JIMMY

Prime Computer, Inc.

PRIME

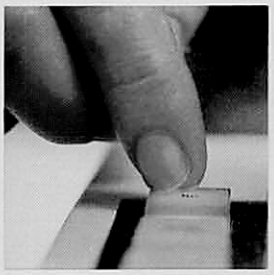
DOC4033-193P
Source Level Debugger User's Guide
DBG Revision 19.2














Source Level Debugger User's Guide

DOC4033-193

Second Edition

by

A. Paul Cioto

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Independent Product Release (IPR) 1.0 - 19.1 (Rev. 1.0 - 19.1).

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

COPYRIGHT INFORMATION

CREDITS

Data Entry: Nancy Cormier
Hope Eldredge
Lydia Herrera
Sabra MacArthur
Mary Mixon

Illustrations: Marcella Gallardo

Graphic Support: Therese Bacharz
Leo Maldonado
Mike Moyle
George Shaw
Robert Stuart

Editing: Mary Callaghan
Joan Frongello
Betty B. Hoskins

Technical Support: Stephen Evans Alley
James Craig Burley
Evelyn Burns
Larry Epstein
Ritchie Kolnos
Anne Ladd
Debra Minard

PRINTING HISTORY — SOURCE LEVEL DEBUGGER USER'S GUIDE

Edition	Date	Number	Software Release
First Edition	January, 1980	IDR4033	17.2
Update 1	December, 1980	PTU2600-068	18.1
Update 2	July, 1982	PTU2600-085	19.0
Second Edition	January, 1984	DOC4033-193	1.0 - 19.1

The Second Edition is a complete rewrite of IDR4033. Therefore, there are no vertical bars in the page margins to indicate changes since the First Edition.

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000 X4837

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

Contents

APPENDIXES

About This Book

PURPOSE AND AUDIENCE

This book is a user's guide to Prime's Source Level Debugger, which is a powerful, interactive, high-level language debugger that helps you find out why your program failed at execution time.

This book documents Debugger features up to and including Master Disk Revision 19.2 (Rev. 19.2) plus the features contained in this Independent Product Release (IPR) 1.0 - 19.1 (Rev. 1.0 - 19.1). These features include Debugger support for seven Prime high-level languages -- FORTRAN IV, FORTRAN 77, Pascal, PL/I Subset G, COBOL 74, RPG II V-mode, and C.

As a user's guide, this book is written and organized for users who are not necessarily familiar with the Debugger. You do not need to be an experienced programmer or know anything about assembly language or machine architecture to use this book or learn the Debugger. All you need is the ability to write programs in any of the Prime high-level languages mentioned above.

HOW TO USE THIS BOOK

This book is divided into three parts. Part I (Chapters 1 and 2) explains what the Source Level Debugger is, how it works, and what it can do. Chapter 1 also gives a chapter-by-chapter description of the contents of this book and offers some useful related documents. Part II (Chapters 3 to 11) teaches you how to invoke the Debugger and how to use it. Part III (Chapters 12 and 13) discusses advanced techniques and features. Examples are given extensively throughout the book.

Twelve appendixes follow Part III. Appendixes A to G offer sample debugging sessions with each of Prime's high-level languages. Appendix H offers some special considerations for using the Debugger with all languages. Appendix I offers some sample debugging strategies. Appendix J summarizes all of the Debugger's 53 commands in alphabetical order, giving a brief description of each command, a reference to the chapter in which it is discussed, and brief descriptions of command line arguments and options. Appendix K is a glossary of Debugger-related terms used throughout the book. Appendix L lists Debugger commands chapter by chapter.

If you are not familiar with the Debugger, read Chapters 1, 2, and 3, in that order. Chapter 3, GETTING STARTED, teaches you how to invoke the Debugger and begin using some of its fundamental commands. You should be able to begin using the Debugger after reading this chapter. After using the commands presented in Chapter 3, you should read Chapter 4, CONVENTIONS, TERMS, AND CONCEPTS, and then the other chapters to learn the vast assortment of Debugger features. You may also want to look at Appendix I, STRATEGIES IN DEBUGGING, and the appendixes that offer sample sessions with the languages you use.

If you are already familiar with the Debugger, turn to the sections in Parts II and III that explain the Debugger features with which you might need help. You should also use Appendix J, which summarizes all Debugger commands and command line syntax.

For your convenience, the first page of each chapter in Parts II and III lists the commands that are discussed in that chapter. Also, a chapter-by-chapter list of all Debugger commands is given just before the index in Appendix L. Official command line formats are highlighted with gray shades throughout the book (excluding Appendix J). It is hoped that these highlighting techniques will help you locate a command more quickly and easily.

DOCUMENTATION CONVENTIONS

The following conventions are used throughout this book:

Convention	Explanation	Example
underlining in examples of computer-user dialog	In examples of computer-user dialog, user input is underlined and system output is not.	> TYPE A + B fixed decimal (4)
underlining for abbreviations	In command line formats, legal abbreviations for command names and option names are underlined.	RESUBMIT
UPPERCASE	Words in uppercase signify command names, option names, variable names, and the names of other data objects. They may be entered in either uppercase or lowercase.	The RESTART command The -DEBUG option
UPPERCASE in examples	User input in examples of computer-user dialog appears in uppercase for consistency.	> MACRO RS -DELETE >
lowercase	In command and statement formats, words in lowercase indicate items for which you must substitute a suitable value.	LOADSTATE filename
brackets []	In command formats, brackets enclose a list of one or more optional items. Choose none, one, or more of these items. (Do not confuse these brackets with action list or command list brackets.)	STEPIN [value]
braces {}	In command formats, braces enclose a vertical list of items. Choose one and only one of these items.	ETRACE { ON / ARGS / OFF }
ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	[,variable-2 ...]

parentheses ()	In command and statement formats, parentheses must be entered exactly as shown.	[(argument-list)]
hyphen -	Whenever a hyphen appears in a command line option, it is a required part of that option.	MACRO -EDIT
backward and forward slashes \ /	In command and statement formats, backward and forward slashes must be entered exactly as shown.	program-block-name\
plus sign +	In statement formats, a plus sign must be entered exactly as shown.	+line-offset
angle brackets <>	Angle brackets must be used as shown to separate the elements of a pathname.	<FOREST>BEACH>LEAF
shaded areas	In addition to some illustrations, gray shades appear on all Debugger command line formats throughout the book. (The shades do not appear in Appendix J.)	RESTART [step-command]

PART I

Overview

1

What Is the Source Level Debugger?

INTRODUCTION

What is the Source Level Debugger? The Source Level Debugger is Prime's powerful, easy-to-use, interactive, high-level-language debugging tool that helps you find out why your program failed at execution time.

Why is the Source Level Debugger easy to use? Through the use of several Debugger commands, you can control and monitor the execution of your program at the source code level. This means the Debugger understands the language in which your program is coded. You can "talk" with the Debugger interactively while you debug.

Can novice programmers use the Debugger? Yes — you do not need to know anything about assembly language or machine architecture to use the Debugger. All you need is the ability to write programs in one of these seven Prime high-level languages:

- FORTRAN IV
- FORTRAN 77
- Pascal
- PL/I Subset G
- COBOL 74
- RPG II V-mode
- C

You simply create and edit your program using one of Prime's text editors, then compile, load, execute, and test your program interactively under the control of the Debugger.

Does one Debugger handle programs written in all of these languages? Yes, because the Debugger is multilingual. Prime's Source Level Debugger understands the syntax of all languages. It communicates with you in your program's language, switching from one language syntax to another, if necessary. This is particularly useful when your programs call procedures, functions, subroutines, or other programs that are written in different languages. You need only learn one set of Debugger commands to debug a program written in any language.

Figure 1-1 illustrates the Debugger's multilingual capabilities — how it helps you get rid of bugs in programs written in seven of Prime's high-level languages.

HOW DOES THE DEBUGGER WORK?

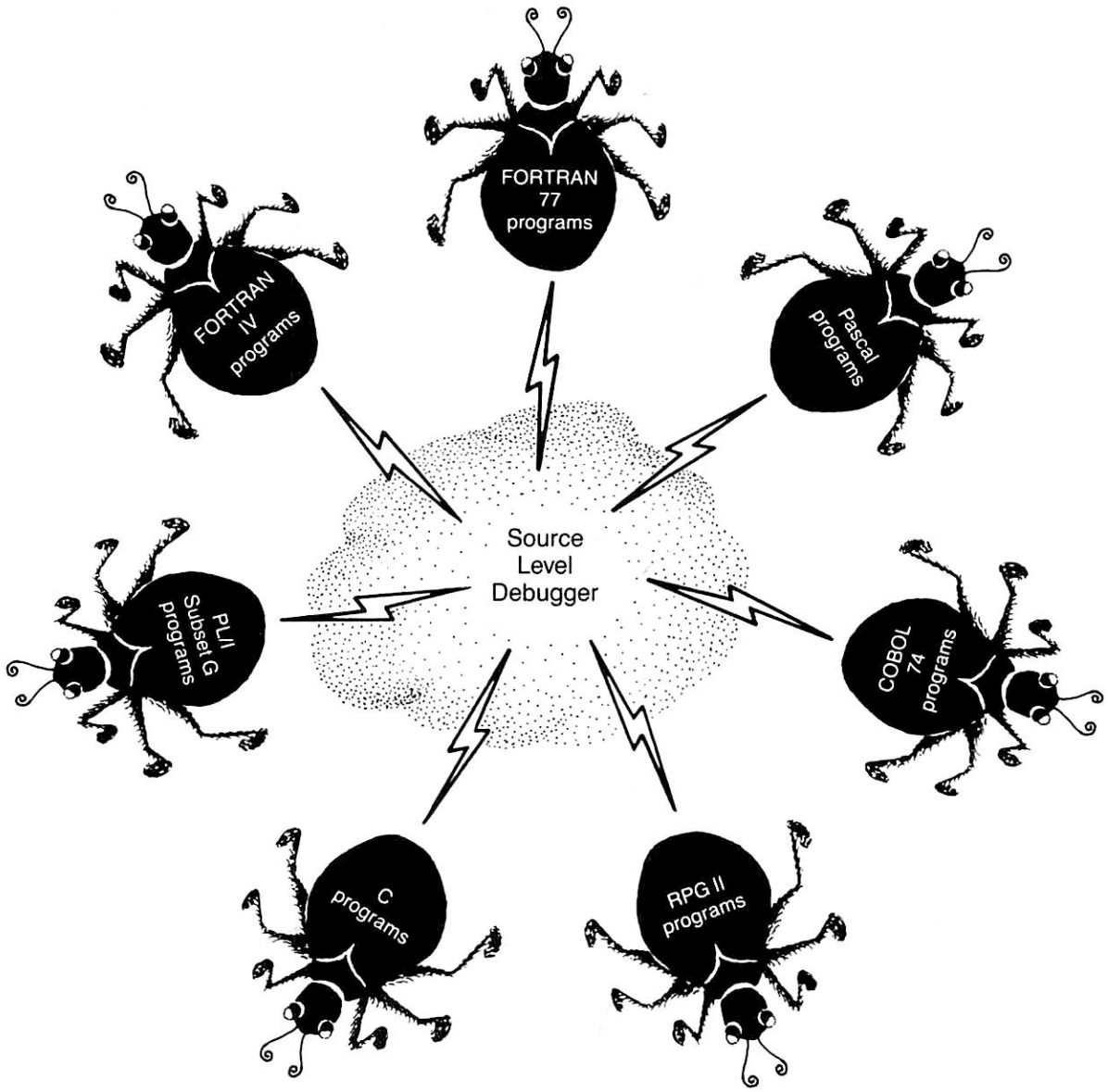
Prime's Source Level Debugger can run on any Prime CPU capable of supporting 64V or 32I addressing mode. This includes Prime's 50 Series machines. In addition to the Debugger itself, your system would require Prime-supplied software comprising a software development environment — the PRIMOS operating system, compilers, loaders, libraries, as well as other utilities such as Prime's text editors, EDITOR and EMACS.

