "&1".

Explanation: A constant is being assigned to a signed bit field that cannot represent the constant. Precision might be lost and the stored value will be incorrect.

Programmer Response: Increase the size of the bit field.

EQA2085E Precision will be lost in assignment to bit-field "&1".

Explanation: A constant is being assigned to a bit field, and because the bit field has a smaller size, the precision will be lost.

Programmer Response: Change the assignment expression.

EQA2086E Enumeration type clash with the "&1" operator.

Problem Determination: (*where* &1 is a C++ operator)

Explanation: Operands from two different enumerations are used in an operation.

Programmer Response: Ensure both operands are from the same enumeration.

EQA2087E Comparison of an unsigned value with a negative constant.

Explanation: An unsigned value is being compared to a negative number. The unsigned value will always compare greater than the negative number. This might be a programming error.

Programmer Response: Remove the comparison or change the type.

EQA2088E Unsigned comparison is always true or always false.

Explanation: The comparison is either "unsigned >= 0", which is always true, or "unsigned < 0", which is always false.

Programmer Response: Remove or change the comparison.

EQA2089E Comparison is equivalent to "unsigned value &1 0".

Explanation: The comparison is either "unsigned > 0" or "unsigned <= 0", and could be written as "unsigned != 0" or "unsigned == 0".

Programmer Response: Change the comparison.

EQA2090E Argument &1 for "&2" must be of type "&3".

Problem Determination: (*where* &1 is an argument number &2 is a function name &3 is a C++ type)

Explanation: The indicated function requires an argument of a particular type. However, the argument specified is of a different type than the type required.

Programmer Response: Ensure that the argument is of the correct type.

EQA2091E Definition of "&1" is not allowed.

Problem Determination: (*where* &1 is the keyword class, struct, union or enum.)

Explanation: You cannot define a type in a type cast or a conversion function declaration.

Programmer Response: Move the definition to a new location, or remove it.

EQA2092E Reference to "&1" is not allowed.

Problem Determination: (*where* &1 is a C++ name)

Explanation: The name has a special meaning in a C++ program and cannot be referenced in this way.

Programmer Response: Remove the reference.

EQA2093E Escape sequence &1 is out of the range 0-&2. Value is truncated.

Problem Determination: (*where* &2 is the maximum allowed value of the escape sequence)

Programmer Response: Make the escape sequence small enough to fit the specified range.

EQA2094E A wide character constant is larger than the size of a "wchar_t". Only the last character is used.

Explanation: A wide character constant can only contain one character.

Programmer Response: Make the wide character constant smaller.

EQA2095E A character constant is larger than the size of an "int". Only the rightmost &1 characters are used.

Problem Determination: (*where* &1 is an integer number)

Programmer Response: Make the character constant smaller.

EQA2096E Linkage specification must be at file scope.

Explanation: A linkage specification can only be defined at file scope, that is, outside all functions and classes.

Programmer Response: Move the linkage specification or remove it.

EQA2097E Default initializers cannot be followed by uninitialized arguments.

Explanation: If a default initializer is specified in an argument list, all following arguments must also have default initializers.

Programmer Response: Remove the default initializers, or provide them for the following arguments, or move the arguments to the end of the list.

EQA2098E You cannot take the address of "&1".

Problem Determination: (*where* &1 is a C++ name)

Explanation: You cannot take the address of a constructor, a destructor or a reference member.

Programmer Response: Remove the address operator (&) from the expression or remove the expression.

EQA2099E Duplicate qualifier "&1" ignored.

Problem Determination: (*where* &1 is a keyword)

Explanation: The keyword has been specified more than once. Extra occurrences are ignored.

Programmer Response: Remove one of the duplicate qualifiers.

EQA2100E "&1" operator cannot be overloaded.

Problem Determination: (*where* &1 is an operator name)

Explanation: The specified operator cannot be overloaded using an operator function. The following operators cannot be overloaded: . * :: ?:

Programmer Response: Remove the overloading declaration or definition.

EQA2101E At least one argument of "&1" must be of class or enum type.

Problem Determination: (*where* &1 is an operator function name)

Explanation: The nonmember operator function must have at least one argument which is of class or enum type.

Programmer Response: Add an argument of class or enum type.

EQA2102E The divisor for the modulus or division operator cannot be zero.

Programmer Response: Change the expression used in the divisor.

EQA2103E The address of the bit-field "&1" cannot be taken.

Problem Determination: (*where* &1 is a member name)

Explanation: An expression attempts to take the address of a bit-field, or to use the bit-field to initialize a reference variable or argument.

Programmer Response: Remove the expression causing the error.

EQA2104E "&1" must not have default initializers.

Problem Determination: (*where* &1 is an operator function name or "template function")

Explanation: Default initializers are not allowed within the declaration of an operator function or a template function.

Programmer Response: Remove the default initializers.

EQA2105E The &1 "&2" cannot be initialized because it does not have a default constructor.

Problem Determination: (*where* &1 is 'base class' or 'class member' &2 is a C++ name)

Explanation: The specified base class or member cannot be constructed since it is not initialized in the constructor initializer list and its class has no default constructor.

Programmer Response: Specify a default constructor for the class or initialize it in the constructor initializer list.

EQA2106E Template class "&1" has the wrong number of arguments.

Problem Determination: (*where* &1 is a template class name)

Explanation: A template class instantiation has a different number of template arguments than the template declaration.

Programmer Response: Ensure that the template class has the same number of declarations as the template declaration.

EQA2107E Non-&1 member function "&2" cannot be called for a &1 object.

Problem Determination: (*where* &2 is a function name with arguments)

Explanation: The member function is being called for a const or volatile object but the member function has not been declared with the const or volatile qualifier.

Programmer Response: Supply a version of the member function with the correct set of "const" and "volatile" qualifiers.

EQA2108E Null statement.

Explanation: Possible extraneous semicolon has been specified.

Programmer Response: Check for extra semicolons in statement.

EQA2109E Bit-field "&1" cannot be used in a conditional expression that is to be modified.

Explanation: The bit-field is part of a conditional expression that is to be modified. Only objects that can have their address taken are allowed as part of such an expression, and you cannot take the address of a bit field.

Programmer Response: Remove the bit-field from the conditional expression.

EQA2110E The "&1" qualifier cannot be applied to "&2".

Problem Determination: (*where* &2 is a name or a type)

Explanation: The qualifier is being applied to a name or a type for which it is not valid.

Programmer Response: Remove the qualifier.

EQA2111E Local type "&1" cannot be used as a &2 argument.

Problem Determination: (*where* &2 is either the keyword template or the keyword function)

Explanation: The type cannot be used as a function argument or in the instantiation of a template because the scope of the type is limited to the current function.

Programmer Response: Remove the local type.

EQA2112E Default initializers for nontype template arguments are only allowed for class templates.

Explanation: Default initializers have been given for nontype template arguments, but the template is not declaring a class.

Programmer Response: Remove the default initializers.

EQA2113E A function argument must not have type "void".

Explanation: A function argument can be an expression of any object type. However, "void" is not the type of any object and cannot be used as an argument type.

Programmer Response: Change the type of the function argument.

EQA2114E Insufficient memory in line &1 of file "&2".

Problem Determination: (*where* &1 is a line number &2 is a file name)

Explanation: The debugger ran out of memory.

Programmer Response: Increase your storage and rerun.

EQA2115E Unable to initialize source conversion from codepage &1 to codepage &2.

Problem Determination: (*where* &1 is a codepage name i.e. IBM-1047 &2 is a codepage name i.e. IBM-1047)

Explanation: An error occurred when attempting to convert source between the codepages specified.

Programmer Response: Ensure the codepages are correct and that conversion between these codepages is supported.

EQA2116E An object of abstract class "&1" cannot be created.

Problem Determination: (*where* &1 is a class name)

Explanation: You cannot create instances of abstract classes. An abstract class is a class that has or inherits at least one pure virtual function.

Programmer Response: Derive another object from the abstract class.

EQA2117E Invalid use of an abstract class.

Explanation: An abstract class must not be used as an argument type, as a function return type, or as the type of an explicit conversion.

Programmer Response: Derive another class from the abstract, instantiate it so it becomes a concrete object, and then use it instead.

EQA2118E "&1" has been used more than once in the same base class list.

Problem Determination: (*where* &1 is base class name)

Explanation: A base class can only be specified once in the base class list for a derived class.

Programmer Response: Remove one of the specifications.

EQA2119E Template argument &1 of type "&2" does not match declared type "&3".

Problem Determination: (*where* &1 is an integer number &2 is a C++ type &3 is a C++ type)

Explanation: A nontype template argument must have a type that exactly matches the type of the corresponding argument in the template declaration.

Programmer Response: Ensure that the types match.

EQA2120E Template argument &1 of type "&2" is not an allowable constant value or address.

Problem Determination: (*where* &1 is an integer number &2 is a C++ type)

Explanation: A nontype template argument must be a constant value or the address of an object, function, or static data member that has external linkage. String literals cannot be used as template arguments because they have no name, and therefore no linkage.

Programmer Response: Change the template argument.

EQA2121E Template argument list is empty.

Explanation: At least one template argument must be specified in a template declaration.

Programmer Response: Specify a template argument in the declaration.

EQA2122E Formal template argument &1 is of type "&2" which is not an allowable integral, enumeration, or pointer type.

Problem Determination: (*where* &1 is an integer number &2 is a C++ type)

Explanation: A nontype template argument must be of integral, or enumeration, or pointer type, so that it can be matched with a constant integral value.

Programmer Response: Change the template argument.

EQA2123E "&1" is defined in a template declaration but it is not a static member.

Problem Determination: (*where* &1 is a C++ name)

Explanation: A member of a template class defined in a template declaration must be a static member.

Programmer Response: Make the member static or remove it from the template declaration.

EQA2124E Template argument "&1" is not used in the declaration of the name or the argument list of "&2".

Problem Determination: (*where* &1 is a template argument name &2 is a C++ name)

Explanation: All template arguments for a nonclass template must be used in the declaration of the name or the function argument list.

Programmer Response: Ensure all template arguments are used in the declaration of the name or the function argument list.

EQA2125E Template declaration does not declare a class, a function, or a template class member.

Explanation: Following the template argument, a template declaration must declare a class, a function, or a static data member of a template class.

Programmer Response: Change the template declaration to declare a class, a function, or a template class member.

EQA2126E Return type "&1" for function "&2" differs from previous return type of "&3".

Problem Determination: (*where* &1 is a C/C++ type &2 is a function name &3 is a C/C++ type)

Explanation: The declaration of the function differs from a previous declaration in only the return type.

Programmer Response: Change the return type so that it matches the previous return type.

EQA2127E "&1" is a member of "&2" and cannot be used without qualification.

Problem Determination: (*where* &2 is a possibly qualified class name)

Explanation: The specified name is a class member, but no class qualification has been used to reference it.

Programmer Response: Add a class qualification to the class member.

EQA2128E "&1" cannot be initialized multiple times.

Problem Determination: (*where* &1 is a member or base class name)

Explanation: An initializer was already specified in the constructor definition.

Programmer Response: Remove the additional initializer.

EQA2129E No suitable copy assignment operator exists to perform the assignment.

Explanation: A copy assignment operator exists but it does not accept the type of the given parameter.

Programmer Response: Change the copy assignment operator.

EQA2130E Explicit call to constructor "&1" is not allowed.

Problem Determination: (*where* &1 is a constructor name)

Explanation: You cannot call a constructor explicitly. It is called implicitly when an object of the class is created.

Programmer Response: Remove the call to the constructor.

EQA2131E No default constructor exists for "&1".

Problem Determination: (*where* &1 is a class name)

Explanation: An array of class objects must be initialized by calling the default constructor, but one has not been declared.

Programmer Response: Declare a default constructor for the array.

EQA2132E More than one default constructor exists for "&1".

Problem Determination: (*where* &1 is a class name)

Explanation: An array of class objects must be initialized by calling the default constructor, but the call is ambiguous.

Programmer Response: Ensure that only one default constructor exists.

EQA2133E The debugger cannot generate a default copy constructor for "&1".

Explanation: The default copy constructor cannot be generated for this class because there exists a member or base class that has a private copy constructor, or there are ambiguous base classes, or this class has no name.

Programmer Response: Ensure that a member or base class does not have a private copy constructor. If not then ensure the class is named and there are no ambiguous references to base classes.

EQA2134E The debugger cannot generate a default copy assignment operator for "&1".

Explanation: The default copy assignment operator cannot be generated for this class because it has a const member or a reference member or a member (or base class) with a private copy assignment operator.

Programmer Response: Ensure there are no const members, reference members or members with a private copy assignment operator.

EQA2135E Pure virtual function called.

Explanation: A call has been made to a pure virtual function from a constructor or destructor. In such functions, the pure virtual function would not have been overridden by a derived class and a run-time error would occur.

Programmer Response: Remove the call to the pure virtual function.

EQA2136E "&1" is not allowed as a conversion function type.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: A conversion function cannot be declared with a function or an array as its conversion type, since the type cannot be returned from the function.

Programmer Response: Declare the function as converting to a pointer to the function or the array element type.

EQA2137E Syntax error - "&1" is followed by "&3" but is not the name of a &2.

Problem Determination: (*where* &1 is a C++ name &2 is the keyword class or template &3 is the token '::' or '<')

Explanation: The name is not a class or template name but the context implies that it should be.

Programmer Response: Change the name to a class or template name.

EQA2138E The previous &1 messages apply to the definition of template "&2".

Problem Determination: (*where* &1 is an integer number &2 is a template name)

Explanation: The instantiation of the specified template caused the messages, even though the line numbers in the messages refer to the original template declaration.

Programmer Response: This message supplies additional information for previously emitted messages. Refer to the descriptions of those messages for recovery information.

EQA2139E The previous message applies to the definition of template "&1".

Problem Determination: (*where* &1 is a template name)

Explanation: The instantiation of the specified template caused the message, even though the line number in the message refers to the original template declaration.

Programmer Response: This message supplies additional information for previously emitted messages. Refer to the descriptions of those messages for recovery information.

EQA2140E No suitable constructor exists for conversion from "&1" to "&2".

Problem Determination: (*where* &1 is a class name &2 is a C++ type)

Explanation: A constructor is required for the class but no user-defined constructor exists and the debugger could not generate one.

Programmer Response: Create a suitable constructor for conversion.

EQA2141E class "&1" does not have a copy assignment operator.

Problem Determination: (*where* &1 is a class name)

Explanation: A copy assignment operator is required for the class but no user-defined copy assignment operator exists and the debugger could not generate one.

Programmer Response: Create a copy assignment operator.

EQA2142E "&1" cannot be used as a template name since it is already known in this scope.

Problem Determination: (*where* &1 is a C++ name)

Explanation: A template name must not match the name of an existing template, class, function, object, value or type.

Programmer Response: Change one of the template names.

EQA2143E "&1" is expected for template argument &2.

Problem Determination: (*where* &1 is either 'expression' or 'type name' &2 is an integer number)

Explanation: Either the argument is a type and the template has a nontype argument, or the argument is an expression and the template has a type argument.

Programmer Response: Ensure the argument matches the template.

EQA2144E "&1" cannot be defined before the template definition of which it is an instance.

Problem Determination: (*where* &1 is a class template name)

Explanation: An explicit definition of a template class cannot be given before the corresponding template definition.

Programmer Response: Move the template definition so that it occurs before any template class definitions.

EQA2145E An ellipsis (...) cannot be used in the argument list of a template function.

Explanation: Since an exact match is needed for template functions, an ellipsis cannot be used in the function argument list.

Programmer Response: Remove the ellipsis from the argument list.

EQA2146E The suffix for the floating point constant is not valid.

Explanation: You have provided an incorrect suffix for the floating point constant. Valid suffixes for floating point constants are L and F.

Programmer Response: Change the suffix for the floating point constant.

EQA2147E Statement has no effect.

Explanation: The expression has no side effects and produces a result that is not used.

Programmer Response: Remove the statement or use its result.

EQA2148E The suffix for the integer constant is not valid.

Explanation: The integer constant is a suffix letter that is not recognized as a valid suffix.

Programmer Response: Change the suffix to either "u" or "l".

EQA2149E The expression contains a division by zero.

Programmer Response: Remove the division by zero from the expression

EQA2150E The expression contains a modulus by zero.

Programmer Response: Remove the modulus by zero from the expression.

EQA2151E Static member "&1" can only be defined at file scope.

Programmer Response: Move the static member so that it is defined at file scope.

EQA2152E "&1" needs a constructor because &2 "&3" needs a constructor initializer.

Problem Determination: (*where* &1 is a class name &2 is 'class member' or 'base class' &3 is the member or base class name.)

Explanation: You have not provided a constructor for the class, because the member or base class does not have a default constructor.

Programmer Response: Add a constructor.

EQA2153E Conversion from "&1" to a reference to a non-const type "&2" requires a temporary.

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: A temporary can only be used for conversion to a reference type when the reference is to a const type.

Programmer Response: Change to a const type.

EQA2154E "&2" is too small to hold a value of type "&1".

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: A conversion from a pointer type to an integral type is only valid if the integral type is large enough to hold the pointer value.

Programmer Response: Remove the conversion from a pointer type to an integral type or use a larger integral type.

EQA2155E Object of type "&1" cannot be constructed from "&2" expression.

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: There is no constructor taking a single argument that can be called using the given expression.

Programmer Response: Change the expression.

EQA2156E The debugger cannot generate a copy constructor for conversion to "&1".

Problem Determination: (*where* &1 is a C++ type)

Explanation: A copy constructor is required for the conversion. No suitable user-defined copy constructor exists and the debugger could not generate one.

Programmer Response: Create a copy constructor for the conversion.

EQA2157E No suitable constructor or conversion function exists for conversion from "&1" to "&2".

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: A constructor or conversion function is required for the conversion but no such constructor or function exists.

Programmer Response: Create a constructor or conversion function for the conversion.

EQA2158E Syntax error - "&1" has been inserted before "&2".

Problem Determination: (*where* &1 is a token &2 is a token)

Explanation: A syntax error was found while parsing the expression. The message identifies what the debugger expected and what it actually found.

Programmer Response: Correct the syntax.

EQA2159E Call to "&1" matches some functions best in some arguments, but no function is a best match for all arguments.

Problem Determination: (*where* &1 is a function name)

Explanation: No function matches each call argument as well as or better than all other functions.

Programmer Response: Change the function call so that it matches only one function.

EQA2160E Call matches "&1".

Problem Determination: (*where* &1 is a function name and type)

Explanation: The debugger detected an overloaded function or operator that is similar to another and is providing additional information.

Programmer Response: Ensure this is the desired match.

EQA2161E Cannot adjust access of "&1::&2" because a member in "&3" hides it.

Problem Determination: (*where* &1 is a class name &2 is a member name &3 is the name of the derived class.)

Explanation: You cannot modify the access of the specified member because a member of the same name in the specified class hides it.

Programmer Response: Remove the access adjustment expression or unhide the member.

EQA2162E "&1" cannot be redeclared.

Problem Determination: (*where* &1 is a C++ name)

Explanation: The specified name cannot be redeclared because it has already been used.

Programmer Response: Change or remove one of the declarations.

EQA2163E Syntax error - "&1" is not allowed; "&2" has already been specified.

Problem Determination: (*where* &1 is a keyword &2 is a keyword)

Explanation: You cannot use both of the specified attributes in the same declaration.

Programmer Response: Remove the attributes.

EQA2164E Call to "&1" matches more than one template function.

Problem Determination: (*where* &1 is a function name and type)

Explanation: More than one template for the function matches equally well with the argument list specified on the call.

Programmer Response: Change the call so that it matches only one template function.

EQA2165E "&1" is declared inline, but is undefined.

Problem Determination: (*where* &1 is a function name and type)

Explanation: An inline function must be defined in every compilation unit in which it is used.

Programmer Response: Define the inline function in this compilation unit.

EQA2166E Non-&1 member function called for a &1 object via pointer of type "&2".

Problem Determination: (*where* &2 is a pointer or member-pointer type)

Explanation: The member function is being called indirectly for a const or volatile object but it has not been declared with the corresponding const or volatile attribute.

Programmer Response: Ensure that the function call and the function declaration match.

EQA2167E "&1" cannot be a base of "&2" because "&3" contains the type name "&2".

Problem Determination: (*where* &1 is a class name &2 is both the derived class name and a type name &3 is the class containing &2)

Explanation: A class cannot inherit a type name that is the same as the class name.

Programmer Response: Change the name of either the derived class or the inherited class.

EQA2168E "&1" cannot be a base of "&2" because "&3" contains the enumerator "&2".

Problem Determination: (*where* &1 is a class name &2 is both the derived class name and the enumerator name &3 is the class containing &2)

Explanation: A class cannot inherit an enumerator with the same name as the class name.

Programmer Response: Change the name of either the derived class or the inherited enumerator.

EQA2169E Symbol length of &1 exceeds limit of &2 bytes.

Problem Determination: (*where* &1 is an integer number &2 is an integer number)

Explanation: The debugger limit for the length of a symbol has been exceeded.

Programmer Response: Shorten the symbol length.

EQA2170E The result of this pointer to member operator can be used only as the operand of the function call operator ().

Explanation: If the result of the .* or →* is a function, that result can be used only as the operand for the function call operator ().

Programmer Response: Make the result the operand of the function call operator ().

EQA2171E When "&1" is used as an operand to the arrow or dot operator, the result must be used with the function call operator ().

Problem Determination: (*where* &1 is a member name)

Explanation: If the result of the dot or arrow operator is a function, that result can be used only as the operand for the function call operator ().

Programmer Response: Make the result the operand of the function call operator ().

EQA2172E A class with a reference or const member needs a constructor.

Explanation: const and reference members must be initialized in a constructor initializer list.

Programmer Response: Add a constructor to the class.

EQA2173E Base class initializers cannot contain virtual function calls.

Explanation: The virtual function table pointers are not set up until after the base classes are initialized.

Programmer Response: Remove the call to a virtual function in the base class initializer.

EQA2174E The previous declaration of "&1" did not have a linkage specification.

Explanation: If you want to declare a linkage specification for a function, it must appear in the first declaration of the function.

Programmer Response: Add a linkage specification to the first declaration of the function.

EQA2175E The destructor for "&1" does not exist. The call is ignored.

Problem Determination: (*where* &1 is a C++ type)

Explanation: The destructor call is for a type that does not have a destructor. The call is ignored.

Programmer Response: Add a destructor to the type.

EQA2176E "&1" has been added to the scope of "&2".

Problem Determination: (*where* &1 is the name on a friend declaration &2 is a class name)

Explanation: Because the friend class has not been declared yet, its name has been added to the scope of the class containing the friend declaration.

Programmer Response: If this is not intended, move the declaration of the friend class so that it appears before it is declared as a friend.

EQA2177E The body of friend member function "&1" cannot be defined in the member list of "&2".

Problem Determination: (*where* &1 is the friend member function &2 is a class name)

Explanation: A friend function that is a member of another class cannot be defined inline in the member list.

Programmer Response: Define the body of the friend function at file scope.

EQA2178E The initializer list must be complete because "&1" does not have a default constructor.

Problem Determination: (*where* &1 is a class without a default constructor.)

Explanation: An array of objects of a class with constructors uses the constructors in initialization. If there are fewer initializers in the list than elements in the array, the default constructor is used. If there is no default constructor the initializer list must be complete.

Programmer Response: Complete the initializer list or add a default constructor to the class.

EQA2179E A pure virtual destructor needs an out-of-line definition in order for its class to be a base of another class.

Programmer Response: Move the definition of the pure virtual destructor so that it is not inline.

EQA2180E The braces in the initializer are incorrect.

Programmer Response: Correct the braces on the initializer.

EQA2181E Invalid octal integer constant.

Explanation: The octal integer constant contains an '8' or a '9'. Octal numbers include 0 through 7.

Programmer Response: Ensure that the octal integer constant is valid.

EQA2182E All the arguments must be specified for "&1" because its default arguments have not been checked yet.

Problem Determination: (*where* &1 is a function name and type)

Explanation: For member functions, names in default argument expressions are bound at the end of the class declaration. Calling a member function as part of a second member function's default argument is an error if the first member function's default arguments have not been checked and the call does not specify all of the arguments.

Programmer Response: Specify all the arguments for the function.

EQA2183E Ellipsis (...) cannot be used for "&1".

Problem Determination: (*where* &1 is an operator name)

Explanation: An operator function has been specified with an ellipsis (...), but since the number of operands of an operator are fixed, an ellipsis is not allowed.

Programmer Response: Remove the ellipsis, and specify the correct number of operands.

EQA2184E Syntax error - expected "&1" or "&2" and found "&3".

Problem Determination: (*where* &1 is a token &2 is a token &3 is a token)

Explanation: A syntax error was found while parsing the program. The message identifies what the debugger expected and what it actually found.

Programmer Response: Correct the syntax error.

EQA2185E A character constant must end before the end of the line.

Explanation: The debugger detected a character constant that was not terminated before an end-of-line character was found.

Programmer Response: End the character constant or use "\" to continue it on the next line. The "\" must be the last character on the line.

EQA2186E A pure virtual function initializer must be 0.

Explanation: To declare a pure virtual function use an initializer of 0.

Programmer Response: Set the virtual function initializer to 0.

EQA2187E "&1" is given "&2" access.

Problem Determination: (*where* &1 is a member name &1 is the keyword public, protected or private)

Explanation: Access of the class has changed.

Programmer Response: Ensure this change is as intended.

EQA2188E "&1" has been qualified with the "this" pointer.

Problem Determination: (*where* &1 is a member name)

Programmer Response: Ensure this qualification is intended.

EQA2189E Invalid escape sequence; the backslash is ignored.

Explanation: You have provided invalid character(s) after the backslash that does not represent an escape sequence. Therefore, the backslash is ignored and the rest of the escape sequence is read as is.

Programmer Response: Ensure the escape sequence is valid.

EQA2190E The result of an address expression is being deleted.

Programmer Response: Ensure this action is intended.

EQA2191E Conversion from "&1" to "&2" matches more than one conversion function.

Explanation: More than one conversion function could be used to perform the specified conversion.

Programmer Response: Create a new conversion function for this conversion or change one of the types.

EQA2192E Conversion matches "&1".

Problem Determination: (*where* &1 is a function name and type)

Programmer Response: Ensure this is the intended match.

EQA2193E "&1" cannot be initialized with an initializer list.