Once you have the necessary software, you can debug by following a simple four-step process:

1. Compile your program with the -DEBUG option.
2. Load your program.
3. Execute your program.
4. Enter the Debugger and begin debugging.

During Step 1, compilation, the language compiler translates your program into binary form and adds debugging information, such as information about variables and locations of instructions corresponding to source statements in the object program.

Step 2, loading, produces an executable program (runfile) plus additional debugging information — a symbol table that describes the name, location, and attribute of each source program variable, and a statement map that contains the location of compiled code corresponding to each source language statement.



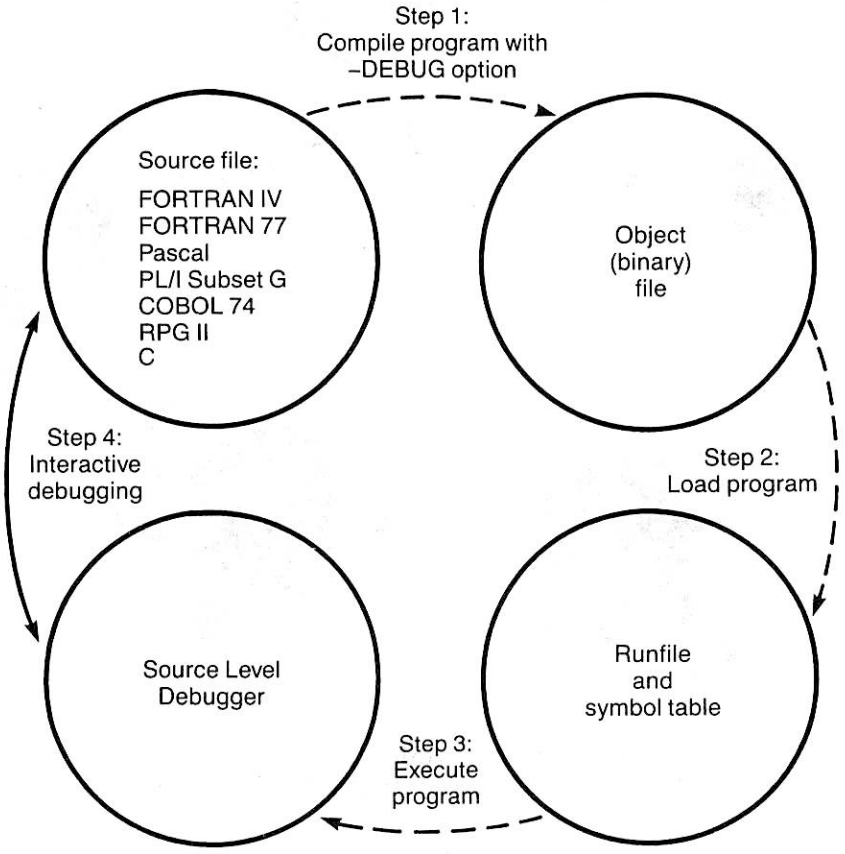
The Source Level Debugger Helps You Get Rid of Bugs in Programs Written in Seven of Prime's High-level Languages

Figure 1-1

Program execution takes place at Step 3. At this point, your program will either execute successfully or unsuccessfully. If it is unsuccessful, your output will be incorrect or a runtime error will prevent your program from completing execution.

Now you will want to go on to Step 4 and begin debugging. After you enter the Debugger using the DBG command, the Debugger uses the special debugging information in the symbol table to relate the object (binary) code, which the machine understands, to the high-level source code, which you understand. Therefore, you and the Debugger can "talk" to each other interactively at the source language level and the Debugger can perform its operations at the machine language level. This process makes it easy for you to manipulate program execution from within the Debugger. As Chapters 2 and 3 point out, you can execute your program, suspend execution, examine and trace the values of variables, continue execution, and do a lot more.

Figure 1-2 illustrates the four-step process.



Debugging Begins With a Four-Step Process
Figure 1-2

CONTENTS OF THIS BOOK

Here is a chapter-by-chapter summary of the contents of this book:

Part I -- Overview

- Chapter 1 describes what the Source Level Debugger is, how it works, what this book contains, and useful related documents.
- Chapter 2 overviews the Debugger's features and commands.

Part II -- Using the Debugger

- Chapter 3 gets you started by showing you how to enter the Debugger and how to use some of the Debugger's fundamental commands.
- Chapter 4 defines conventions, terms, and concepts used throughout this book.
- Chapter 5 discusses breakpoints and most of the Debugger's program control features.
- Chapter 6 discusses data manipulation features.
- Chapter 7 discusses single stepping and calling program blocks.
- Chapter 8 discusses tracing capabilities.
- Chapter 9 shows you how customize your Debugger commands by creating and using command macros.
- Chapter 10 shows you how to modify your Debugger commands and save breakpoints, tracepoints, and macros.
- Chapter 11 discusses miscellaneous Debugger features.

Part III -- Advanced Techniques and Features

- Chapter 12 lists some examples of advanced macros.
- Chapter 13 discusses miscellaneous advanced Debugger features.

Appendixes

- Appendix A gives sample debugging sessions with programs written in FORTRAN IV.
- Appendix B gives sample debugging sessions with FORTRAN 77.
- Appendix C gives sample debugging sessions with Pascal.
- Appendix D gives sample debugging sessions with PL/I Subset G.
- Appendix E gives sample debugging sessions with COBOL 74.
- Appendix F gives sample debugging sessions with RPG II.
- Appendix G gives sample debugging sessions with C.
- Appendix H offers some special considerations for using the Debugger with all languages.
- Appendix I offers some strategies for debugging.
- Appendix J summarizes all of the Debugger's commands, giving each command's format syntax and defining each part of that syntax.
- Appendix K is a glossary of Debugger-related terms used throughout this book.
- Appendix L lists Debugger commands chapter by chapter, providing you with a quick reference.

OTHER USEFUL RELATED DOCUMENTS

In addition to the Source Level Debugger User's Guide, some other documents would be useful at your installation. These documents include:

- Prime User's Guide — This guide provides complete information on all Prime system utilities, including instructions for creating, loading, and executing programs written in Prime's high-level languages.
- The guides that document the Prime high-level languages you use:

  The FORTRAN Reference Guide (for FORTRAN IV)

  The FORTRAN 77 Reference Guide

  The Pascal Reference Guide

  The PL/I Subset G Reference Guide

The COBOL 74 Reference Guide

The RPG II V-mode Compiler Reference Guide

The C User's Guide

- SEG and LOAD Reference Guide — If you wish to know and control the load process in more detail and use the full range of Prime loader capabilities, get this book. Ordinarily, to load and execute programs you need only the information given in the Prime User's Guide.

- EMACS Primer and EMACS Reference Guide — These guides describe Prime's screen text editor, EMACS. The Primer is written for users not familiar with EMACS. The reference guide is for users already familiar with EMACS.

- New User's Guide to EDITOR and RUNOFF — This guide contains complete information about Prime's line text editor, EDITOR, and Prime's text formatting utility, RUNOFF. It also provides a basic introduction to the Prime system for users with little or no computer experience.

- PRIMOS Commands Reference Guide — This guide provides a complete description of all PRIMOS level commands.

- Subroutines Reference Guide — This guide describes Prime's large selection of applications-level subroutines and PRIMOS operating system subroutines, which can be declared in your program and then referenced from any point within the program.

- Assembly Language Programmer's Guide — This guide contains information needed to write programs written in the Prime Macro Assembler (PMA) language. It also describes Prime's machine-level debugger, the 64V-mode Prime Symbolic Debugger (VPSD).

- Guide to Prime User Documents — Descriptions of all Prime user documents are listed in this guide. Each document includes a description of the product, printing history, audience level, and more.

- The on-line HELP facility. (Enter HELP DBG.)

- The most recent Software Release Document (MRU) for your release of software.

2

Overview of Debugger Features

INTRODUCTION

Prime's Source Level Debugger provides you with a host of powerful debugging features that help you track down problems in your program execution. You can implement these features through 53 Debugger commands. Although you may never use all of these commands -- some commands are much more fundamental than others -- they are always at your disposal whenever you might need them.