Problem Determination: (*where* &1 is a class name)

Explanation: Only an object of a class with no constructors, no private or protected members, no virtual functions and no base classes can be initialized with an initializer list.

Programmer Response: Remove the class from the initializer list.

EQA2194E A pointer to a virtual base "&1" cannot be converted to a pointer to a derived class "&2".

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: A pointer to a class B can be explicitly converted to a pointer to a class D that has B as a direct or indirect base class, only if an unambiguous conversion from D to B exists, and B is not a virtual base class.

Programmer Response: Remove the conversion of the pointer.

EQA2195E The arguments passed using the ellipsis may not be accessible.

Explanation: Arguments passed using an ellipsis are only accessible if there is an argument preceding the ellipsis and the preceding argument is not passed by reference.

Programmer Response: Ensure that there is an argument preceding the ellipsis and that the preceding argument is not passed by reference.

EQA2196E Assignment to a constant expression is not allowed.

Explanation: The left hand side of the assignment operator is an expression referring to a "const" location. For example, in "a.b", either "b" is a "const" member or "a" is a "const" variable.

Programmer Response: Remove the assignment.

EQA2197E Assignment to const variable "&1" is not allowed.

Problem Determination: (*where* &1 is the variable name)

Explanation: The left hand side of the assignment operator is a variable with the "const" attribute. "const" variables can be initialized once at the point where they are declared, but cannot be subsequently assigned new values.

Programmer Response: Remove the assignment to the const variable.

EQA2198E The return type for the "operator->" cannot be the containing class.

Explanation: The return type for the "operator->" must be a pointer to a class type, a class type, or a reference to a class type. If it is a class or reference, the class must be previously defined and must contain an "operator->" function.

Programmer Response: Change the return type for the "operator->".

EQA2199E The previous message applies to function argument &1.

Problem Determination: (*where* &1 is an integer corresponding to the function argument number)

Explanation: The previous message applies to the specified argument number. This message does not indicate another error or warning, it indicates which argument of the function call is the subject of the previous message.

EQA2200E Conversion from "&1" to a reference to a non-const type "&2" requires a temporary.

Problem Determination: (*where* &1 is a C++ type &2 is a C++ type)

Explanation: A temporary can only be used for conversion to a reference type when the reference is to a const type. This is a warning rather than an error message because the "compat" language level is active.

Programmer Response: Change the reference so that it is to a const type.

EQA2201E The pointer to member function must be bound to an object when it is used with the function call operator ().

Explanation: The pointer to member function must be associated with an object or a pointer to an object when it is used with the function call operator ().

Programmer Response: Remove the pointer or associate it with an object.

EQA2202E The direct base "&1" of class "&2" is ignored because "&1" is also an indirect base of "&2".

Problem Determination: (*where* &1 is a base class name)

Explanation: A reference to a member of "&1" will be ambiguous because it is inherited from two different paths.

Programmer Response: Remove the indirect inheritance.

EQA2203E The "&1" operator cannot be applied to undefined class "&2".

Problem Determination: (*where* &1 is a class type)

Explanation: A class is undefined until the definition of its tag has been completed. A class tag is undefined when the list describing the name and type of its members has not been specified. The definition of the tag must be given before the operator is applied to the class.

Programmer Response: Complete the definition of the class before applying an operator to it.

EQA2204E "&1" hides the &2 "&3".

Problem Determination: (*where* &1 is the name of the derived class's member &2 is "pure virtual" or "virtual" &3 is the name of the hidden virtual function)

Explanation: A member in the derived class hides a virtual function member in a base class.

Programmer Response: Ensure the hiding of the virtual function member is intended.

EQA2205E "&1" is not the name of a function.

Problem Determination: (*where* &1 is a C++ name)

Explanation: A function name is required in this context. The specified name has been declared but it is not the name of a function.

Programmer Response: Check the spelling. If necessary, change to a function name.

EQA2206E The virtual functions "&1" and "&2" are ambiguous since they override the same function in virtual base class "&3".

Problem Determination: (*where* &1 is a function name and type &2 is a function name and type)

Explanation: The two functions are ambiguous and the virtual function call mechanism will not be able to choose the correct one at run time.

Programmer Response: Remove one of the virtual functions.

EQA2207E The "this" address for "&1" is ambiguous because there are multiple instances of "&2".

Problem Determination: (*where* &1 is a function name and type &2 is a class name)

Explanation: Two or more "this" addresses are possible for this virtual function. The virtual function call mechanism will not be able to determine the correct address at run time.

Programmer Response: Remove the "this" expression or change the function name.

EQA2208E Conversion from "&1" matches more than one conversion function.

Problem Determination: (*where* &1 is a function name and type)

Explanation: More than one conversion function could be applied to perform the conversion from the specified type.

Programmer Response: Create a new conversion function or remove the conversion.

EQA2209E "&1" cannot be a base of "&2" because "&3" contains a member function called "&2".

Problem Determination: (*where* &1 is a class name &2 is both the derived class name and the member function &3 is the class containing &2)

Explanation: A class cannot inherit a function that has the same as the class.

Programmer Response: Change the name of either the base class or the inherited function.

EQA2210E Forward declaration of the enumeration "&1" is not allowed.

Explanation: The declaration of an enumeration must contain its member list.

Programmer Response: Fully declare the enumeration.

EQA2211E The previous message applies to argument &1 of function "&2".

Problem Determination: (*where* &1 is the argument number &2 is the function name and type)

Explanation: The previous message applies to the specified argument number. This message does not indicate another error or warning, it indicates which argument of the function call is the subject of the previous message.

EQA2212E The nested class object "&1" needs a constructor so that its &2 members can be initialized.

Problem Determination: (*where* &1 is the nested class name &2 is the word const or reference)

Programmer Response: Create a constructor for the nested class object.

EQA2213E The integer constant is out of range.

Explanation: You have provided an integer constant that is out of range. For the range of integer constants check limits.h.

Programmer Response: Ensure the integer constant is in range.

EQA2214E The floating point constant is out of range.

Explanation: You have provided a floating point constant that is out of range. For the range of floating point constants check float.h.

Programmer Response: Ensure the floating point constant is in range.

EQA2215E The &1 member "&2" must be initialized in the constructor's initializer list.

Problem Determination: (*where* &1 is the word const or reference &2 is the member name)

Explanation: Using the constructor's member initializer list is the only way to initialize nonstatic const and reference members.

Programmer Response: Initialize the member in the constructor's initializer list.

EQA2216E Constructors and conversion functions are not considered when resolving an explicit cast to a reference type.

Explanation: You cannot resolve an explicit cast to a reference type using constructors or conversion functions.

Programmer Response: Cast the type to a temporary type and then take the reference to it.

EQA2217E A character string literal cannot be concatenated with a wide string literal.

Explanation: A string that has a prefix L cannot be concatenated with a string that is not prefixed.

Programmer Response: Ensure both strings have the same prefix, or no prefix at all.

EQA2218E All members of type "&1" must be explicitly initialized with all default arguments specified.

Problem Determination: (*where* &1 is a class name &2 is the member name)

Explanation: Default arguments for member functions are not checked until the end of the class definition. Default arguments for member functions of nested classes are not semantically checked until the containing class is defined. A call to a member function must specify all of the arguments before the default arguments have been checked.

Programmer Response: Specify all default arguments with all members of the type.

EQA2219E The address of an overloaded function can be taken only in an initialization or an assignment.

Programmer Response: Ensure the address of an overloaded function is used on an initialization or an assignment, or remove the expression.

EQA2220E Syntax error - found "&1 &2" : "&1" is not a type name.

Problem Determination: (*where* &1 is a token &2 is a token)

Explanation: The debugger detected a nontype symbol where a type is required. A type must be used to declare an object.

Programmer Response: Change to a type name or remove the expression.

EQA2221E A temporary of type "&1" is needed: "&2" is an abstract class.

Explanation: The debugger has determined that it must use a temporary to store the result of the expression, but the result is an abstract base type. An abstract base type cannot be used to create an object.

Programmer Response: Change the type of the result.

EQA2222E "&1" hides pure virtual function "&2" in the nonvirtual base "&3".

Problem Determination: (*where* &1 is the derived member's name &2 is the name of the pure virtual function &3 is the name of the class that contains the pure virtual)

Explanation: The pure virtual function in a nonvirtual base cannot be overridden once it has been hidden.

Programmer Response: Make the pure virtual function visible, or make the base it is derived from virtual.

EQA2223E The class qualifier "&1" for "&2" must be a template class that uses the template arguments.

Problem Determination: (*where* &1 is a (possibly qualified) class name. &2 is a C++ name.)

Explanation: A nonclass template can only declare a global function or a member of a template class. If it declares a member of a template class, the template class arguments must include at least one of the nonclass template arguments.

Programmer Response: Change the template declaration so that it either declares a global function or a member of a template class that uses the nonclass template arguments.

EQA2224E The class "&1" cannot be passed by value because it does not have a copy constructor.

Problem Determination: (*where* &1 is a class name)

Explanation: The debugger needs to generate a temporary to hold the return value of the function. To generate the temporary object, a copy constructor is

needed to copy the contents of the object being returned into the temporary object.

Programmer Response: Create a copy constructor for the class or change the argument to pass by value.

EQA2225E "&1" cannot have an initializer list.

Problem Determination: (*where* &1 is a function name)

Explanation: A member function that is not a constructor is defined with an initializer list.

Programmer Response: Remove the initializer list.

EQA2226E Return value of type "&1" is expected.

Problem Determination: (*where* &1 is a C/C++ type)

Explanation: No return value is returned from the current function but the function is expecting a nonvoid return value.

Programmer Response: Ensure a value is returned, or change the return type of the function to void.

EQA2227E "&1" bypasses initialization of "&2".

Problem Determination: (*where* &1 is one of the keywords default, case &2 is the variable being initialized)

Explanation: It is invalid to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is also jumped past.

Programmer Response: Enclose the initialization in a block statement.

EQA2228E "&1" is being redeclared as a member function. It was originally declared as a data member.

Problem Determination: (*where* &1 is a variable name)

Explanation: The template redeclares a data member of a class template as a member function.

Programmer Response: Change the original declaration of the variable to a member function, or change the redeclaration of the variable to a data member.

EQA2229E "&1" is being redeclared as a nonfunction member or has syntax errors in its argument list.

Problem Determination: (*where* &1 is a variable name)

Explanation: The template redeclares a member function of a class template as a data member. There might be syntax errors in the declaration.

Programmer Response: Change one of the declarations, if necessary.

EQA2230E A string literal cannot be longer than &1 characters.

Problem Determination: (*where* &1 is a number. This number is system dependent.)

Explanation: The debugger limit for the length of a string literal has been exceeded. The string literal is too long for the debugger to handle.

Programmer Response: Specify a shorter string literal.

EQA2231E A wide string literal cannot be longer than &1 characters.

Problem Determination: (*where* &1 is a number. This number is system dependent.)

Explanation: The debugger limit for the length of a wide string literal has been exceeded. The wide string literal is too long for the debugger to handle.

Programmer Response: Specify a shorter string literal.

EQA2232E Invalid "multibyte character sequence character" (MBCS) character.

Explanation: The debugger has detected a multibyte character sequence that it does not recognize.

Programmer Response: Replace the "multibyte character sequence character" (MBCS) character.

EQA2233E "&1" is an undefined pure virtual function.

Explanation: The user tried to call a member function that was declared to be a pure virtual function.

Programmer Response: Remove or define the function as pure virtual.

EQA2234E Template "&1" cannot be instantiated because the actual argument for formal argument "&2" has more than one variant.

Problem Determination: (*where* &1 is the name of a function template. &2 is the name of a formal template argument.)

Explanation: The argument is a function template or an overloaded function with two or more variants. The debugger cannot decide which variant to choose to bind to the argument type.

Programmer Response: Change the formal template argument or remove the extra variants.

EQA2235E Pointer to a built-in function not allowed.

Explanation: Because you cannot take the address of a built-in function, you cannot declare a pointer to a built-in function.

Programmer Response: Remove the pointer.

EQA2236E Built-in function "&1" not recognized.

Problem Determination: (*where* &1 is the name of a function.)

Explanation: The function declared as a built-in is not recognized by the debugger as being a built-in function.

Programmer Response: Ensure the function is a built-in function or remove the built-in keyword from the declaration.

EQA2237E "&1" is not supported.

Problem Determination: (*where* &1 is a C++ operator)

Programmer Response: Remove the operator from the expression.

EQA2238E Function calls are not supported.

Explanation: You can only generate this message in the debugger, when you use an expression that includes a function call.

Programmer Response: Remove function calls from the expression.

EQA2239E The expression is too complicated.

Programmer Response: Simplify the expression.

EQA2240E Evaluation of the expression requires a temporary.

Programmer Response: Change the expression so that a temporary object is not required.

EQA2241E "&1" is an overloaded function.

Problem Determination: (*where* &1 is the name of a function.)

Explanation: The identifier refers to an overloaded function with two or more variants. The debugger requires a prototype argument list to decide which variant to process.

Programmer Response: Specify a prototype argument list or remove variants of the overloaded function.

EQA2242E The bit-field length must not be negative.

Explanation: The bit-field length must be a nonnegative integer value.

Programmer Response: Change the bit-field length to a nonnegative integer value.

EQA2243E A zero-length bit-field must not have a name.

Explanation: A named bit-field must have a positive length; a zero-length bit-field is used for alignment only, and must not be named.

Programmer Response: Remove the name from the zero-length bit-field.

EQA2244E The bit-field is too small; &1 bits are needed for "&2".

Problem Determination: (*where* &2 is a C++ name)

Explanation: The bit-field length is smaller than the number of bits needed to hold all values of the enum.

Programmer Response: Increase the bit-field length.

EQA2245E The bit-field is larger than necessary; only &1 bits are needed for "&2".

Problem Determination: (*where* &2 is a C++ name)

Explanation: The bit-field length is larger than the number of bits needed to hold all values of the enum.

Programmer Response: Decrease the bit-field length.

EQA2246E A template friend declaration can only declare, not define, a class or function.

Explanation: The class or function declared in the template friend declaration must be defined at file scope.

Programmer Response: Remove the definition from the template friend declaration.

EQA2247E The function "&1" must not be declared "&2" at block scope.

Problem Determination: (*where* &2 is a C++ keyword.)

Explanation: There can be no static or inline function declarations at block scope.

Programmer Response: Move the function so that it is not defined at block scope.

EQA2248E The previous &1 messages apply to function argument &2.

Problem Determination: (*where* &1 is an integer corresponding to the function argument number)

Explanation: The previous message applies to the specified argument number. This message does not indicate another error or warning, it indicates which argument of the function call is the subject of the previous message.

EQA2249E The previous &1 messages apply to argument &2 of function "&3".

Problem Determination: (*where* &1 is the number of messages &2 is the argument number &3 is the function name and type)

Explanation: The previous message applies to the specified argument number. This message does not indicate another error or warning, it indicates which argument of the function call is the subject of the previous message.

EQA2250E "&1" is not a static member of "&2".

Problem Determination: (*where* &2 is a class name.)

Explanation: Nonstatic data members cannot be defined outside the class definition.

Programmer Response: Make the member a static member or move it into the class definition.

EQA2251E The initializer must be enclosed in braces.

Explanation: Array element initializers must be enclosed in braces.

Programmer Response: Put braces around the initializer.

EQA2252E union "&1" has multiple initializers associated with its constructor "&2".

Explanation: A union can only contain one member object at any time, and therefore can be initialized to only one value.

Programmer Response: Remove all but one of the initializers.

EQA2253E You cannot override virtual function "&1" because "&3" is an ambiguous base class of "&2".

Problem Determination: (*where* &3 is the class name of an ambiguous base of &2)

Explanation: The debugger must generate code to convert the actual return type into the type that the overridden function returns (so that calls to the original

overridden function is supported). However, the conversion is ambiguous.

Programmer Response: Clarify the base class.

EQA2254E "&1" is not initialized until after the base class is initialized.

Problem Determination: (*where* &1 is the class member referenced in the base class initializer.)

Explanation: First, the base classes are initialized in declaration order, then the members are initialized in declaration order, then the body of the constructor is executed.

Programmer Response: Do not reference the class member in the base class initializer.

EQA2255E The expression to the left of the "&1" operator is a relational expression ("&2"). The "&3" operator might have been intended.

Problem Determination: (*where* &1 is the bitwise operator | or &. &2 is one of the relational operators. &3 is either the operator || or the operator &&.)

Explanation: The debugger has detected the mixing of relational and bitwise operators in what was determined to be a conditional expression.

Programmer Response: Ensure the correct operator is being used.

EQA2256E The expression to the left of the "&1" operator is a logical expression ("&2"). The "&3" operator may have been intended.

Problem Determination: (*where* &1 is the bitwise operator | or &. &2 is one of the relational operators. &3 is either the operator || or the operator &&.)

Explanation: The debugger has detected the mixing of relational and bitwise operators in what was determined to be a conditional expression.

Programmer Response: Ensure the correct operator is being used.

EQA2257E The expression to the left of the "&1" operator is an equality expression ("&2"). The "&3" operator may have been intended.

Problem Determination: (*where* &1 is the bitwise operator | or &. &2 is one of the relational operators. &3 is either the operator || or the operator &&.)

Explanation: The debugger has detected the mixing of relational and bitwise operators in what was determined to be a conditional expression.

Programmer Response: Ensure the correct operator is being used.

EQA2258E The expression to the right of the "&1" operator is a relational expression ("&2"). The "&3" operator may have been intended.

Problem Determination: (*where* &1 is the bitwise operator | or &. &2 is one of the relational operators. &3 is either the operator || or the operator &&.)

Explanation: This message is generated by the /Wcnd option. This option warns of possible redundancies or problems in conditional expressions involving relational expressions and bitwise operators.

Programmer Response: Ensure the correct operator is being used.

EQA2259E Assignment to the "this" pointer is not allowed.

Explanation: The "this" pointer is a const pointer and cannot be modified.

Programmer Response: Remove the assignment to the "this" pointer.

EQA2260E "&1" must not have any arguments.

Problem Determination: (*where* &1 is a special member function.)

Programmer Response: Remove all arguments from the special member function.

EQA2261E The second operand to the "offsetof" operator is not valid.

Explanation: The second operand to the "offsetof" operator must consist only of "." operators and "[]" operators with constant bounds.

Programmer Response: Remove or change the second operand.

EQA2262E "&1" is a member of "&2" and cannot be used without qualification.

Problem Determination: (*where* &2 is a possibly qualified class name)

Explanation: The specified name is a class member, but no class qualification has been used to reference it.

Programmer Response: Use the scope operator (::) to qualify the name.

EQA2263E `sdq.&1` is undefined. Every variable of type `&2` will assume `&1` has no virtual bases and no multiple inheritance.

Problem Determination: (*where* `&2` is a pointer to member type)

Explanation: The definition of the class is not given but the debugger must implement the pointer to member. It will do so by assuming the class has at most one nonvirtual base class.

Programmer Response: If this assumption is incorrect, define the class before declaring the member pointer.

EQA2264E `&1` is undefined. The delete operator will not call a destructor.

Problem Determination: (*where* `&1` is a name of a class, struct, or union)

Explanation: The definition of the class is not given so the debugger does not know whether the class has a destructor. No destructors will be called.

Programmer Response: Define the class.

EQA2265E Label `&1` is undefined.

Problem Determination: (*where* `&1` is a C++ name)

Explanation: The specified label is used but is not defined.

Programmer Response: Define the label before using it.

EQA2266E The initializer for enumerator `&1` must be an integral constant expression.

Problem Determination: (*where* `&1` is an enumerator name)

Explanation: The value of an enumerator must be a constant expression that is promotable to a signed int value. A constant expression has a value that can be determined during compilation and does not change during program execution.

Programmer Response: Change the initializer to an integral constant expression.

EQA2267E Overriding virtual function `&1` may not return `&2` because class `&3` has multiple base classes or a virtual base class.

Problem Determination: (*where* `&1` is the name of a virtual function `&2` is an abstract declarator `&3` is the class being returned)

Explanation: Contravariant virtual functions are sup-

ported only for classes with single inheritance and no virtual bases.

Programmer Response: Ensure the class has single inheritance and no virtual bases.

EQA2268E Virtual function `&1` is not a valid virtual function override because `&3` is an inaccessible base class of `&2`.

Problem Determination: (*where* `&3` is the class name of an inaccessible base of `&2`)

Explanation: The debugger must generate code to convert the actual return type into the type that the overridden function returns (so that calls to the original overridden function is supported). However, the target type is inaccessible to the overriding function.

Programmer Response: Make the base class accessible.

EQA2269E `&1` is a member of `&2` classes. To reference one of these members, `&3` must be qualified.

Problem Determination: (*where* `&1` is a C++ member name `&2` is an integer greater than 1 `&3` is a C++ member name)

Explanation: The class member specified is defined in more than one class nested within the base class and cannot be referenced from the base class if it is not qualified. This message is generated by the `/Wund` option.

Programmer Response: Use the scope operator (`::`) to qualify the name.

EQA2270E `&1` is not the name of a function.

Problem Determination: (*where* `&1` is a name)

Explanation: A function name is required in this context. The specified name has been declared but it is not the name of a function.

Programmer Response: Ensure the name is the correctly spelled name of a function.

EQA2271E Enum type `&1` cannot contain both negative and unsigned values.

Explanation: The enumerator type values should fit into an integer. Specifying both unsigned and negative values will exceed this limit.

Programmer Response: Remove the negative or unsigned values.

EQA2272E Cannot take the address of the machine-coded function "&1".

Explanation: Because the function is machine-coded, you cannot take its address.

Programmer Response: Remove the reference to that function.

EQA2273E An initializer is not allowed for the non-virtual function "&1".

Problem Determination: (*where* &1 is a function name)

Explanation: The declaration of a pure virtual function must include the keyword `virtual`.

Programmer Response: Remove the initializer.

EQA2274E A local variable or debugger temporary is being used to initialize reference member "&1".

Explanation: The local variable is only active until the end of the function, but it is being used to initialize a member reference variable.

Programmer Response: Ensure that no part of your program depends on the variable or temporary.

EQA2275E "&1" is not the SOM name of a SOM class.

Explanation: A SOM name that represents a SOM class is expected, and was not found. The SOM name of a class might differ from its C++ name.

Programmer Response: Ensure that you use the correct SOM name for the class.

EQA2276E Definition of "&1" is only allowed at file scope.

Problem Determination: (*where* &1 is a C++ template class type)

Explanation: A template class is being defined in a scope other than file scope. Because all template class names have file scope this definition is not allowed.

Programmer Response: Move the template class definition to file scope.

EQA2277E Class template "&1" cannot be used until its containing template has been instantiated.

Problem Determination: (*where* &1 is a C++ class template type)

Explanation: The class template referenced cannot be used until the template that contains it has been instantiated.

Programmer Response: Declare the class template at file scope or instantiate the template that contains it.

EQA2278E Invalid `wchar_t` value &1.

Problem Determination: (*where* &1 is the value which is not valid)

Explanation: A multibyte character or escape sequence in a literal has been converted to an invalid value for type `wchar_t`.

Programmer Response: Change the character or escape sequence.

EQA2279E The string must be terminated before the end of the line.

Explanation: The debugger detected a string that was not terminated before an end-of-line character was found.

Programmer Response: End the string or use `"\"` to continue the string on the next line. The `"\"` must be the last character on the line.

EQA2280E A character constant must end before the end of the line.

Explanation: The debugger detected a character constant that was not terminated before an end-of-line character was found.

Programmer Response: End the character constant or use `"\"` to continue it on the next line. The `"\"` must be the last character on the line.

EQA2281E A matching &1 function named "&2" could not be found.

Problem Determination: (*where* &1 is one of 'const', 'volatile' or 'const volatile'. &2 is the name of the called function (without the argument list).)

Explanation: The call might have failed because no member function exists that accepts the 'const/volatile' qualifications of the object.

Programmer Response: Ensure the type qualifier is correct and that the function name is spelled correctly.

EQA2282E "&1" is a type name being used where a variable name is expected.

Problem Determination: (*where* &1 is a C/C++ name)

Explanation: The identifier must be a variable name not a type name.

Programmer Response: Check that the identifier is a variable name and ensure the variable is not hidden by a type name.

EQA2283E Template "&1" has a missing or incorrect template argument list.

Problem Determination: (*where* &1 is a C++ name)

Explanation: A template name was found where a variable name was expected.

Programmer Response: Complete the template argument list or change the identifier to a variable name.

EQA2284E Template friend declaration does not declare a class or a function.

Explanation: A template friend declaration must declare a class or a function following the template arguments.

Programmer Response: Change the template declaration to declare a class or a function.

EQA2285E The 'const' object has been cast to a non-'const' object.

Explanation: A cast has been used to possibly modify a 'const' object. This might cause undefined behaviour at run time.

Programmer Response: Remove the cast or make the object nonconst.

EQA2286E Global friend functions may not be defined in a local class.

Explanation: A local class cannot have a friend function.

Programmer Response: Make the function a member function in the local class.

EQA2287E The address of data member "&1" cannot be taken because the member is being referenced through a `_get_` function.

Explanation: An attribute is accessed through a `"_get_"` method if its backing data is not accessible, or if the `SOMNoDataDirect` pragma is in effect for the class. Since the `"__get"` method returns the value of the member, and not its address, it isn't possible to use the address operator `"&"` on the member to create an ordinary pointer. This error can also be generated if you haven't used the `"&"` operator explicitly, but the debugger needs to use it to implement your code. You can create a pointer-to-member that refers to an attribute.

Programmer Response: Rewrite the expression that causes the address to be taken, or remove the `SOMAttribute` pragma.

EQA2288E '!' was specified for "&1", which was introduced in the current class.

Problem Determination: (*where* &1 is a C++ member name.)

Explanation: `!` must only be used for names introduced in a base class.

Programmer Response: Remove the `!` from the `SOMReleaseOrder` entry.

EQA2289E Function linkage differs from that of overridden function "&1".

Explanation: The linkage of a virtual function must agree with the linkage of base class member functions that it overrides.

Programmer Response: Change the linkage keyword to agree with the base class method.

EQA2290E The physical size of a struct or union is too large.

Explanation: The debugger cannot handle any size which is too large to be represented internally.

Programmer Response: Reduce the size of the struct or union members.

EQA2291E The "&1" qualifier is not supported on the target platform.

Explanation: A qualifier has been specified on a platform that does not support it.

Programmer Response: Remove the qualifier.