As Chapter 1 pointed out, the Debugger allows you to control, monitor, and manipulate the execution of your program. This chapter overviews the Debugger's features, which can generally be categorized in the following groups:

- Program control
- Data manipulation
- Tracing
- Debugger control
- Information request
- Miscellaneous

Note

Although these Debugger features and their associated commands can be grouped together similarly by function, this book does not necessarily present them according to these groups. Rather, this book presents these features and commands in such a way that you can best learn, understand, and use the Debugger overall. Debugger commands from different groups work in conjunction with each other.

PROGRAM CONTROL FEATURES

Several Debugger features, known as program control features, allow you to use the Debugger to control the execution of your program. For example, you can start and restart execution, suspend execution, execute one statement at a time, and call any procedure, function, or subroutine. These program control capabilities are used to discover where and why your program failed. In other words, the Debugger lets you freeze the action at strategic points in your program's execution so you can see what is really happening — or not happening.

Specifically, here are the program control features along with the program control commands associated with these features:

- Restarting the program — at any time during your debugging session, you may begin execution of the program being debugged, no matter where execution is suspended (RESTART command).
- Setting breakpoints — you may suspend your program's execution at any executable statement or at any entry to or exit from a procedure, function, or subroutine (BREAKPOINT command).
- Using breakpoint action lists — you may specify that one or more Debugger commands be executed each time a breakpoint occurs (BREAKPOINT command).
- Using conditional action lists — you may specify that a breakpoint action list be executed only if a given condition is true (BREAKPOINT and IF commands).
- Setting conditional breakpoints — you may specify that a breakpoint occur only if a given condition is true (BREAKPOINT command).
- Continuing program execution — you may resume program execution after execution has been suspended (CONTINUE command).
- Single stepping — you may execute one or more statements at a time; step across, into, and out of procedures, functions, or subroutines (STEP, STEPIN, IN, and OUT commands).

- Displaying breakpoints and tracepoints — you may display one or more breakpoints or tracepoints (LIST and LISTALL commands).

- Deleting breakpoints and tracepoints -- you may delete one or more breakpoints or tracepoints (CLEAR and CLEARALL commands).

- Transferring control — you may transfer the position at which execution is to resume from one statement in your program to another (GOTO command).

- Calling procedures, functions, and subroutines -- from Debugger command level, you may call a procedure, function, or subroutine, supplying argument lists if needed (CALL command).

- Defining the main program — you may tell the Debugger what program block to recognize as the main program (MAIN command).

- Erasing the call/return stack — you may erase the call/return stack, which is a list of currently active program blocks in your program's execution (UNWIND command).

Figure 2-1 lists the Debugger's program control commands in alphabetical order.

BREAKPOINT	LIST
CALL	LISTALL
CLEAR	MAIN
CLEARALL	OUT
CONTINUE	RESTART
GOTO	STEP
IF	STEPIN
IN	UNWIND

Program Control Commands
Figure 2-1

DATA MANIPULATION FEATURES

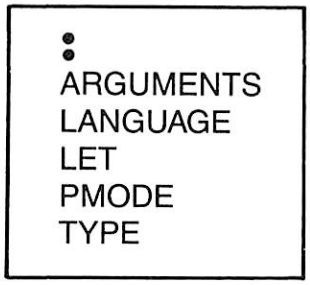
Another very important part of debugging, in addition to controlling program execution, is the ability to examine, evaluate, and modify expressions. These features will be referred throughout this book as data manipulation features. Whenever execution is suspended, you can examine the values of a variable or expression, examine its data type, modify the value of a variable or expression, and use built-in source language functions to evaluate a variable or expression. These data manipulation features are very useful in detecting errors because you

can see what values your data objects hold anywhere in the execution of your program.

Specifically, here are the Debugger's data manipulation features and their related commands:

- Examining variables — the Debugger can display the value of any scalar, array, or structured variable (: command).
- Evaluating expressions — you may evaluate any expression allowed by any source language (: command).
- Assigning values — you may modify the value of a variable (LET command).
- Examining data types — you may examine the data type of a variable or expression (TYPE command).
- Examining the values of arguments — you may examine the values of arguments passed to procedures, functions, or subroutines (ARGUMENTS command).
- Using built-in functions — you may use several source language built-in functions to help evaluate a variable or expression.
- Changing the language of evaluation — you may change the language in which the Debugger evaluates expressions at any time during debugging (LANGUAGE command).
- Setting the print mode — you may explicitly set the print mode to be used to evaluate a variable (PMODE command).

Figure 2-2 lists the Debugger's data manipulation commands in alphabetical order.



Data Manipulation Commands
Figure 2-2

TRACING FEATURES

As its name implies, the Debugger's tracing features offer you the ability to trace the progress of your program's execution from beginning to end. For example, you can have trace messages displayed at strategic points during program execution and you can trace the value of a variable as it changes throughout program execution.

Specifically, here are the Debugger's tracing features and related commands:

- Setting tracepoints -- you may specify that a trace message be displayed at the execution of a specified statement or at the entry to or exit from a procedure, function, or subroutine (TRACEPOINT command).

- Tracing values -- you may specify that a message be displayed whenever the value of a specified variable changes during execution. This message tells you the old value, the new value, and location in your program where the change was detected (WATCH, VTRACE, UNWATCH, and WATCHLIST commands).

- Tracing statements -- you may specify that a trace message be displayed prior to the execution of each statement and/or each labelled statement (STRACE command).

- Tracing at entries and exits -- you may specify that a message be displayed each time any procedure, function, or subroutine is called or returns (ETRACE command).

- Tracing the currently active program blocks -- the Debugger can display a list of program block calls that are currently active throughout your program's execution (TRACEBACK command).

Figure 2-3 lists the Debugger's tracing commands in alphabetical order.

ETRACE	UNWATCH
STRACE	VTRACE
TRACEBACK	WATCH
TRACEPOINT	WATCHLIST

Tracing Commands
Figure 2-3

DEBUGGER CONTROL FEATURES

Another group of features can be generally categorized as Debugger control features. Debugger control commands make the Debugger interpret or treat information in a particular way.

Specifically, here are the Debugger control features and related commands:

- Changing the evaluation environment -- you may change the environment (program block) in which the Debugger evaluates expressions (ENVIRONMENT and ENVLIST commands).
- Using the Debugger's command line editor -- you may edit the most recent command given by invoking the Debugger's command line editor (RESUBMIT command).
- Displaying action lists and macros -- you may display the contents of breakpoint action lists and macro command lists immediately prior to their execution (ACTIONLIST command).
- Changing special symbols -- you may change the Debugger's interpretation of certain characters that have special meaning to the Debugger (PSYMBOL and SYMBOL command).

Figure 2-4 lists the Debugger control commands in alphabetical order.

ACTIONLIST	PSYMBOL
ENVIRONMENT	RESUBMIT
ENVLIST	SYMBOL

Debugger Control Commands
Figure 2-4

INFORMATION REQUEST FEATURES

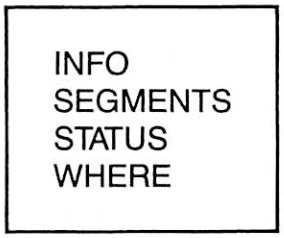
There are four Debugger commands that request information. These information request features and their related commands are:

- Displaying the execution environment pointer — you may request that the Debugger display the location of the execution environment pointer, which is the place where program execution is to resume (WHERE command).

- Displaying attributes about a program block or statement -- you may request that the Debugger display certain information about a program block or statement (INFO command).

- Displaying a list of segments -- you may request that the Debugger display a list of segments in memory that are currently in use (SEGMENTS command).

- Displaying the status of your debugging environment -- you may request that the Debugger display information about your current debugging environment (STATUS command).



Information Request Commands
Figure 2-5

MISCELLANEOUS FEATURES

The Debugger has many useful features that really do not fit into any logical category, so they are referred to as miscellaneous features.

Specifically, here are the Debugger's miscellaneous features and related commands:

- Examining the source file -- you may look at, but cannot change, your source files while debugging (SOURCE command).

- Creating Debugger command macros -- you may create a macro to take the place of one or more Debugger commands (MACRO and MACROLIST commands).

- Saving breakpoints, tracepoints, and macros -- you may save all of your breakpoints, tracepoints, and macros in PRIMOS files, then use them again in future debugging sessions (SAVESTATE and LOADSTATE commands).

- Getting help -- you may ask the Debugger for help in understanding command syntax definitions (HELP command).

- Entering PRIMOS commands -- you may enter and execute certain PRIMOS commands from Debugger command level (! command).

- Suspending the debugging session -- you may temporarily suspend your debugging session and return to PRIMOS command level (PAUSE command).

- Entering the machine level debugger -- you may invoke the 64V-mode Prime Symbolic Debugger (VPSD) from Source Level Debugger command level (VPSD command).

- Repeating Debugger commands -- you may repeat the execution of a Debugger command (* and AGAIN commands).

- Leaving the Debugger -- you may leave the Source Level Debugger and return to PRIMOS command level (QUIT command).

Figure 2-6 lists the Debugger's miscellaneous commands in alphabetical order.

AGAIN	QUIT
CMDLINE	SAVESTATE
HELP	SOURCE
LOADSTATE	VPSD
MACRO	!
MACROLIST	*
PAUSE	

Miscellaneous Commands
Figure 2-6

PART II

Using the Debugger

3
Getting Started

Commands discussed in this chapter:		
DBG	BREAKPOINT	LET
RESTART	CONTINUE	WATCH
CMDLINE	:	HELP
SOURCE	TYPE	QUIT

INTRODUCTION

Now that you know what the Source Level Debugger is and what it can do, you are ready to invoke the Debugger and begin using it.

This chapter shows you how to invoke the Debugger and use some of the Debugger's fundamental commands. Specifically, this chapter teaches you how to:

- Invoke the Debugger from PRIMOS command level with the DBG command.
- Execute your program from within the Debugger with the RESTART and CMDLINE commands.
- Examine your source program with the SOURCE command.
- Suspend program execution with the BREAKPOINT command.
- Continue program execution with the CONTINUE command.
- Examine and modify program data with the : (evaluation), TYPE, and LET commands.
- Trace the changing value of a variable through the execution of your program with the WATCH command.

- Get help in understanding Debugger command syntax with the HELP command.

- Leave the Debugger and return to PRIMOS with the QUIT command.

After reading this chapter and learning the commands listed above, you should be able to debug simple programs. However, it is recommended that you keep reading and learn the vast assortment of powerful debugging capabilities that are presented in the remainder of this book.

INVOKING THE DEBUGGER

In order to use the Debugger with programs written in any of Prime's seven supported high-level languages, you must compile and load your program successfully. As was pointed out in Part I, the Debugger uncovers program errors at execution time or runtime, while the program executes. Therefore, an executable file or runfile must be created before debugging can take place.

There are four simple steps to invoking the Debugger:

1. Compile your program with the -DEBUG option.

2. Load your program's object (binary) file.

3. Execute your program.

4. Enter the Debugger with the DBG command.

Compiling with -DEBUG Option

You must tell the compiler that you intend to use the Debugger by entering the -DEBUG compile-time option on the command line. For example, if you want to debug a FORTRAN 77 program named TEST.F77, you would enter this command:

```
OK, F77 TEST -DEBUG
```

Similarly, if you were compiling a Pascal program named TEST.PASCAL, you would enter:

```
OK, PASCAL TEST -DEBUG
```

The -DEBUG option causes the compiler to generate the special Debugger information contained in the symbol table, which was described in Chapter 1.

The -DEBUG option, which can be abbreviated -DE, is used with all seven of the supported Prime languages -- FORTRAN IV, FORTRAN 77, Pascal, PL/I Subset G, COBOL 74, RPG II V-mode, and C.

Note

At Rev. 18, Prime employed new, more efficient file naming conventions called the suffix conventions. The suffix conventions identify the source file with a compiler suffix, the object (binary) file with a .BIN suffix, and the executable file with a .SEG suffix. The suffix conventions are explained in the Prime User's Guide. The old-style prefix file naming conventions may still be used.

For your convenience, Table 3-1 lists the PRIMOS compile commands, source file suffixes, and language libraries for all seven languages.

Table 3-1
Compile Commands, Source File Suffixes, and Libraries

Language	Compile Command	Compiler Source File Suffix	Language Library
FORTRAN IV	FTN	.FTN	none
FORTRAN 77	F77	.F77	none
Pascal	PASCAL	.PASCAL	PASLIB
PL/I Subset G	PL1G	.PL1G	PL1GLB
COBOL 74	CBL	.CBL	CBLLIB
RPG II V-mode	VRPG	.RPG	VRPGLB
C	CC	.CC	CCLIB

Loading Your Program

There is no change in the way a program is loaded. Programs that are compiled with the -DEBUG option are loaded the same way as those that are not. Here is an example of loading a COBOL 74 program:

```
OK, SEG -LOAD          Enter SEG's load subprocessor.
$ LOAD TEST            Load the program's object file.
$ LIBRARY CBLLIB       Load the COBOL library.
$ LIBRARY              Load the standard system libraries.
LOAD COMPLETE          Loader indicates load is complete.
$ QUIT                 Save executable file and return to PRIMOS.
OK,
```

See Table 3-1 for the appropriate language library to load.

Executing Your Program

When you execute your program in the usual way, one of four things could happen:

1. Your program runs until completion and produces correct results.

2. Your program runs until completion and produces incorrect results or no results at all.

3. Your program terminates abnormally, displaying an error message and returning to PRIMOS command level.

4. Your program does not terminate.

If you are fortunate enough to have case 1, then you probably will not need the Debugger. Otherwise, in cases 2, 3, and 4, the Debugger can be very useful when you ask "What happened?". (See Appendix I for sample strategies in debugging.)

Entering the Debugger

Enter the Debugger from PRIMOS command level by issuing the DBG command. The format of the DBG command is:

```
DBG program-name [option-1 [option-2 ...]]
```

The program-name is the name of the program file you want to debug. The program-name is an executable (SEG) file. For example, if your executable file were called TEST.SEG, you would enter this command:

```
OK, DBG TEST
```

It is not necessary, though acceptable, to enter TEST.SEG.

option-1, option-2, etc., are optional command line parameters, which you may specify to make the Debugger perform or not perform certain functions during its operations. (These options are described in Chapter 13.)

When you enter the DBG command, the Debugger reads the program and symbol table into memory. Then, an identification message, which includes the software release number, is displayed at your terminal. For example:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

>
```

The right angle bracket shown above is the Debugger's prompt symbol. When this symbol appears, it means that you have entered the Debugger and that the Debugger is ready for command input. Debugger commands are given at this prompt.

EXECUTING YOUR PROGRAM WITHIN DEBUGGER

In trying to find out what went wrong at runtime, you can use the Debugger to manipulate the execution of your program. You can easily activate the execution of your program from within the Debugger using the RESTART command.

The RESTART command, abbreviated RST, starts or "restarts" program execution at any point it is given within the Debugger.

A Program Example

To demonstrate the RESTART command and the other commands in this chapter, the following PL/I Subset G program is used. At the end of the chapter, a Pascal program and a FORTRAN IV program, each having runtime errors, will demonstrate how the Debugger commands that you are about to learn can solve runtime problems.

The PL/I-G program follows:

```
TEST : PROCEDURE;
  DECLARE (X, Y, Z) FIXED BIN(15);
  DECLARE (A, B, C) FIXED DEC(4, 2);
    X = 5;
    Y = 3;
    Z = X + Y;
    PUT SKIP LIST('The integer sum is', Z);
    PUT SKIP;
    A = 5.5;
    B = 4.3;
    C = A + B;
    PUT SKIP LIST('The decimal sum is', C);
    PUT SKIP;
    X = Z;
    PUT SKIP LIST('The integer sum is now', X + Y);
    PUT SKIP;
END TEST;
```

This program adds two integers, adds two floating point numbers, changes the value of one of the integers, and adds the integers again. When the program executes, the three sums are displayed on the terminal. The output looks like this:

```
The integer sum is          8

The decimal sum is      9.80

The integer sum is now            11
```

Using RESTART

The following debugging session illustrates use of the RESTART command with the PL/I-G program. The RESTART command is given twice — immediately upon entering the Debugger and immediately after program execution is complete:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> RESTART

The integer sum is          8

The decimal sum is      9.80

The integer sum is now             11

**** Program execution complete.
> RESTART

The integer sum is          8

The decimal sum is      9.80

The integer sum is now             11

**** Program execution complete.
>
```

In the example above, notice how the program's output is displayed on the terminal, as it should be. If the program's input and output involved data files, then normal I/O to and from data files would take place as well, without interference from the Debugger. Also notice the convenient "Program execution complete" message.

For more information on the RESTART command, see Chapter 5.

Executing with Command Line Arguments Using CMDLINE

Sometimes the execution of your program may require command line arguments. Suppose the PL/I-G program TEST had a command line argument named MYFILE. You would normally execute the program this way:

```
OK, SEG TEST MYFILE
```

But since the argument MYFILE cannot be entered on the DBG command line, the Debugger's CMDLINE command, abbreviated CL, allows you to enter your arguments from within the Debugger.

Here is an example of CMDLINE:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> CMDLINE
Enter command line:
MYFILE
>
```

For more information on the CMDLINE command, see Chapter 13.

EXAMINING YOUR SOURCE PROGRAM

During most of your debugging sessions, you will want to examine the contents of your source program. Examining your program is necessary for determining the strategic locations in your program where certain Debugger features are to be used.

You can easily examine your program on your terminal from within the Debugger using the SOURCE command. SOURCE saves you the trouble of dealing with hard (paper) copies. The SOURCE command, which displays the contents of a file, works like Prime's line EDITOR. The SOURCE command, followed by certain EDITOR subcommands, makes you feel as though you are indeed using the EDITOR. (For more information on EDITOR, see the New User's Guide to EDITOR and RUNOFF.)

The format of the SOURCE command, abbreviated SRC, is:

```
SOURCE source-command [argument]
```

The source-command is any EDITOR subcommand that can be used with SOURCE. A subset of 14 EDITOR subcommands that examine — but do not modify — a file is provided. These subcommands, along with brief descriptions, are listed in Table 3-2. (See also the New User's Guide to EDITOR and RUNOFF.)

The argument is an EDITOR source subcommand object such as a line number or text string. It may or may not be used, depending on which EDITOR source subcommand you specify.

Table 3-2

Source EDITOR Subcommands
(Abbreviations are underlined.)

Subcommand	Description
TOP	Position line pointer to top of file.
BOTTOM	Position pointer to bottom of file.
BRIEF	Don't print target lines of FIND, LOCATE, POINT, and NEXT operations.
VERIFY	Print target lines of FIND, LOCATE, POINT, and NEXT operations.
PRINT	Print one or more lines.
WHERE	Print current line number.
POINT	Position to specific line.
NEXT	Move line pointer forward or backward.
MODE	Set edit mode; the only mode implemented is NUMBER/NNUMBER.
LOCATE	Locate line with the specified text string.
FIND	Locate line with the specified text string beginning in a given column.
PSYMBOL	Print character symbols; see also Debugger PSYMBOL command (Chapter 4).
SYMBOL	Set character symbol; see also Debugger SYMBOL command (Chapter 4).
*	Repeat command line; see also Debugger REPEAT (*) command (Chapter 11).

Note

See also the New User's Guide to EDITOR and RUNOFF.

A typical SOURCE command line, without an argument, looks like this:

```
> SOURCE TOP
```

The command shown above will take the EDITOR line pointer to the top of your program file, just above the first line.

A typical SOURCE command line with an argument looks like this:

```
> SOURCE PRINT 23
```

The command shown above will display 23 lines of your program file, beginning with the current line.

The following example demonstrates the SOURCE command used with the PRINT, NEXT, LOCATE, and POINT subcommands:

```
> SOURCE PRINT 5
   1: TEST : PROCEDURE;
   2:    DECLARE (X, Y, Z) FIXED BIN(15);
   3:    DECLARE (A, B, C) FIXED DEC(4, 2);
   4:      X = 5;
   5:      Y = 3;
> SOURCE NEXT
   6:      Z = X + Y;
> SOURCE NEXT 3
   9:      A = 5.5;
> SOURCE LOCATE X + Y
  15:      PUT SKIP LIST('The integer sum is now', X + Y);
> SOURCE POINT 4
   4:      X = 5;
>
```

Notice how source file line numbers appear to the left of the source code. These line numbers are used in many of the Debugger's functions.

For more information on the SOURCE command, including three additional source subcommands (EX, NAME, and RENAME), see Chapter 11.

SUSPENDING PROGRAM EXECUTION

One of the most fundamental and useful Debugger commands is BREAKPOINT, abbreviated BRK. The BREAKPOINT command can suspend execution almost anywhere in your program. The suspension of execution, commonly called breaking, allows you to examine data at strategic points of the execution — at the beginning or end of a loop, for example. Examining data while execution is frozen is one of the fundamental ways of finding out why your program failed. A breakpoint is like a snapshot of a moving object. You can see what's really happening while the action is frozen.

The BREAKPOINT command, abbreviated BRK, has many powerful capabilities. All of these capabilities are discussed in Chapter 5. For now, simple examples of BREAKPOINT will be used to get you started.

Breakpoints are set on any executable statement in your program -- statements that perform some action. The BREAKPOINT command is followed by a breakpoint-identifier, which usually specifies the number of the line in your program on which you want to break. For example:

```
> BREAKPOINT 6
```

The command shown above will cause the Debugger to suspend execution immediately before source line number 6 of your program. (You can determine a source line number by examining your source file with the SOURCE command.)

Line 6 might contain an executable statement such as:

```
Z = X + Y;
```

Any attempt to break on non-executable syntax, such as a data declaration, a comment, or a DATA DIVISION line in COBOL, will generate the following error message:

```
No such statement.
```

The following example sets a breakpoint on line 12 of the sample PL/I-G program, then executes the program:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> SOURCE PRINT 23
    1: TEST : PROCEDURE;
    2:   DECLARE (X, Y, Z) FIXED BIN(15);
    3:   DECLARE (A, B, C) FIXED DEC(4, 2);
    4:     X = 5;
    5:     Y = 3;
    6:     Z = X + Y;
    7:     PUT SKIP LIST('The integer sum is', Z);
    8:     PUT SKIP;
    9:     A = 5.5;
   10:     B = 4.3;
   11:     C = A + B;
   12:     PUT SKIP LIST('The decimal sum is', C);
   13:     PUT SKIP;
   14:     X = Z;
   15:     PUT SKIP LIST('The integer sum is now', X + Y);
   16:     PUT SKIP;
   17: END TEST;
BOTTOM
> BREAKPOINT 12
> RESTART

The integer sum is          8

**** breakpointed at TEST\12
>
```

In the example shown above, execution stops just before line 12, and the information on line 12 is not output. Notice the breakpoint message that tells you the name of the program block (TEST) and the line number on which the breakpoint occurred (12).

Note

The Debugger allows you to display and delete breakpoints. For complete information, see Chapter 5.

CONTINUING PROGRAM EXECUTION

Whenever program execution has been suspended and you want to resume execution, simply give the CONTINUE command, abbreviated C.

The following example uses a CONTINUE command to resume execution of the PL/I-G program after it has been suspended with a breakpoint:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT 12
> RESTART

The integer sum is           8

**** breakpointed at TEST\12
> CONTINUE

The decimal sum is      9.80

The integer sum is now            11

**** Program execution complete.
>
```

For more information on the CONTINUE command, see Chapter 5.

EXAMINING AND MODIFYING PROGRAM DATA

A breakpoint is usually set to allow examination and modification of program data. This is known as data manipulation. There are several data manipulation commands. Three of the most common are:

- : (evaluation command)
- TYPE
- LET

This chapter gives only simple examples of the Debugger's data manipulation features. (For a full explanation and demonstration of these features, see Chapter 6.)

The : Command

The : command, which is a colon on your keyboard, is used to examine the value of any given variable or expression while program execution is suspended. Commonly known as the evaluation command, : lets you know that something has gone wrong with the logic of your program by discovering unusual values assigned to variables or expressions.

To evaluate a variable or expression, specify the variable or expression after the : command. The colon must be followed by a space. For example:

```
> : X

> : X + Y
```

The Debugger outputs the value.

The following example breaks two times and evaluates some expressions while execution is suspended:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT 7
> BREAKPOINT 15
> RESTART

**** breakpointed at TEST\7
> : Z
Z = 8
> CONTINUE

The integer sum is          8

The decimal sum is       9.80

**** breakpointed at TEST\15
> : X * Z + Y
67
> CONTINUE

The integer sum is now             11

**** Program execution complete.
>
```

For more information on the : command, see Chapter 6.

The TYPE Command

As its name implies, the TYPE command tells you the data type of any given variable or expression. The TYPE command is useful for detecting data type incompatibility. Often when a program has many variables and expressions of many different complex data types, a type mismatch could cause the program to fail.

To evaluate the data type of a variable or expression, specify the variable or expression following the TYPE command. The Debugger outputs the data type.

The following example uses the TYPE command twice:

```
> BREAKPOINT 7
> BREAKPOINT 12
> RESTART

**** breakpointed at TEST\7
> TYPE Z
fixed binary (15) automatic
> CONTINUE

The integer sum is          8

**** breakpointed at TEST\12
> TYPE A + B
fixed decimal (4)
>
```

For more information on the TYPE command, see Chapter 6.

The LET Command

The LET command allows you to assign a new value to any variable. Assigning new values to variables lets you see what would happen to your program execution, given these new values. With LET, you can feed your program correct or incorrect values, and then study the output.

When you use the LET command, an expression is assigned to a variable with an equals sign (=). The variable is on the left-hand side and the expression is on the right.

Here is an example that uses the LET command:

```
> BREAKPOINT 5
> BREAKPOINT 14
> RESTART

**** breakpointed at TEST\5
> : X
X = 5
> LET X = 10
> : X
X = 10
> CONTINUE

The integer sum is          13

The decimal sum is       9.80

**** breakpointed at TEST\14
> : X
X = 10
> LET X = Z + X
> : X
X = 23
> CONTINUE

The integer sum is now              16

**** Program execution complete.
>
```

For more information on the LET command, see Chapter 6.

TRACING A VALUE

As was pointed out in Chapter 2, the Debugger has certain tracing features. One useful tracing feature is value tracing, enabled with the WATCH command. Value tracing simply means watching a variable as its value changes during program execution. Tracing a variable's value can help you pinpoint the spot where an unusual value is assigned. Value tracing, in general, is more useful for debugging small programs.

To trace the value of one or more variables, specify those variables following the WATCH command, which is abbreviated WA. When you specify two or more variables, the variables must be separated by commas.

This chapter gives simple examples of value tracing. (For complete information on the Debugger's tracing capabilities, see Chapter 8.)

In the following example, the variable X is watched during the execution of the PL/I-G program:

```
> WATCH X
> RESTART
The value of TEST\X has been changed at TEST\5
  from 0
  to   5

The integer sum is           8

The decimal sum is       9.80
The value of TEST\X has been changed at TEST\15
  from 5
  to   8

The integer sum is now              11

**** Program execution complete.
>
```

In the example above, notice that the variable name is given along with the source line number, the old value, and the new value.

Here is another example that traces the changing value of a Pascal array of characters. The Pascal program is displayed in the Debugger session:

```
> SOURCE PRINT 23
    1: PROGRAM Animal;
    2: VAR
    3:   A : ARRAY[1..3] OF CHAR;
    4: BEGIN
    5:   A := 'CAT';
    6:   A := 'DOG';
    7:   A := 'RAT';
    8:   WRITELN('The value of A is ', A)
    9: END.
BOTTOM
> WATCH A
> RESTART
The value of ANIMAL\A has been changed at ANIMAL\6
  from ''
  to   'CAT'
The value of ANIMAL\A has been changed at ANIMAL\7
  from 'CAT'
  to   'DOG'
The value of ANIMAL\A has been changed at ANIMAL\8
  from 'DOG'
  to   'RAT'
The value of A is RAT

**** Program execution complete.
>
```

GETTING HELP WITH DEBUGGER COMMANDS

If you need help in remembering and understanding the functions of any Debugger command while you are using the Debugger, the HELP command can provide answers. The HELP command can display a list of all Debugger commands, the syntax of any Debugger command, a list of all syntax symbols used in Debugger command syntax descriptions, or the definition of a command syntax symbol.

The format of the HELP command is:

```
HELP [ -LIST
       -SYM_LIST
       command-name
       syntax-symbol ]
```

The command-name is the name or abbreviation of any Debugger command. When you give a command name, its command line syntax is displayed:

```
> HELP TYPE
TYPE  <expression>
> HELP WATCH
WAtch  <variable-list>
```

In the examples shown above, capital letters in command names identify abbreviations. Words enclosed in angle brackets are command syntax symbols.

The syntax-symbol is any symbol, such as variable-list, that is used in command syntax descriptions. When you give a syntax symbol, a definition of that symbol is displayed:

```
> HELP EXPRESSION
<EXPRESSION>:
any valid expression in the default evaluation language
>
```

If you specify -LIST, a list of all Debugger commands is displayed:

```
> HELP -LIST
```

If you specify -SYM_LIST, a list of all Debugger syntax symbols used in Debugger command syntax descriptions is displayed:

```
> HELP -SYM_LIST
```

If you just enter the command HELP by itself with no arguments, the most recent documentation and the HELP command syntax are displayed:

```
> HELP
For help, refer to DOC4033-193 Source Level Debugger User's Guide.

HELP -LIST                prints a list of all DBG commands
HELP -SYM_LIST            prints a list of all syntax symbols
HELP <command-name>       prints the syntax of <command-name>
HELP <syntax-symbol>      prints the definition of <syntax symbol>
>
```

LEAVING THE DEBUGGER

Whenever you are finished using the Debugger and want to return to PRIMOS command level, enter the QUIT command.

The format of the QUIT command, abbreviated Q, is:

```
QUIT
```

Here is an example of the QUIT command:

```
> QUIT
OK,
```

DEBUGGING EXAMPLES

In the remainder of this chapter, some of the Debugger commands that you just learned will be used to debug two programs. The first program is written in Pascal. The second program is written in FORTRAN IV. Both have logic problems that prevent the output of correct data.