EQA2292E The array bound is too large.

Explanation: The array bound should be a value less than or equal to `max int`.

Programmer Response: Reduce the number of elements in the array.

EQA2293E "&1" was not specified in the previous declaration of "&2".

Problem Determination: (*where* &1 is an attribute. &2 is a name.)

Explanation: An attribute has been specified that conflicts with the previous declaration of a name.

Programmer Response: Remove the attribute.

EQA2500E Incorrect or missing data

Explanation: The data at the cursor location is either incorrect or some data is missing. There could be several reasons for this:

1. Invalid combination of options specified.
2. Invalid data for field.
3. Data not entered, when required by options given.
4. Quotes specified when not allowed.

Programmer Response: Correct the entry where the cursor is positioned and invoke the function again. You can use Help (PF1) to find the context sensitive help for that field.

EQA2501E DTCN internal error

Explanation: DTCN discovered internal error.

Programmer Response: Contact IBM service.

EQA2502E Internal CICS error

Explanation: During processing, DTCN discovered 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.

EQA2503E Key Not Defined.

Explanation: There is no action defined with the PF key used by the user.

Programmer Response: Use the keys displayed in the bottom line. For more information about the actions defined for this panel, use PF2 key for general help.

EQA2504E 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.

EQA2505E 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.

EQA2506E 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.

EQA2507E 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.

EQA2508E Next failed - profile does not exist

Explanation: There are no more profiles in the DT/370 Profile Repository.

EQA2510I DTCN closed

Explanation: DTCN deleted all profiles stored in the Debug Tool Profiles Repository. This action affects all users working with that CICS region.

EQA2511E 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 regionwide) or both.

EQA2512E TCP/IP for CICS is not active

Explanation: You have tried to set up a debug session using TCP/IP, but TCP/IP for CICS is not active in the CICS region.

Programmer Response: Either set up a non-TCP/IP session, or refer to the TCP/IP for CICS publications for guidance on activating it.

EQA2514I Debug Tool profile added

Explanation: A new profile was added to the Debug Tool Profile Repository

EQA2515I Debug Tool profile replaced

Explanation: Existing profile was updated in the Debug Tool Profile Repository.

EQA2516I Debug Tool profile deleted

Explanation: Existing profile was deleted from the Debug Tool Profile Repository

Bibliography

High level language publications

OS/390 C/C++

OS/390 C/C++ Compiler and Run-Time Migration Guide, SC09-2359

OS/390 C/C++ IBM Open Class Library Reference, SC09-2364

OS/390 C/C++ IBM Open Class Library User's Guide, SC09-2363

OS/390 C/C++ Database Access Class Library Utility Guide, SC09-2412

OS/390 C/C++ Language Reference, SC09-2360

OS/390 C/C++ Programming Guide, SC09-2362

OS/390 C/C++ Reference Summary, SX09-1313

OS/390 C/C++ User's Guide, SC09-2361

OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference, SC09-2366

OS/390 C/C++ Run-Time Library Reference, SC28-1663

OS/390 C Curses, SC28-1907

IBM COBOL for OS/390 & VM

Licensed Program Specifications, GC26-9044

Installation and Customization under MVS, GC26-9045

Language Reference, SC26-9046

Diagnosis Guide, GC26-9047

Fact Sheet, GC26-9048

Programming Guide, SC26-9049

Compiler and Run-Time Migration Guide, GC26-4764

IBM COBOL for MVS & VM

Programming Guide, SC26-4767

Language Reference, SC26-4769

Licensed Program Specifications, GC26-4761

Compiler and Run-Time Migration Guide, GC26-4764

Installation and Customization under MVS, GC26-4766

Diagnosis Guide, SC26-3138

PL/I for MVS & VM

Programming Guide, SC26-3113

Language Reference, SC26-3114

Licensed Program Specifications, GC26-3116

Compiler and Run-Time Migration Guide, SC26-3118

Installation and Customization under MVS, SC26-3119

Diagnosis Guide, SC26-3149

Compile-Time Messages and Codes, SC26-3229

Reference Summary, SX26-3821

Related publications

OS/390 Language Environment

OS/390 Language Environment Debugging Guide and Run-Time Messages, SC28-1942

OS/390 Language Environment Programming Guide, SC28-1939

OS/390 Language Environment Programming Reference, SC28-1940

OS/390 Language Environment Writing Interlanguage Applications, SC28-1943

OS/390 Language Environment Customization, SC28-1941

OS/390 Language Environment Run-Time Migration Guide, SC28-1944

OS/390 Language Environment Concepts Guide, GC28-1945

MVS/ESA

JCL User's Guide, GC28-1653

JCL Reference, GC28-1654

MVS/Enterprise Systems Architecture Application Development Guide, GC28-1821

MVS/Enterprise Systems Architecture System Commands, GC28-1826

VM/ESA

Virtual Machine/Enterprise Systems Architecture Operating System CP System Command Reference, SC24-5434

TSO

MVS/Enterprise Systems Architecture TSO Programming, GC28-1565

CICS

CICS Application Programming Primer (VS COBOL II), SC33-0674

CICS/ESA Application Programming Reference, SC33-1170

CICS/ESA Application Programming Guide, SC33-1169

IMS

IMS/ESA Application Programming, SC26-4274

IMS/VS Application Programming, SH20-9026

IMS/VS Batch Terminal Simulator Program Reference and Operations Manual, SH20-5523

DB2 for MVS

Master Index, GC26-3271

Licensed Program Specifications, GC26-3272

Administration Guide, SC26-3265

Application Programming and SQL Guide, SC26-3266

Command Reference, SC26-3267

Installation Guide, SC26-3456

Messages and Codes, SC26-3268

Data Sharing: Planning and Administration, SC26-3269

Release Guide, SC26-3394

SQL Reference, SC26-3270

Reference for Remote DRDA Requesters and Servers, SC26-3282

Reference Summary, SX26-3829

Utilities Guide and Reference, SC26-3395

Diagnosis Guide and Reference, LY27-9618

Diagnostic Quick Reference Card, LY27-9622

DB2 and SQL/DS

IBM DATABASE 2 Application Programming Guide, SC26-4293

IBM DATABASE 2 Administration Guide, SC26-4374

IBM DATABASE 2 SQL Reference, SC26-4346

Softcopy publications

BookManager Read and Build General Information, GC23-0447

BookManager Read/VM: Getting Started and Command Summary, SC23-0448

BookManager Read/VM: Displaying On-line Books, SC23-0449

BookManager Read/MVS: Getting Started and Command Summary, SC38-2033

BookManager Read/MVS: Displaying On-line Books, SC38-2034

Glossary

A

active block. The currently executing block that invokes Debug Tool or any of the blocks in the CALL chain that leads up to this one.

active server. A server that is being used by a CODE/370 session. Contrast with *inactive server*. See also *server*.

alias. An alternative name for a field used in some high-level programming languages.

animation. The execution of instructions one at a time with a delay between each so that any results of an instruction can be viewed.

attention interrupt. An I/O interrupt caused by a terminal or workstation user pressing an attention key, or its equivalent.

attention key. A function key on terminals or workstations that, when pressed, causes an I/O interrupt in the processing unit.

attribute. A characteristic or trait the user can specify.

Autosave. A choice allowing the user to automatically save work at regular intervals.

B

batch. Pertaining to a predefined series of actions performed with little or no interaction between the user and the system. Contrast with *interactive*.

batch job. A job submitted for batch processing. See *batch*. Contrast with *interactive*.

batch mode. An interface mode for use with the MFI Debug Tool 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.

C

century window (COBOL). The 100-year interval in which COBOL assumes all windowed years lie. The start of the COBOL century window is defined by the COBOL YEARWINDOW compiler option.

Change. A push button that changes a value associated with an entry or entries in a list to another value specified by the user.

CODE/370. The IBM product formally called the CoOperative Development Environment/370, an application development and maintenance facility for editing, compiling, and debugging third-generation programming languages.

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 Programming Interface (CPI). Definitions of those application development languages and services that have (or are intended to have) implementations on and a high degree of commonality across SAA environments. See also *Systems Application Architecture*.

compile. To translate a program written in a high level language into a machine-language program.

compile unit. A sequence of HLL statements that make a portion of a program complete enough to compile correctly. Each HLL product has different rules for what comprises a compile unit.

compiler. A program that translates instructions written in a high level programming language into machine language.

condition. Any synchronous event that might need to be brought to the attention of an executing program or the language routines supporting that program. Conditions fall into two major categories: conditions detected by the hardware or operating system, which result in an interrupt; and conditions defined by the programming language and detected by language-specific generated code or language library code. See also *exception*.

container. A system object that contains and organizes source files. For example, a VM minidisk, or an MVS partitioned data set.

conversational. A transaction type which accepts input from the user, performs a task, then returns to get more input from the user.

CPI. See *Common Programming Interface*.

currently-qualified. See *qualification*.

D

data type. A characteristic that determines the kind of value that a field can assume.

data set. The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

date field. A COBOL data item that can be any of the following:

- A data item whose data description entry includes a DATE FORMAT clause.
- A value returned by one of the following intrinsic functions:

DATE-OF-INTEGER
DATE-TO-YYYYMMDD
DATEVAL
DAY-OF-INTEGER
DAY-TO-YYYYDDD
YEAR-TO-YYYY
YEARWINDOW

- The conceptual data items DATE and DAY in the ACCEPT FROM DATE and ACCEPT FROM DAY statements, respectively.
- The result of certain arithmetic operations.

The term date field refers to both *expanded date field* and *windowed date field*. See also *nondate*.

date processing statement. A COBOL statement that references a date field, or an EVALUATE or SEARCH statement WHEN phrase that references a date field.

DBCS. See *double-byte character set*.

debug. To detect, diagnose, and eliminate errors in programs.

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*.

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously. See also *load module*.

E

enclave. An independent collection of routines in Language Environment, one of which is designated as the MAIN program. The enclave contains at least one thread and is roughly analogous to a program or routine. See also *thread*.

entry point. The address or label of the first instruction executed on entering a computer program, routine, or subroutine. A computer program can have a number of different entry points, each perhaps corresponding to a different function or purpose.

exception. An abnormal situation in the execution of a program 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*.

expanded date field. A COBOL date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year. In COBOL, four digits representing a year, including the century (for example, 1998). Appears in expanded date fields. Compare with *windowed year*.

expression. A group of constants or variables separated by operators that yields a single value. An expression can be arithmetic, relational, logical, or a character string.

F

file. (1) A named set of records stored or processed as a unit. (2) An element included in a container: for example, a VM file, an MVS member or partitioned data set. See *container*. See also *data set*.

frequency count. A count of the number of times statements in the currently qualified program unit have been run.

full-screen mode. An interface mode for use with a nonprogrammable terminal which displays a variety of information about the program you are debugging.

H

high level language (HLL). A programming language such as C, COBOL, or PL/I.

HLL. See *high level language*.

hook. An instruction inserted into a program by a compiler at compile-time. Using a hook, you can set break-points to instruct Debug Tool to gain control of the program at selected points during its execution.

I

IBM SAA AD/Cycle Language Environment/370.
See *Language Environment*.

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*.

interactive. Pertaining to a program or system that alternately accepts input and then responds. An interactive system is conversational; that is, a continuous dialog exists between the user and the system. Contrast with *batch*.

I/O. Input/output.

L

Language Environment. An IBM software product that provides a common run-time environment and common run-time services for IBM high level language compilers.

library routine. A routine maintained in a program library.

line mode. An interface mode for use with a nonprogrammable terminal which uses a single command line to accept Debug Tool commands.

line wrap. The function that automatically moves the display of a character string (separated from the rest of a line by a blank) to a new line if it would otherwise overrun the right margin setting.

link-edit. To create a loadable computer program using a linkage editor.

linkage editor. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single relocatable load module.

listing. A printout that lists the source language statements of a program with all preprocessor statements, includes, and macros expanded.

load module. A program in a form suitable for loading into main storage for execution. In this document this term is also used to refer to a Dynamic Load Library (DLL).

M

multitasking. A mode of operation that enables the concurrent performance, or interleaved execution, of two or more tasks.

N

nonconversational. A transaction type which accepts input, performs a task, and then ends.

nondate. A COBOL data item that can be any of the following:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A reference modification of a date field
- The result of certain arithmetic operations that may include date field operands; for example, the difference between two compatible date fields.

The value of a nondate may or may not represent a date.

O

Options. A choice that lets the user customize objects or parts of objects in an application.

P

panel. In Debug Tool, an area of the screen used to display a specific type of information.

parameter. Data passed between programs or procedures.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

path point. A point in the program where control is about to be transferred to another location or a point in the program where control has just been given.

PDS. See *partitioned data set*.

prefix area. The eight columns to the left of the program source or listing containing line numbers. Statement breakpoints can be set in the prefix area.

primary entry point. See *entry point*.

procedure. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements. For example, a VM exec, or an MVS CLIST.

process. The highest level of the Language Environment program management model. It is a collection of resources, both program code and data, and consists of at least one enclave.

profile. A group of customizable settings that govern how the user's session appears and operates.

Profile. A choice that allows the user to change some characteristics of the working environment, such as the pace of statement execution in the Debug Tool.

program. A sequence of instructions suitable for processing by a computer. Processing can include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it.

program unit. See *compile unit*.

program variable. A predefined variable that exists when Debug Tool was invoked.

pseudo-conversational transaction. The result of a technique in CICS called pseudo-conversational processing in which a series of nonconversational transactions gives the appearance (to the user) of a single conversational transaction. See *conversational* and *nonconversational*.