Pascal Sample Program

Consider the following Pascal program:

```
 1: PROGRAM Loop;
 2: VAR
 3:    I : INTEGER;
 4:    CH : CHAR;
 5: BEGIN
 6:    CH := 'A';
 7:    WRITELN('The value of CH is initialized to ', CH);
 8:    FOR I := 1 TO 5 DO
 9:        CH := (SUCC(CH));
10:        WRITELN('The value of CH is now ', CH)
11: END.
```

(Debugger source line numbers have been added for your convenience.)

This program is supposed to write out six consecutive capital letters, A through F. The program compiles and loads successfully, but it generates incorrect output:

```
OK, PASCAL LOOP -DEBUG
[PASCAL Rev. 19.2]
0000 ERRORS (PASCAL-REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD LOOP
$ LIBRARY PASLIB
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
The value of CH is initialized to A
The value of CH is now F
OK,
```

You wonder why only the A and F are written out and suspect the problem is within the FOR loop. You place a BREAKPOINT on line 10, the WRITELN statement, execute the program with RESTART, and examine the values of CH and I while execution is suspended:

```
OK, DBG LOOP

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT 10
> RESTART
The value of CH is initialized to A

**** breakpointed at LOOP\10
> : CH
CH = 'F'
> : I
I = 5
> CONTINUE
The value of CH is now F

**** Program execution complete.
>
```

Now you've discovered the problem. You see that the first time line 10 is executed is when the value of CH becomes F and the FOR loop completes execution. Only line 9 goes through the loop. To be sure, you can trace the value of CH as it changes throughout the program:

```
> WATCH CH
> RESTART
The value of LOOP\CH has been changed at LOOP\7
  from 'F'
  to   'A'
The value of CH is initialized to A
The value of LOOP\CH has been changed at LOOP\9
  from 'A'
  to   'B'
The value of LOOP\CH has been changed at LOOP\9
  from 'B'
  to   'C'
The value of LOOP\CH has been changed at LOOP\9
  from 'C'
  to   'D'
The value of LOOP\CH has been changed at LOOP\9
  from 'D'
  to   'E'
The value of LOOP\CH has been changed at LOOP\10
  from 'E'
  to   'F'

**** breakpointed at LOOP\10
> QUIT
OK,
```

Sure enough, the value of CH changes the way it should, but the WRITELN statement is not included in the loop. You realize that two or more statements in Pascal -- a compound statement -- require the keyword delimiters BEGIN and END. You correct the FOR Loop in your program:

```
FOR I  := 1 TO 5 DO
  BEGIN
    CH := (SUCC(CH));
    WRITELN('The value of CH is now', CH)
  END
```

With this change, you compile, load, and execute the program again and get correct output:

```
OK, SEG LOOP
The value of CH is initialized to A
The value of CH is now B
The value of CH is now C
The value of CH is now D
The value of CH is now E
The value of CH is now F
OK,
```

FORTRAN Sample Program

Consider the following FORTRAN IV program:

```
 1: C   Print the squares of the numbers 1 through 10.
 2: C
 3:         DO 100 I = 1, 10
 4:         J = SQUARE (I)
 5:         WRITE (1, 80) J
 6: 80      FORMAT (I5)
 7: 100     CONTINUE
 8:         CALL EXIT
 9:         END
10:
11:         INTEGER FUNCTION SQUARE (I)
12:         SQUARE = I ** 2
13:         RETURN
14:         END
```

(Debugger source line numbers have been added for your convenience.)

This program is supposed to print out the squares of the numbers 1 through 10. The program compiles and loads successfully, but when you execute it you get incorrect output:

```
OK, FTN SQUARES -64V -DEBUG
0000 ERRORS [<.MAIN.>FTN-REV19.2]
0000 ERRORS [<SQUARE>FTN-REV19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD SQUARES
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
OK,
```

Note

You must include the -64V option with the -DEBUG option when you debug FORTRAN IV programs.

You get into the Debugger to try to determine what went wrong. You place a breakpoint following the call to function SQUARE in order to look at the returned value:

```
OK, DBG SQUARES

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT 5
>
```

You begin program execution with the RESTART command, and execution breaks at line 5:

```
> RESTART

**** breakpointed at $MAIN\5
>
```

(The term $MAIN is supplied by the Debugger when a FORTRAN main program block is not named.) At this breakpoint, you inspect the variables I and J:

```
> : I
I = 1
> : J
J = 0
>
```

You see that the value of I is correct. (I is the control variable for the DO loop, and this is the first iteration through the loop.) However, the value of J is wrong. Seeing this error, you decide to suspend program execution earlier in the loop, inside function SQUARE, immediately after the function value is computed:

```
> BREAKPOINT SQUARE\13
> CONTINUE
   0

**** breakpointed at SQUARE\13
>
```

Note

The BREAKPOINT command line shown above specifies the name of the function (SQUARE) as well as the source line number (13). Sometimes the name of a program block must be specified in this way. For more information, see Chapters 4 and 5.

You look at the value of the returned function:

```
> : SQUARE
SQUARE = 4
>
```

You find the function value is correct, and you continue execution until it breaks on line 5 again:

```
> CONTINUE

**** breakpointed at $MAIN\5
>
```

Once again, looking at the returned value:

```
> : J
J = 0
>
```

you find it wrong. You suspect that the data types of the SQUARE function mismatch between the FORTRAN main program and SQUARE. This is confirmed by examining the data types:

```
> TYPE SQUARE
entry constant (real*4 function)
> TYPE SQUARE\SQUARE
integer*2 static
>
```

You see that the data types do indeed mismatch -- REAL*4 vs. INTEGER*2. Upon correcting, recompiling, and reloading the program, you find that it works:

```
OK, DBG SQUARES

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> RESTART
    1
    4
    9
   16
   25
   36
   49
   64
   81
  100

EXIT.  Program exit from $MAIN\8 ($100+1).
> QUIT
OK,
```

4

Conventions, Terms, and Concepts

Commands discussed in this chapter:

PSYMBOL SYMBOL

INTRODUCTION

This chapter defines various Debugger conventions, terms, and concepts. Before you learn about the other Debugger features that are presented in the remainder of this book, you should read this chapter. The conventions, terms, and concepts that are outlined in this chapter are related to all Debugger functions.

The following topics are covered in this chapter:

- Debugger command format conventions
- Program blocks
- Environments
- Language of evaluation
- Activations
- Active program blocks
- Identifying variables
- Identifying statements
- Special characters
- Special symbols (PSYMBOL and SYMBOL commands)

DEBUGGER COMMAND FORMAT CONVENTIONS

General Format

Debugger commands are entered at the prompt character >, the right angle bracket. The general command format is:

```
> command-name [modifier] [argument-1 [argument-2 ...]]
```

The command-name is the name of the Debugger command. The modifier is an optional command modifier, such as ON or OFF. argument-1, argument-2, etc., represent one or more optional command arguments, such as an expression, symbol name, or statement identifier.

The exception to the format shown above is the evaluation command (:). For more information on the evaluation command, see Chapter 6.

Note

You may enter command line text in uppercase or lowercase characters, since the Debugger maps lowercase characters to uppercase, except characters appearing within paired quotes. A line beginning with /* is interpreted as a comment and is ignored by the Debugger.

Multiple Commands per Line

You may enter multiple Debugger commands on a single command line by placing the Debugger's separator character (a semicolon) between commands. For example:

```
> BREAKPOINT 22; LET I = 8; RESTART
```

Commands are executed from left to right. If any command causes an error, the text to the right of the erroneous command is ignored.

The semicolon separator character is interpreted literally as a semicolon when it appears within a pair of quotes, within the square brackets of an action list, or if it is preceded by an escape character, which is an up-arrow or circumflex.

PROGRAM BLOCKS

In its operations, the Debugger uses the names of program blocks to identify a variable or statement. The term program block is used throughout this book as a universal language-independent definition of a main program, procedure, function, subroutine, BEGIN block, or any other program unit in any of Prime's languages.

Brief definitions of what program blocks are in the context of the seven supported languages follow.

FORTRAN IV and FORTRAN 77 Program Blocks

A FORTRAN program block is a main program, subroutine, or function. A main program is identified by its name, if a name has been provided in a FORTRAN PROGRAM statement. If a PROGRAM statement is not used, the name $MAIN is provided. FORTRAN subroutines and functions are identified by their respective names.

PL/I Subset G Program Blocks

A PL/I Subset G block is a procedure block or a BEGIN block. The procedure block can be the main program or any other procedure.

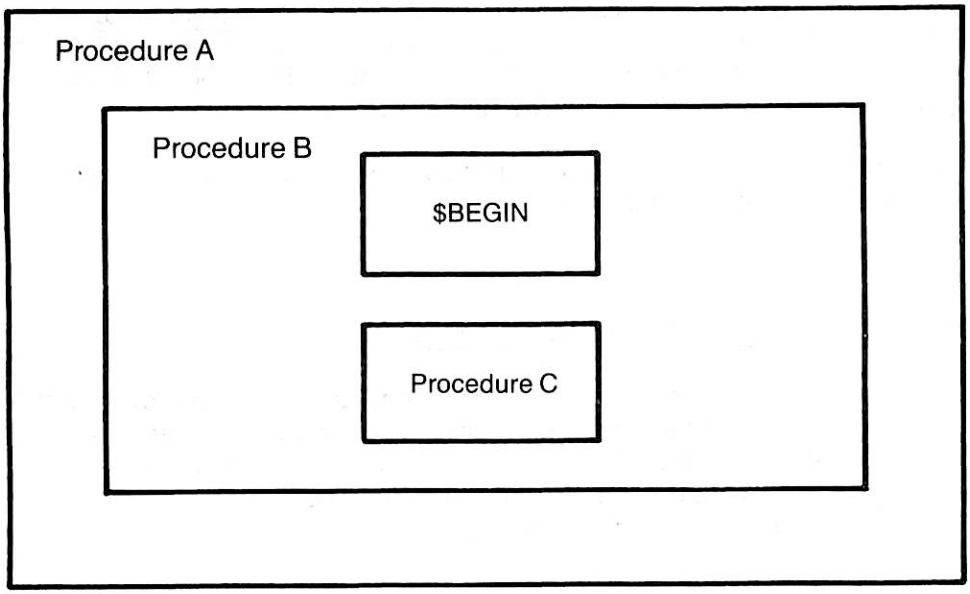
A procedure block is identified by its procedure name. BEGIN blocks, which are delimited by BEGIN and END statements, are identified by the Debugger-supplied name, $BEGIN, followed by the source line number on which the BEGIN block starts. For example, $BEGIN38 would identify a BEGIN block that starts on source line number 38.

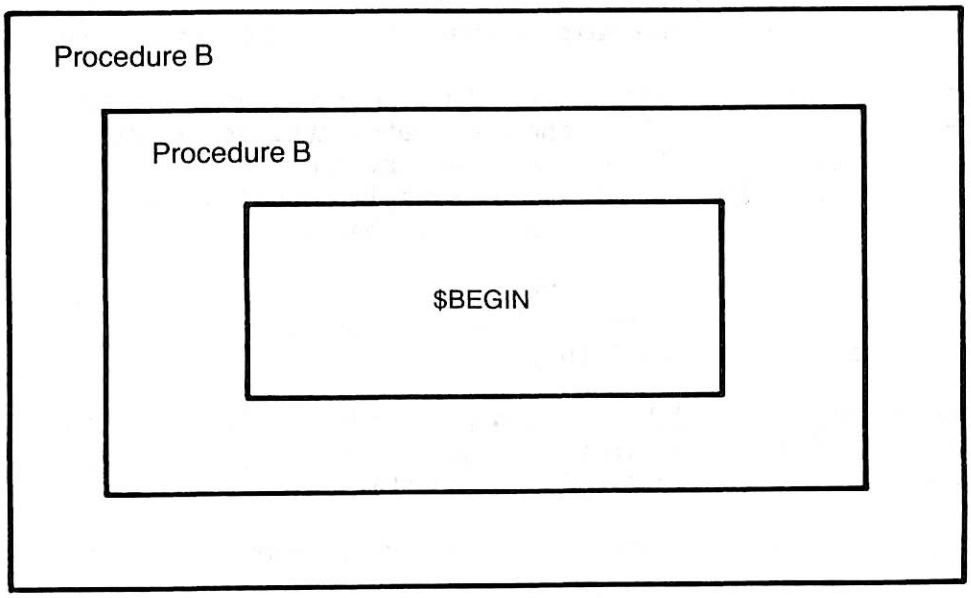
When two or more procedures or BEGIN blocks have identical names, you must identify the nested program blocks in a way that will differentiate them for the Debugger.

Consider Figure 4-1. Although it may be undesirable to use the program block naming scheme shown in this figure, this diagram demonstrates how the program block names are uniquely identified.

The program blocks in Figure 4-1 are uniquely identified as follows:

- External procedure A is identified by the reference A.
- Internal procedure B inside external procedure A is identified by the reference A.B.
- Internal BEGIN block inside internal procedure A.B is identified by A.B.$BEGIN38. (The 38 is a sample line number.)
- Internal procedure C may be referred to as C, A.C, B.C, or as A.B.C because there are no other procedures named C.





Uniquely defined Program Blocks
Figure 4-1

- External procedure B is identified by the reference B.
- Internal procedure B inside external procedure B is identified by the reference B.B.
- Internal BEGIN block inside internal procedure B.B is identified by the reference B.B.$BEGIN23. (The 23 is a sample line number.)

The Debugger will display the following error message when a program block name is ambiguously identified:

```
ambiguous block reference
```

The Debugger always displays program block names in fully qualified form.

Pascal Program Blocks

A Pascal program block is a main program, procedure, or function. It is identified by the name provided in the PROGRAM, PROCEDURE, or FUNCTION statement. If the main program name is not given in a PROGRAM statement, then the name $$MAIN$$ is provided. The rules for identifying names of Pascal nested procedures and functions are the same as PL/I-G. (See the PL/I-G discussion and Figure 4-1 in the previous section.)

COBOL 74 Program Blocks

A COBOL 74 program block is one complete program. It is identified by the name specified in the PROGRAM-ID statement.

RPG II Program Blocks

An RPG II program block is a main program or a subroutine. The Debugger identifies the main program with the name RPG$MAIN. A subroutine is identified by the name specified in the BEGSR statement.

C Program Blocks

A C program block is a function that can be either the main program or any other function. All functions are identified by their function names, which are supplied by the programmer.

Debugger-defined Blocks

The Debugger defines two program block names that may be used to identify variables uniquely. They are:

- $DBG
- $EXTERNAL

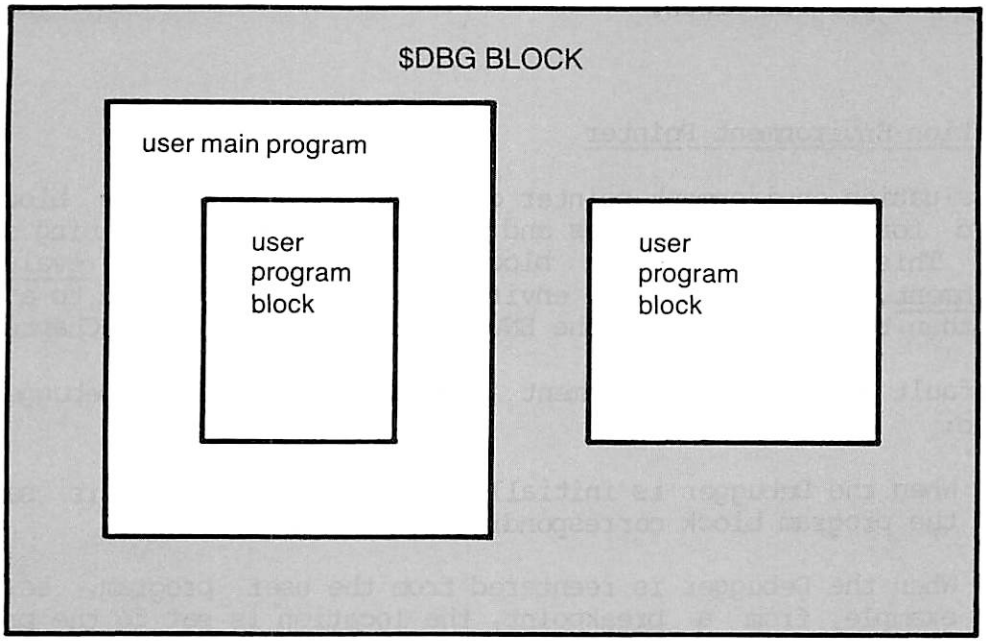
$DBG Program Block: During every debugging session, the Debugger creates a program block called $DBG. This block, which is invisible or "imaginary" to users, is global to any program that you debug. That is, anything that the Debugger defines within the $DBG block, such as a Debugger-defined variable or built-in language function, is visible within all user program blocks.

The Debugger defines three special Debugger-defined variables within $DBG. They are $MR, $COUNT, and $COUNTERS. You can reference these variables to provide you with special information. (See Chapter 6.) The Debugger also defines built-in functions for the supported languages so that you can use these functions to evaluate expressions during your Debugger sessions. (See Chapter 6.)

The $DBG block name is useful when you are trying to reference a Debugger variable when a user variable of the same name exists in the current evaluation environment.

Figure 4-2 illustrates the imaginary $DBG block in relation to user blocks. In a way, the $DBG block can be thought of as the invisible "parent" to all user external blocks. In Figure 4-2, the shaded area is invisible to the user. (For an explanation of how to use Debugger-defined variables and built-in functions, see Chapter 6.)

$EXTERNAL Program Block: The Debugger's $EXTERNAL program block, which is also invisible to users, may be used to reference user variables that are declared to be external. This is useful when trying to reference external variables that have not been declared in the program block corresponding to the current evaluation environment. Structure or record variable names must be qualified at the top level when used with $EXTERNAL. (For an explanation of how to reference external variables, see Chapter 6.)



$DBG Program Block
Figure 4-2

ENVIRONMENTS

To correctly identify the location of a variable or an executable statement, the Debugger maintains two distinct pointers, which are used as defaults to certain commands. These pointers are:

- Execution environment pointer
- Evaluation environment pointer

The term environment refers to a location that the Debugger recognizes as the current environment. The pointers, therefore, help the Debugger identify variables and program locations.

Execution Environment Pointer

The execution environment pointer gives the location at which execution resumes when a CONTINUE or single-step command is given. The execution environment is undefined — not known to the Debugger — at the beginning of a Debugger session, before the first RESTART, or after program execution is complete. The execution environment pointer has a defined location only when execution has been suspended. The location

of this pointer may be changed with the GOTO command and determined with the WHERE command. (See Chapter 5.) The execution environment pointer corresponds to a statement, an entry to a program block, or exit from a program block.

Evaluation Environment Pointer

The evaluation environment pointer gives the default program block to be used for finding variables and statements and for examining source files. This default program block is known as the evaluation environment. The evaluation environment may be changed to a block other than the default using the ENVIRONMENT command. (See Chapter 6.)

The default evaluation environment depends on how the Debugger is entered:

- When the Debugger is initially entered, the location is set to the program block corresponding to the main program.
- When the Debugger is reentered from the user program, as, for example, from a breakpoint, the location is set to the program block corresponding to the execution environment pointer.
- When the Debugger is reentered following user program termination, the location is once again set to the program block of the main program.

The evaluation environment pointer simplifies the use of the Debugger by providing a default evaluation environment. Without it, each reference to a variable or statement would require an accompanying program block name to identify the block where it could be found.

LANGUAGE OF EVALUATION

The language that the Debugger uses at any given time to evaluate data is called the language of evaluation. The language of evaluation tells the Debugger which language syntax rules to use when expressions are evaluated.

The default language used for evaluation is set to the source language of the program block containing the evaluation environment pointer. You can change the language of evaluation using the LANGUAGE command. (See Chapter 6.)

Another term to describe the default language and all of its syntax rules is host language.

ACTIVATIONS

An activation refers to a particular execution of a program block. An activation number specifies a particular activation of a program block when more than one activation can exist. Activation 2 refers to the second execution of a program block, activation 3 refers to the third execution, and so on. More than one activation of a program block can exist if the block calls itself (recursion) or causes itself to be called. More than one activation can also exist when you use the Debugger's CALL command. (The CALL command is described in Chapter 7.)

You may specify activation numbers as absolute or relative.