Q

qualification. A method used to specify to what procedure or load module a particular variable name, function name, label, or statement id belongs. The SETT 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.

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

SAA. See *Systems Application Architecture*.

SAA language. Any of the high level languages included in IBM's Systems Application Architecture. This includes the following IBM programming languages: C/370, PL/I 370, and COBOL/370.

Glossary

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 VM file, or an MVS member or partitioned data set.

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.

Source window. A Debug Tool window that contains a display of either the source code or the listing of the program being debugged.

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.

Step. A choice the user can make to execute one or more statements in the application being debugged.

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.

subroutine. (1) A sequenced set of instructions or statements that can be used in one or more computer programs at one or more points in a computer program.

suffix area. A variable-sized column to the right of the program source or listing statements, containing frequency counts for the first statement or verb on each line. Debug Tool optionally displays the suffix area in the Source window. See also *prefix area*.

syntactic analysis. An analysis of a program done by a compiler to determine the structure of the program and the construction of its source statements to deter-

mine whether it is valid for a given programming language. See also *syntax checker*, *syntax error*.

syntax. The rules governing the structure of a programming language and the construction of a statement in a programming language.

syntax error. Any deviation from the grammar (rules) of a given programming language appearing when a compiler performs a syntactic analysis of a source program. See also *syntactic analysis*.

Systems Application Architecture (SAA). A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

T

temporary variable. See *session variable*.

thread. The basic line of execution within the Language Environment program model. It is dispatched with its own instruction counter and registers by the system. Threads can execute, concurrently with other threads. The thread is where actual code resides. It is synonymous with a CICS transaction or task. See also *enclave*.

thread id. A small positive number assigned by Debug Tool to an Language Environment task.

token. A character string in a specific format that has some defined significance in a programming language.

trigraph. A group of three characters which, taken together, are equivalent to a single special character.

U

utility. A computer program in general support of computer processes; for example, a diagnostic program, a trace program, or a sort program.

V

variable. A name used to represent a data item whose value can be changed while the program is running.

W

windowed date field. A COBOL date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year. In COBOL, two digits representing a year within a century window (for example, 98).

| Appears in windowed date fields. See also *century*
| *window (COBOL)*.

| Compare with *expanded year*.

word wrap. See *line wrap*.

Index

A

abbreviating commands 92
 abbreviating keywords 194
 abnormal end of application, setting breakpoint at 133
 accessing PL/I program variables 189
 active block, definition of 424
 active server, definition of 424
 %ADDRESS
 description of 128
 for C/C++ 142
 for COBOL 177
 alias, definition of 424
 ALLOCATE, AT command (PL/I), syntax 209
 allowable comparisons for Debug Tool IF command 350
 allowable moves for Debug Tool MOVE command 352
 allowable moves for Debug Tool SET command 354
 alternative methods of input under IMS 115
 %AMODE
 description of 128
 for C/C++ 142
 for COBOL 177
 ANALYZE command (PL/I), syntax 205
 animation, definition of 424
 APPEARANCE, AT command, syntax 210
 assigning values to variables 140, 172
 Assignment command (PL/I), syntax 206
 AT ALLOCATE command (PL/I), syntax 209
 AT APPEARANCE command, syntax 210
 AT CALL breakpoints, for C++ 164
 AT CALL command, syntax 212
 AT CHANGE command, syntax 214
 AT CURSOR command (Full-Screen Mode), syntax 217
 AT DATE command (COBOL), syntax 218
 AT DELETE command, syntax 218
 AT ENTRY command, syntax 219
 AT ENTRY/EXIT breakpoints
 for C++ 163
 AT EXIT command, syntax 219
 AT GLOBAL command, syntax 220
 AT LABEL command, syntax 222
 AT LINE, using 107
 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 commands 207, 232
 summary table 207

attention interrupt
 definition of 424
 effect of during interactive sessions 135
 how to initiate 135
 in Debug Tool 135
 required Language Environment run-time options 135
 attention key, definition of 424
 attribute variables
 for C/C++ 249
 for PL/I 254
 attribute, definition of 424
 attributes of variables 125
 Autosave, definition of 424

B

basic tasks of Debug Tool 41
 batch job, definition of 424
 batch mode
 debugging DB2 programs in 111
 debugging IMS programs in 115
 batch mode (CICS) 116
 Batch Terminal Simulator (BTS) Full-Screen Image Support (FSS) 113, 114
 batch, definition of 424
 BEGIN command (PL/I), syntax 232
 bind requirements for DB2 111
 blanks, significance of 196
 block
 definition of 424
 description of 128
 for C/C++ 142
 for COBOL 177
 size 3
 using, for C 156
 block command (C/C++), syntax 233
 block_name, description of 199
 block_spec, description of 199
 break command (C/C++), syntax 233
 breakpoint
 definition of 424
 in unknown compile unit 211
 removing 240
 using within multiple enclaves 104
 BTS full-screen image support (FSS) 113, 114
 built-in functions 135
 Debug Tool, using with C/C++ 148
 for PL/I 191
 generation 137
 %HEX 148
 instances 136, 148

built-in functions (*continued*)

%RECURSION 148

%STORAGE 148

C

C/C++

attributes for variables 140, 249

declarations, syntax 247

equivalents for Language Environment

conditions 348

notes on using 194

reserved keywords 347

C/C++ commands

interpretive subset of 347

C++

See also ?

AT CALL breakpoints 164

AT ENTRY/EXIT breakpoints 163

blocks and block identifiers 157

classes 165

examining C++ objects 164

objects. 165

overloaded operator in C++ 163

setting breakpoints in C++ 163

static data 165

stepping through C++ programs 163

tasks for Debug Tool session

capturing output to stdout 62

debugging a DLL 62

debugging when not all parts compiled with

TEST 61

displaying raw storage 62

finding storage overwrite errors 64

finding uninitialized storage errors 65

getting a function traceback 63

getting a run-time frequency count 63

invoking interactive function calls

modifying value of variable 59

setting a breakpoint to halt 59, 65

stopping on a line if condition true 60

tracing the run-time path 63

viewing and modifying data members 60

template in C++ 163

using a C++ program 55

%CAAADDRESS

description of 128

for C/C++ 143

for COBOL 177

CAF (call access facility), using to invoke DB2

program 113

CALL %DUMP

syntax 235

call access facility (CAF), using to invoke DB2

program 113

CALL commands 234—240

summary table 234

CALL entry_name 239

CALL procedure, syntax 240

CALL, AT command, syntax 212

calls, function, for C/C++ 147

CALLS, LIST command, syntax 278

capturing I/O to system console, in COBOL 72

capturing output to stdout

in C 51

in C++ 62

CEETEST

examples, for C 33

examples, for COBOL 34

examples, for PL/I 35

invoking Debug Tool with 32

syntax 32

using 115

CEEUOPT run-time options module 111

CHANGE, AT command, syntax 214

CHANGE, Debug Tool setting, definition of 424

CHANGE, SET command, syntax 305

changing point of view 132

for C/C++ 162

for COBOL 185

changing source file in window 43

changing window layout in the session panel 95

character set 193

characters, searching 94

CICS

requirements for using Debug Tool in 116

restrictions for debugging 123

classes 165

classes in C++ 165

CLEAR commands 240—244

CLEAR prefix (Full-Screen Mode), syntax 243

closing Debug Tool session panel windows 96

CMS command (VM), syntax 244

COBOL

attributes for variables 175

command format 171

compatible attributes for 252

declarations 250

listing files 350

notes on using 194

reserved keywords 350

tasks

capturing I/O to system console 72

debugging when not all parts compiled with

TEST 72

displaying raw storage 73

finding storage overwrite errors 74

modifying the value of a variable 70

sample program for COBOL 66

setting a breakpoint to halt 70

setting breakpoint to halt 75

stopping on line if condition true 71

Index

- COBOL (*continued*)
 - tasks (*continued*)
 - tracing the run-time path 73
- COBOL commands
 - CALL entry_name 239
 - COMPUTE 245
 - declarations 250
 - EVALUATE 262
 - IF 271
 - INPUT 274
 - interpretive subset of 350
 - MOVE 288
 - PERFORM 293
 - SET 328
 - SET INTERCEPT 314
- CODE/370, definition of 424
- coexistence of Debug Tool with other debuggers 343
- coexistence with unsupported HLL modules 343
- Color Selection panel 98
- COLOR, SET command, syntax 306
- command format
 - Debug Tool 193
 - for COBOL 171
- Command line, Debug Tool 89
- command list, definition of 424
- command sequencing, full-screen mode 90
- command syntax help, getting for session 101
- commands
 - abbreviating 92, 194
 - alternative methods of input under IMS 115
 - delimiting 232
 - entering 193
 - entering Debug Tool 85
 - entering multiple line, without continuation 196
 - for C/C++, Debug Tool subset 138
 - for COBOL, Debug Tool subset 168
 - for PL/I, Debug Tool subset 186
 - getting online help for 198
 - interpretive subsets, description of 130
 - interpretive subsets, for C/C++ 347
 - interpretive subsets, for COBOL 350
 - line mode 107
 - multiline 195
 - order of processing, Debug Tool 90
 - prefix, using in Debug Tool 91
 - restrictions, COBOL 169
 - retrieving from log and source windows, Debug Tool 198
 - retrieving with RETRIEVE command 92
 - TSO, using to debug DB2 program 112
- COMMENT command, syntax 245
- comments, inserting into command stream 197
- Common Programming Interface (CPI), definition of 424
- common syntax elements 199
- compile unit, definition of 424
- compile_unit_name, description of 200
- compile, definition of 424
- compiler requirements
 - for DB2 110
 - for IMS 114
- compiler, definition of 424
- %CONDITION
 - constants, for C 334
 - description of 128
 - for C 225
 - for C/C++ 143
 - for COBOL 177
 - handling of 133, 188
 - Language Environment, C/C++ equivalents 348
- condition, definition of 424
- constants
 - Debug Tool interpretation of HLL 126
 - entering 197
 - HLL 126
 - PL/I 191
 - using in expressions, for COBOL 182
- constructor, stepping through 163
- container, definition of 424
- continuing lines 195
- conversational, definition of 424
- %COUNTRY
 - description of 128
 - for C/C++ 143
 - for COBOL 177
- COUNTRY, SET command, syntax 308
- CPI (Common Programming Interface), definition of 424
- __ctest()
 - examples 38
 - invoking Debug Tool with 38
 - syntax 38
- %CU
 - description of 128
 - for C/C++ 143
 - for COBOL 178
- cu_spec, description of 201
- CURSOR
 - syntax 246
 - using 93
- cursor commands
 - CLOSE command 96
 - CURSOR command 93
 - FIND command 94
 - OPEN command 96
 - SCROLL commands 85, 93
 - SIZE command 96
 - using in Debug Tool 91
 - WINDOW ZOOM command 97
- CURSOR, AT command, syntax 217

CURSOR, LIST command (Full-Screen Mode),
 syntax 278
 customizing screens 84

D

data set
 block size 3
 definition of 425
 specifying 88
 data type, definition of 425
 date field, definition of 425
 DATE, AT command, syntax 218
 DB2
 bind requirements for 111
 programming considerations 110
 programs, debugging in batch or line mode 111,
 112
 using Debug Tool with 109
 DBCS
 definition of 425
 SET DBCS command, syntax 308
 using 193
 using with C 194
 using with COBOL 175
 variable, assigning new value to 289
 DBCS, SET command, syntax 308
 debug session
 C tasks 41
 capturing output to stdout 51
 debugging a DLL 52
 debugging when not all parts compiled with
 TEST 51
 displaying raw storage 52
 finding storage overwrite errors 53
 finding uninitialized storage errors 54
 getting a function traceback 52
 invoking interactive function calls 52
 modifying value of variable 50
 setting a breakpoint to halt 50
 setting breakpoint to halt 55
 stopping on line if condition true 51
 tracing run-time path for code compiled with
 TEST 53
 C++ tasks 59
 COBOL tasks 70
 ending 41
 invoking your program 40
 PL/I tasks 79
 preparing for 40
 using a C program 45
 using a COBOL program 66
 using a PL/I program 75
 Debug Tool
 C/C++ commands, interpretive subset 138
 COBOL commands, interpretive subset 168

Debug Tool (*continued*)
 commands, subset 130
 condition handling 133
 DB2 programs, using with 111
 evaluation of HLL expressions 126
 exception handling, for C/C++ and PL/I 134
 functions, using with C/C++ 148
 IF command, allowable comparisons 350
 IMS programs, using with 114
 interface 84
 interpretation of HLL variables 126
 invoking under IMS 115
 MOVE command, allowable moves 352
 multilanguage programs, using 16
 optimized programs, using with 345
 PL/I commands, interpretive subset 186
 procedure, definition of 425
 SET command, allowable moves 354
 using in batch mode 107, 108
 using to debug ISPF applications 109
 variable, definition of 425
 Debug Tool interface 42
 Debug Tool, invoking your program with 40
 debug, definition of 425
 debuggers, other, coexistence with 343
 debugging a DLL
 in C 52
 in C++ 62
 debugging DB2 programs 109
 debugging in full-screen mode 40
 debugging when not all parts compiled with TEST
 in C++ 61
 in COBOL 72
 in PL/I 81
 declarations, for C/C++, syntax 247
 declarations, for COBOL 250
 DECLARE command (PL/I) 253
 declaring temporary variables, for C 139
 declaring temporary variables, for COBOL 174
 DEFAULT LISTINGS, SET command, syntax 309
 DEFAULT WINDOW, SET command, syntax 310
 default, definition of 425
 DELETE, AT command, syntax 218
 DESCRIBE command
 syntax 255
 using 157
 destructor, stepping through 163
 diagnostics, expression, for C/C++ 150
 DISABLE command, syntax 257, 258
 displaying halted location 44
 displaying raw storage
 in C 52
 in C++ 62
 in COBOL 73
 in PL/I 81

Index

- displaying variable value 44
- DO command (PL/I), syntax 259
- do/while command (C/C++), syntax 259
- double-byte character set (DBCS), definition of 425
- DOWN, SCROLL command 93
- DTCN, using
 - data entry verification 122
 - header area 118
 - input area 119
 - message line 120
 - modifying other options 122
 - PF key definitions 121
 - profile repository 121
 - screen 118
 - screen areas 118
- dual terminal mode (CICS) 116
- %DUMP, CALL command, syntax 235
 - See also* CALL %DUMP
- dynamic link library, definition of 425
- dynamic, definition of 425

E

- ECHO, SET command, syntax 310
- elements, unsupported, for PL/I 192
- ENABLE command, syntax 262
- enclave
 - definition of 425
 - invoking 103
 - multiple, debugging interlanguage communication application in 125
- ending a debug session 41
- ending Debug Tool within multiple enclaves 104
- enhancements xiii
- entering multiline commands without continuation 196
- entering PL/I statements, freeform 189
- entry point, definition of 425
- entry_name, CALL command (COBOL), syntax 239
- ENTRY, AT command, syntax 219
- %EPA
 - description of 128
 - for C/C++ 143
 - for COBOL 178
- %EPRn
 - description of 128
 - for C/C++ 142
 - for COBOL 177
- error numbers in LOG window 45
- EVALUATE command
 - restrictions on 170
 - syntax 262
- evaluating expressions
 - C/C++ 150
 - COBOL 181
 - HLL 126
- evaluation, expression, for C/C++ 145
- every_clause, description 208
- examining C++ objects 164
- examples
 - %HEX function for C/C++ 148
 - %HEX function for COBOL 182
 - %INSTANCES function for C/C++ 148
 - %STORAGE function for C/C++ 148
 - %STORAGE function for COBOL 182
 - assigning values to variables, for C/C++ 140
 - CEDF procedure 123
 - CEETEST calls, for PL/I 35
 - CEETEST function calls, for C 33
 - CEETEST function calls, for COBOL 34
 - changing point of view, for C/C++ 162
 - changing point of view, for COBOL 185
 - changing point of view, general 132
 - compile_unit_name 200
 - ctest function 38
 - declaring variables, for COBOL 174
 - displaying program variables, for C 139
 - displaying results of expression evaluation, for COBOL 181
 - displaying values of COBOL variables 174
 - expression evaluation, for C/C++ 146
 - function calls, for C/C++ 149
 - getting online command syntax help 198
 - invoking Debug Tool under CMS 30
 - invoking Debug Tool under MVS with TSO 29
 - line continuation, for COBOL 196
 - PLITEST Calls for PL/I 37
 - run-time TEST option 27
 - scope, for C 154
 - specifying run-time TEST option with #pragma 39
 - using #pragma for compile-time TEST option 8
 - using blocks in C 156
 - using COMPUTE command to assign values 173
 - using constants 197
 - using constants in expressions, for COBOL 182
 - using continuation characters 195
 - using Debug Tool with OPTIMIZE compile-time option 346
 - using MOVE command to assign values 173
 - using qualification, for C/C++ 158, 161
 - using qualification, for COBOL 183
 - using SET command to assign values 173
- exception handling for C/C++ and PL/I 134
- exception, definition of 425
- execute, definition of 425
- execution time, definition of 425
- execution-time environment, definition of 425
- EXIT, AT command, syntax 219
- expanded date field, definition of 425
- Expression command (C/C++), syntax 264
- Expression, LIST command, syntax 279

expressions
 definition of 425
 description of 201
 diagnostics, for C/C++ 150
 displaying values, for C 139
 displaying values, for COBOL 181
 evaluation for C/C++ 145, 150
 evaluation for COBOL 181
 evaluation of HLL 126
 evaluation, operators and operands for C 347
 for PL/I 191
 subset, description of 202
 using constants in, for COBOL 182

F

file, definition of 426
 finding characters or strings 94
 finding renamed source or listing file 45
 finding storage overwrite errors
 in C 53
 in C++ 64
 in COBOL 74
 in PL/I 83
 finding text in window 43
 finding uninitialized storage errors
 in C 54
 in C++ 65
 for command (C/C++), syntax 266
 freeform input, PL/I statements 189
 frequency count, definition of 426
 FREQUENCY, SET command, syntax 312
 full-screen image support (FSS), BTS 113
 full-screen mode
 AT CURSOR 217
 AT prefix 229
 CLEAR prefix 243
 continuation character, using in 195
 CURSOR 91, 93, 246
 definition of 426
 DESCRIBE CURSOR 256
 DISABLE prefix 258
 ENABLE Prefix 262
 FIND 265
 IMMEDIATE 273
 LIST CURSOR 278
 PANEL 291
 PANEL COLORS 98
 PANEL LAYOUT 95
 PANEL LISTINGS 291
 PANEL PROFILE 99
 PANEL SOURCE 291
 Prefix 295
 QUERY Prefix 299
 RETRIEVE 300
 SCROLL 93

full-screen mode (*continued*)
 SET COLOR 306
 SET DEFAULT SCROLL 309
 SET DEFAULT WINDOW 310
 SET KEYS 315
 SET LOG NUMBERS 316
 SET MONITOR NUMBERS 316
 SET PROMPT 320
 SET SCREEN 323
 SET SCROLL DISPLAY 324
 SET SUFFIX 325
 SHOW Prefix 329
 WINDOW CLOSE 96, 339
 WINDOW OPEN 96, 340
 WINDOW SIZE 96, 340
 WINDOW ZOOM 97, 341

full-screen mode, debugging in 40
 full-screen mode, using session panel in 85
 full-screen session interface, description 1
 function calls, for C/C++ 147
 function, unsupported for PL/I 192

G

generation function 137
 getting a function traceback
 in C 52
 in C++ 63
 in PL/I 81
 getting a run-time frequency count in C++ 63
 global data 166
 global scope operator 166
 GLOBAL, AT command, syntax 220
 GO command, syntax 267
 GOTO command 268
 GOTO LABEL command, syntax 269
 %GPRn
 description of 128
 for C/C++ 142
 for COBOL 176

H

H constant (COBOL) 197
 halted location, displaying 44
 %HARDWARE
 description of 128
 for C/C++ 143
 for COBOL 178
 header fields, Debug Tool session panel 84
 help, how to find 42
 hex function 148
 for C/C++ 136
 for COBOL 136, 182
 for PL/I 136
 hex
 hex 136, 182

Index

High Level Language (HLL), definition of 426
highlighting, changing in Debug Tool session panel 98
HISTORY, SET command, syntax 313
HLL, definition of 426
hook
 compiling with, C 5
 compiling with, COBOL 10
 compiling with, PL/I 13
 definition of 426
 general description 3
 removing from application 344
 rules for placing 8, 10

I

I/O, definition of 426
IBM SAA AD/Cycle Language Environment/370, definition of 426
IF command
 allowable comparisons, Debug Tool 350
 restrictions on 170
 syntax 271
if command (C/C++), syntax 270
IF command (PL/I), syntax 272
IMMEDIATE command, syntax 273
improving Debug Tool performance 344
IMS
 programming considerations 113
 programs, debugging in batch or line mode 114, 115
 using Debug Tool with 113
inactive block, definition of 426
information, displaying environmental 157
initial setting, definition of 426
input areas, order of processing, Debug Tool 90
INPUT command 274
%INSTANCES 148
 for C/C++ 136
 for PL/I 136
interactive, definition of 426
InterLanguage Communication (ILC) application, debugging 125
interlanguage programs, using with Debug Tool 16
interpretive subset
 of PL/I commands 186
interpretive subsets
 general description 130
 of C/C++ commands 347
 of COBOL commands 168, 350
interpretive subsets.
 of C/C++ commands 138
INTERRUPT, Language Environment run-time option 135
intrinsic functions, Debug Tool 127
invoking Debug Tool 29
 with the CEETEST function call 37

invoking Debug Tool (*continued*)
 ctest function 38—39
 DB2 program with TSO 113
 run-time TEST option 18—25
 under CICS 28
 under CMS 30
 under IMS 115
 under MVS in TSO 29
 within an enclave 103
invoking Debug Tool under CICS with CEDF 123
invoking Debug Tool under CICS with compile-time directives 122
invoking interactive function calls
 in C 52
 in C++ 62
invoking your program 40
ISPF applications 109

K

KEYS, SET command, syntax 315

L

LABEL, AT command, syntax 222
Language Environment
 conditions, C/C++ equivalents 348
 definition of 426
LANGUAGE, SET NATIONAL command, syntax 317
LANGUAGE, SET PROGRAMMING command, syntax 319
LAST, LIST command, syntax 280
LEFT, SCROLL command 93
library routine, definition of 426
%LINE
 description of 129
 for C/C++ 143
 for COBOL 178
line breakpoint, setting 44
line continuation 195
 line continuation, for C 196
line mode
 debugging DB2 programs in 112
 debugging IMS programs in 114
LINE NUMBERS, LIST command, syntax 284
line wrap, definition of 426
LINE, AT command, syntax 230
LINES, LIST command, syntax 285
link requirements
 for DB2 111
 for IMS 114
link-edit, definition of 426
linkage editor, definition of 426
LIST (blank) command, syntax 275
LIST AT command, syntax 275

LIST CALLS command, syntax 278
 LIST CURSOR command (Full-Screen Mode),
 syntax 278
 LIST Expression command, syntax 279
 LIST FREQUENCY 280
 LIST LAST command, syntax 280
 LIST MONITOR command, syntax 281
 LIST NAMES command 281
 LIST ON command (PL/I), syntax 283
 LIST PROCEDURES command, syntax 283
 LIST REGISTERS command
 syntax 284
 using 167
 LIST STATEMENT NUMBERS command, syntax 284
 LIST STATEMENTS command, syntax 285
 LIST STORAGE command
 syntax 286
 using with PL/I 189
 LIST commands 274, 286
 listing
 definition of 426
 files, for COBOL 350
 registers 167
 SET DEFAULT LISTINGS command, syntax 309
 listing file, finding renamed 45
 literal constants, entering 197
 %LOAD
 load module, definition of 426
 load_module_name, description of 202
 load_spec, description of 202
 LOAD, AT command, syntax 224
 log data set, specifying 88
 LOG NUMBERS, SET command, syntax 316
 LOG window, error numbers in 45
 LOG, SET command, syntax 315
 low-level debugging 166
 %LPRn
 description of 128
 for C/C++ 142
 for COBOL 177

M

mechanisms for invoking Debug Tool under CICS 117
 mode
 batch mode (CICS) 116
 dual terminal (CICS) 116
 single terminal (CICS) 116
 modifying value of variable in C 50
 MONITOR NUMBERS, SET command, syntax 316
 MONITOR, LIST command, syntax 281
 more than one language, debugging programs with 16
 MOVE command
 allowable moves, Debug Tool 352
 restrictions on 169
 syntax 288

MOVE command (*continued*)
 using to assign values to variables 173
 moving around windows in Debug Tool 93
 moving the cursor, Debug Tool 93
 MSGID, SET command, syntax 317
 multilanguage programs, using with Debug Tool 16
 multiline commands
 continuation character, using in 195
 without continuation character 196
 multiple enclaves
 ending Debug Tool 104
 interlanguage communication application,
 debugging 125
 invoking 103
 using breakpoints 104
 multitasking, Debug Tool 108
 multitasking, definition of 426
 multitasking, restrictions 109

N

NATIONAL LANGUAGE, SET command, syntax 317
 navigating session panel windows 93
 %NLANGUAGE
 description of 129
 for C/C++ 143
 for COBOL 178
 nonconversational, definition of 426
 nondate, definition of 426
 Null command, syntax 289
 NUMBERS, LIST STATEMENT command, syntax 284
 NUMBERS, SET LOG command, syntax 316
 NUMBERS, SET MONITOR command, syntax 316

O

objects, C 153
 objects. 165
 OCCURRENCE, AT command, syntax 225
 ON command (PL/I), syntax 289
 ON, LIST command (PL/I), syntax 283
 online help, getting for session 101
 Open Edition, PL/I 109
 opening Debug Tool session panel windows 96
 operators and operands for C 347
 OPTIMIZE compile-time option, using Debug Tool
 with 345
 options module, CEEUOPT run-time 111
 Options, definition of 426
 options, run-time 18
 overloaded operator in C++ 163

P

PACE, SET command, syntax 318

Index

- panel
 - Color Selection 98
 - header fields, Session 84
 - Profile 99
 - Session 85, 89
 - Source Identification 292
 - Window Layout Selection 95
- parameter, definition of 427
- partitioned data set (PDS), definition of 427
- path point
 - definition of 427
 - differences between languages 228
- PATH, AT command, syntax 228
- %PATHCODE
 - description of 129
 - for C/C++ 144
 - for COBOL 178
- PDS (partitioned data set), definition of 427
- PERFORM command
 - restrictions on 171
 - syntax 293
- performance, improving Debug Tool 344
- PF key, setting 45
- PF keys, defining in Debug Tool 91
- PL/I
 - attributes for variables 254
 - built-in functions 191
 - condition handling 188
 - constants 191
 - DECLARE 253
 - expressions 191
 - GENERATION function for 137
 - notes on using 194
 - session variables 189
 - statements 186
 - tasks
 - debugging when not all parts compiled with TEST 81
 - displaying raw storage 81
 - finding storage overwrite errors 83
 - getting a function traceback 81
 - modifying value of variable 80
 - setting a breakpoint to halt 79
 - setting breakpoint to halt 83
 - stopping on line if condition is true 80
 - tracing run-time path for code compiled with TEST 82
- PL/I commands
 - ANALYZE 205
 - Assignment 206
 - AT ALLOCATE 209
 - BEGIN 232
 - DO 259
 - IF 272
 - LIST ON 283
 - ON 289
- PL/I commands (*continued*)
 - SELECT 303
 - SET WARNING 327
- PL/I structures 190
- %PLANGUAGE
 - description of 129
 - for C/C++ 144
 - for COBOL 180
- PLITEST 37
- positioning lines at top of windows 94
- #pragma
 - specifying compile-time TEST option 8
 - specifying run-time TEST option with 39
- precompile requirements for DB2 110
- Prefix area, Debug Tool 89
- prefix area, definition of 427
- preparing and using CEEUOPT to invoke Debug Tool 122
- preparing for debugging 40
- primary entry point, definition of 425
- PROCEDURE command 295
- procedure, CALL, syntax 240
- procedure, definition of 427
- PROCEDURES, LIST command, syntax 283
- process, definition of 427
- Profile panel 99
- profile settings, changing in Debug Tool 99
- PROFILE, PANEL command 427
- Profile, definition of 427
- program
 - CICS, debugging 116
 - DB2, debugging 109
 - definition of 427
 - description of 129
 - for C/C++ 144
 - for COBOL 180
 - IMS, debugging 113
 - invoking for a session 29
 - preparing DB2 110
 - preparing IMS 114
 - reducing size 344
 - stepping through 44
 - variables, accessing for C 138
 - variables, accessing for COBOL 172
- program hook
 - compiling with, C 5
 - compiling with, COBOL 10
 - compiling with, PL/I 13
 - general description 3
 - removing 344
 - rules for placing 8, 10
- program preparation
 - compile-time TEST option, for C 5
 - compile-time TEST option, for C++ 9
 - compile-time Test option, for COBOL 10
 - compile-time TEST option, for PL/I 13

program preparation (*continued*)
 considerations, size and performance 344
 for DB2 110
 for IMS 114
 run-time TEST option 18
 program unit, definition of 427
 program variable, definition of 427
 PROGRAMMING LANGUAGE, SET command,
 syntax 319
 PROMPT, SET command, syntax 320
 pseudo-conversational transaction, definition of 427
 PX constant (PL/I) 197

Q

qualification
 definition of 427
 description of, for C/C++ 158
 description of, for COBOL 183
 general description 131
 using, for C/C++ 160
 using, for COBOL 183
 QUERY command, syntax 296
 QUERY Prefix, syntax 299
 QUIT command 299

R

%RC
 description of 129
 for C/C++ 144
 for COBOL 180
 record format, definition of 427
 record, definition of 427
 %RECURSION 148
 for C 136
 for PL/I 136
 recursion 136
 recursion function 148
 reference, definition of 427
 references, description of 202
 registers, listing 167
 removing statement and symbol tables 344
 repeating breakpoints 208
 requirements
 bind, for DB2 111
 compile, for DB2 110
 compile, for IMS 114
 for debugging CICS programs 116
 link, for DB2 111
 link, for IMS 114
 precompile, for DB2 110
 reserved keywords
 for C 347
 for COBOL 350

restrictions
 arithmetic expressions for COBOL 181
 expression evaluation, for COBOL 181
 on COBOL-like commands 169
 string constants in COBOL 182
 when debugging multilanguage applications 109
 when debugging under CICS 123
 when using a continuation character 171
 when using CALL 171
 when using COMPUTE 169
 when using EVALUATE 170
 when using IF 170
 when using MOVE 169
 when using PERFORM 171
 when using SET 170
 retrieving lines from Log or Source windows 92
 REWRITE, SET command, syntax 323
 RIGHT, SCROLL command 93
 RUN command
 See ?
 RUN subcommand 112
 run time
 definition of 427
 environment, displaying attributes of 157
 option, TEST 18
 option, TEST(ERROR, ...), for PL/I 189
 options module, CEEUOPT 111
 specifying TEST option with #pragma 39
 suboption processing order 25
 run unit, definition of 427
 run-time environment, definition of 427
 run, definition of 427
 Run, definition of 427
 %RUNMODE
 description of 129
 for C/C++ 144
 for COBOL 180
 running a program 44

S

SAA (Systems Application Architecture), definition
 of 427
 SAA language, definition of 427
 sample C++ program 55
 SBCS (single-byte character set), definition of 428
 scopes, C 153
 SCREEN, SET command, syntax 323
 screens, customizing 84
 SCROLL
 syntax 301
 using 85, 93
 using SCROLL TO 94
 Scroll area, Debug Tool 89
 SCROLL, SET DEFAULT command, syntax 309

Index

- scrolling SOURCE window 43
- searching for characters or strings 94
- SELECT command (PL/I), syntax 303
- semantic error, definition of 428
- sequence number, definition of 428
- session variable
 - definition of 428
- SET CHANGE command, syntax 305
- SET COLOR command, syntax 306
- SET command
 - allowable moves, Debug Tool 354
 - restrictions on 170
 - syntax 328
 - using to assign values to variables 173
- SET COUNTRY command, syntax 308
- SET DBCS command, syntax 308
- SET DEFAULT LISTINGS command, syntax 309
- SET DEFAULT WINDOW command, syntax 310
- SET ECHO command, syntax 310
- SET EXECUTE command, syntax 312
- SET FREQUENCY command, syntax 312
- SET HISTORY command, syntax 313
- SET INTERCEPT command
 - syntax 314
 - using with C programs 151
- SET KEYS command, syntax 315
- SET LOG command, syntax 315
- SET LOG NUMBERS command, syntax 316
- SET MONITOR NUMBERS command, syntax 316
- SET MSGID command, syntax 317
- SET NATIONAL LANGUAGE command, syntax 317
- SET PACE command, syntax 318
- SET PROGRAMMING LANGUAGE command,
 - syntax 319
- SET PROMPT command, syntax 320
- SET QUALIFY command
 - syntax 321
 - using, for C/C++ 162
 - using, for COBOL 185
- SET REFRESH command
 - syntax 322
 - using 109
- SET REWRITE command, syntax 323
- SET SCREEN command, syntax 323
- SET SOURCE command, syntax 324
- SET SUFFIX command, syntax 325
- SET TEST command, syntax 326
- SET WARNING command
 - syntax 327
 - using with PL/I 192
- SET commands 304, 328
- setting a breakpoint to halt
 - before calling a NULL function
 - in C 55
 - in C++ 65
 - before calling an invalid program (COBOL) 75
- setting a breakpoint to halt (*continued*)
 - before calling an undefined program (PL/I) 83
 - when certain functions are called
 - in C 50
 - in C++ 59
 - in PL/I 79
 - when certain routines are called (COBOL) 70
- setting a line breakpoint 44
- setting a PF key 45
- setting breakpoints in C++ 163
- short forms for commands, using 92
- SHOW Prefix command, syntax 329
- single terminal mode (CICS) 116
- single-byte character set (SBCS), definition of 428
- sizing session panel windows 96
- source file, finding renamed 45
- Source Identification panel, Debug Tool 292
- source window
 - Debug Tool 86, 89
 - definition of 428
 - retrieving input lines from, Debug Tool 198
 - retrieving lines from, Debug Tool 92
- SOURCE, SET command, syntax 324
- specifying a range of statements 204
- %STATEMENT
 - description of 130
 - for C/C++ 145
 - for COBOL 180
- STATEMENT NUMBERS, LIST command, syntax 284
- statement tables, removing 344
- statement_id_range, description of 203
- statement_id, description of 203
- statement_label, description of 204
- STATEMENT, AT command, syntax 230
- STATEMENTS, LIST command, syntax 285
- statements, PL/I 186, 189
- statements, specifying a range 204
- static data 165
- static data in C++ 165
- static, definition of 428
- step, definition of 428
- Step, definition of 428
- stepping through a program 44
- stepping through C++ programs 163
- stmt_id_spec, description of 203
- stopping on line if condition true, C program 51
- storage classes, C 155
- storage function 148
 - for C 136
 - for COBOL 136, 182
 - for PL/I 136
- STORAGE
 - storage 136, 182
- storage, definition of 428
- string substitution, using 92

subroutine, definition of 428
 subset mode, CMS 244
 substitution, using string 92
 %SUBSYSTEM
 description of 130
 for C/C++ 145
 for COBOL 180
 suffix area, definition of 428
 SUFFIX, SET command, syntax 325
 switch command (C/C++), syntax 331
 symbol tables, removing 344
 syntactic analysis, definition of 428
 syntax diagram for
 %GENERATION function 137
 %INSTANCES function 136
 %RECURSION function 136
 %STORAGE function 136
 block_spec 199
 C compile-time TEST option 6
 C++. compile-time TEST option 9
 CEETEST, for C 32
 CEETEST, for COBOL 32
 CEETEST, for PL/I 32
 COBOL compile-time TEST option 10
 ctest function 38
 cu_spec 201
 HEX function 136
 load_spec 202
 PL/I compile-time TEST option 13
 PLITEST built-in subroutine 37
 run-time TEST option 19
 statement_id_range 203
 stmt_id_spec 203
 syntax, common elements 199
 %SYSTEM
 description of 130
 for C/C++ 145
 for COBOL 180
 SYSTEM command, syntax 333
 system commands, issuing, Debug Tool 90
 Systems Application Architecture (SAA), definition of 428

T

template in C++ 163
 temporary variable
 declaring, for COBOL 174
 definition of 428
 TERMINATION, AT command, syntax 231
 TEST
 restrictions 6
 TEST option, compile-time 40
 for C 5
 for C++ 9
 for COBOL 10

TEST option, compile-time (*continued*)
 for PL/I 13
 using for DB2 110
 using for IMS 114
 TEST option, run-time
 as parameter on RUN subcommand 112
 for PL/I 189
 processing order for suboptions 25
 specifying with #pragma 39
 syntax 19
 using 18
 TEST, SET command, syntax 326
 thread id, definition of 428
 thread, definition of 428
 token, definition of 428
 tracing run-time path for code compiled with TEST
 in C 53
 in C++ 63
 in COBOL 73
 in PL/I 82
 TRAP, Language Environment run-time option 133, 135
 TRIGGER command, syntax 334
 truncating commands 92, 194
 TSO command
 syntax 336
 using to debug DB2 program 112

U

unsupported HLL modules, coexistence with 343
 unsupported PL/I language elements 192
 UP, SCROLL command 93
 USE command, syntax 336
 using constants in expressions, for COBOL 182
 using Debug Tool
 finding renamed source or listing file 45
 utility, definition of 428

V

values
 assigning to C variables 140
 assigning to COBOL variables 172
 variable
 accessing program, for C 138
 accessing program, for COBOL 172
 assigning values to, for C 140
 assigning values to, for COBOL 172
 compatible attributes in multiple languages 125
 DBCS, assigning new value to 289
 definition of 428
 description of Debug Tool, for COBOL 175
 displaying, for C 139
 displaying, for COBOL 174
 removing 240

Index

- variable (*continued*)
 - session, for PL/I 189
- variable value, displaying 44
- variables
 - HLL 126
 - modifiable Debug Tool 128
 - nonmodifiable Debug Tool 128
 - qualifying 131
- viewing and modifying data members in C++ 60

W

- warning, for PL/I 192
- while command (C/C++), syntax 338
- window control, Debug Tool
 - changing source files 43
 - displaying halted location 44
 - finding text 43
 - scrolling 43
- Window id area, Debug Tool 89
- Window Layout Selection panel 95
- window, error numbers in 45
- WINDOW, SET DEFAULT command, syntax 310
- windowed date field, definition of 428
- windows, Debug Tool 42
- word wrap, definition of 429

Z

- zooming a window, Debug Tool 97

We'd Like to Hear from You

Debug Tool
User's Guide and Reference
Release 2
Publication No. SC09-2137-03

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.
- 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
User's Guide and Reference
Release 2
Publication No. SC09-2137-03**

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Grammatically correct and consistent	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Graphically well designed	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

Phone No.



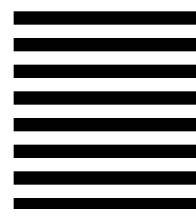
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department W92/H3
International Business Machines Corporation
P.O. Box 49023
San Jose, CA 95161-9945



Fold and Tape

Please do not staple

Fold and Tape



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2137-03



Spine information:



Debug Tool
User's Guide and Reference

Release 2