Absolute Activation Number

An absolute activation number, which is an unsigned integer constant, specifies the actual number of the activation -- second, third, fourth, etc. For example, you would use an absolute activation number to examine a variable:

```
> : RECURSE\2\X
```

In the example shown above, you are evaluating the variable X in the second activation of a program block called RECURSE. Note the use of backslashes.

Relative Activation Number

A relative activation number specifies the number of activations to count backwards from, beginning at the most recent activation of the specified program block. This number is specified by a minus sign (-) immediately followed by an integer constant.

Suppose your evaluation environment pointer were located at the fifth activation of program block FACTORIAL and you wanted to set the evaluation environment to activation 4 of FACTORIAL. You could do this by giving the following command:

```
> ENVIRONMENT FACTORIAL\-1
```

Note

Whenever the Debugger refers to an activation of a program block, the Debugger displays the activation number only if there is more than one activation of the block.

ACTIVE PROGRAM BLOCKS

Certain debugger operations that correspond to a particular program block can occur only when that program block is active. An active block means that the block must have been called, but not yet returned.

IDENTIFYING VARIABLES

The rules for identifying variables within the Debugger are identical to the rules established by the host language. The syntax has been expanded, however, so that you can reference any variable in the debugging environment.

One of the three formats below is used to identify a variable.

▶ variable-name

This format specifies a variable in the default program block described by the current evaluation environment. variable-name is the variable name, possibly qualified and/or subscripted according to the rules of the host language. For example, to reference VAR1 in the current block, specified by the evaluation environment pointer, enter:

```
VAR1
```

▶ program-block-name\variable-name

This format specifies a variable in a named program block. program-block-name is the name of a program block and variable-name is as defined above. This format is normally used to reference variables in program blocks other than the current evaluation environment. If there is more than one activation of this block, the most recent activation is used. For example, to reference VAR1 in program block SUBR1, enter:

```
SUBR1\VAR1
```

▶ program-block-name\activation-number\variable-name

This format specifies a variable in a named program block and activation. program-block-name and variable-name are as defined above. activation-number is the activation number of the program block. For example, to reference VAR1 in absolute activation 2 of subprogram SUBR1, enter:

```
SUBR1\2\VAR1
```

IDENTIFYING STATEMENTS

Debugger commands such as BREAKPOINT and GOTO require the identification of a statement within the debugging environment. In the format descriptions below, the following terms are used:

- source-line is a source line number, the physical line number in the source file. The SOURCE command displays the line numbers for each statement in the source file.
- statement-offset is the number of statements to count from the first statement on a multistatement line. The first statement on a line has a statement offset of 0, the second has an offset of 1, and so on.
- insert-line is a physical line number in a $INSERT or %INCLUDE file.
- statement-label is a statement label number or label constant in any of the seven supported languages. (Labels that begin with a digit are preceded by a dollar sign ($) to distinguish a label from a source line number.)
- line-offset is the number of physical source lines following the line containing statement-label.

Any of the formats described below may be preceded by a backslash and program block name:

```
program-block-name\statement-identifier
```

Absence of the program-block-name indicates that the current evaluation environment pointer is used to find the given statement.

The following FORTRAN function is used as an example in the text below. Debugger source line numbers have been added:

```
1:        INTEGER*2 FUNCTION ABSADD (I, J)
2:        INTEGER*2 I,J
3: 10     ABSADD = I + J
4: 20     IF (ABSADD .LT. 0) ABSADD = -ABSADD
5:        RETURN
6:        END
```

Statement Identifier Formats

There are six possible ways of identifying a statement, three of which use source file line numbers. The other three use statement labels. The six formats follow:

▶ source-line

This format identifies the leftmost statement on the specified source line number. For example, assume that ABSADD is the current evaluation environment. The command:

```
BREAKPOINT 3
```

sets a breakpoint at:

```
ABSADD = I + J
```

on source line 3. If ABSADD were not the evaluation environment, it would be necessary to use the program block name, ABSADD:

```
BREAKPOINT ABSADD\3
```

This format is one of the most common.

Note

As of this software release, you do not always have to specify the program block name, as shown above. If you reference a source line in the same file as the evaluation environment pointer, the Debugger will derive the program block name and give a message indicating what was assumed. This often happens during Debugger operations involving Pascal and PL/I-G programs, which contain so-called internal procedures and functions that are declared in and compiled with the main program.

▶ source-line+statement-offset

This format is useful when multiple statements appear on one line. It specifies a statement that is not the leftmost statement on the source line. For example, source line 4 has two statements: the IF statement and the arithmetic assignment statement, which will be executed if the IF expression is true. To set a breakpoint at the second statement when the evaluation environment is not ABSADD, use:

```
BREAKPOINT ABSADD\4+1
```

Note

There must be no space before or after the +.

To set a breakpoint at the second statement when the evaluation environment is ABSADD, use:

```
BREAKPOINT 4+1
```

▶ source-line (insert-line)
source-line (insert-line+statement-offset)

This format is included for those who use executable statements in $INSERT or %INCLUDE files. This is rare since the primary use of $INSERT and %INCLUDE files is data declaration. In these cases, the source line number is the line number in the primary source file, which includes the $INSERT or %INCLUDE directive. The insert line number and statement offset are used as described above. These were not included in the example program because of their infrequent use.

For example, to set a breakpoint at the leftmost statement on physical line number 2 in the file inserted on primary file line number 12 in routine SUB:

```
BREAKPOINT SUB\12(2)
```

▶ statement-label

This format identifies a statement by the label associated with it. Labels can be used in all seven languages. References to FORTRAN, Pascal, and COBOL 74 statement numbers must be immediately preceded by a dollar sign to distinguish their numeric labels from source line numbers. To set a breakpoint at FORTRAN statement label number 20 in ABSADD, use:

```
BREAKPOINT ABSADD\$20
```

or, if the evaluation environment were set to ABSADD, just use:

```
BREAKPOINT $20
```

This format is one of the most common.

Some characteristics of statement labels in each language are:

- A FORTRAN IV and FORTRAN 77 statement label is numeric and is referenced with a preceding dollar sign to distinguish it from a source line number.
- A Pascal statement label is numeric, and it is declared in the LABEL declaration part of the program. It is also referenced with a preceding dollar sign.
- A PL/I-G statement label is alphanumeric. The first character in the label name must be an alphabetic character. It is not referenced with a preceding dollar sign.
- A COBOL 74 statement label is a COBOL paragraph name or section name that is alphanumeric. It is referenced with a preceding dollar sign only when the first character of the label is numeric.
- An RPG II statement label is an RPG tag that is alphanumeric. Like PL/I-G, the first character in the label must be an alphabetic character. It is not referenced with a preceding dollar sign.

- A C statement label is also alphanumeric. The first character in the label must be an alphabetic character. It is not referenced with a preceding dollar sign.

▶ statement-label+line-offset

This format identifies a statement by a statement label and line offset. The statement referenced using this format is always the leftmost statement on the line. The line-offset lines follow the source line on which the specified label is defined. To set a breakpoint at the RETURN statement on the source line 5 using this format, enter:

```
BREAKPOINT $20+1
```

No backwards referencing is available. That is, you cannot set a breakpoint at the statement on source line 3 by entering:

```
BREAKPOINT $20-1 (This is illegal.)
```

This format should not be confused with the second format described above. The item preceding the plus sign in the second format is a source line number, whereas the item here is a statement label.

▶ statement-label+line-offset+statement-offset

This is similar in function to the preceding format. However, it allows you to identify a statement that is not the leftmost on the source line specified by statement-label+line-offset. To set a breakpoint at the arithmetic assignment statement on source line 4 using this format, enter:

```
BREAKPOINT $10+1+1
```

The first +1 specifies the number of physical source lines beyond the line on which FORTRAN statement label 10 is defined. This is the source line on which the statement will be found. The second +1 specifies the second statement on that line. Remember this is an offset, and the first statement is +0.

Table 4-1 lists all six statement identifier formats.

Table 4-1
Statement Identifier Formats

- source-line
- source-line+statement-offset
- source-line (insert-line)
  source-line (insert-line+statement-offset)
- statement-label
- statement-label+line-offset
- statement-label+line-offset+statement-offset

SPECIAL CHARACTERS

Certain characters that you can enter at the terminal have special meanings to the Debugger. These special characters either cause special actions or are interpreted as part of special command syntax. For instance, when you enter a double quotation mark ("), the Debugger interprets it as the PRIMOS erase character and deletes the immediately preceding character. When you enter a left bracket ([), the Debugger interprets it as the beginning of a multiple command sequence called an action list. (Chapter 5 discusses action lists.)

Table 4-2 lists all of the special characters and their meanings.

Caution

The PRIMOS erase and kill characters listed in Table 4-2 are the system default (Prime-supplied) characters. You or your System Administrator can change these characters. So find out what your erase and kill characters are at your installation.

The Escape Character

The escape character affects the meaning of the character or characters that immediately follow it. For instance, the escape character entered before an erase, kill, or action list bracket character negates the special meanings of these characters so that they are interpreted literally.

Table 4-2
Special Characters

Character	Meaning
Erase character "	Erases the previous character typed. Assuming that your erase character is a double-quote (the system default), the command line: > ENVIO"RONMENT $P"MAI "N would be interpreted by the Debugger as: > ENVIRONMENT $MAIN
Kill character ?	Causes the line typed thus far to be ignored. Assuming your kill character is a question-mark (the system default), the command line: > BREAKPOINT 20?TRACEPOINT 20 would be interpreted by the Debugger as: > TRACEPOINT 20
Backslash \	Used in breakpoints, variable definitions and statement definitions to qualify a program block name. For example: > BREAKPOINT SUBR1\\ENTRY
Left bracket [	Begins an action list. For example: > BREAKPOINT SUBR1\\ENTRY [ARGS] Action lists are described in Chapter 5.

Table 4-2 (continued)
Special Characters

Character	Meaning
Right bracket]	Terminates an action list.
Quote ' "	Begins a text string. The quote may be single or double. Within this text string, the special meanings of semicolon, left bracket, right bracket, and the quote character, which didn't begin the string (double quote if the string is surrounded by single quotes, and vice versa) are ignored. These characters are interpreted literally. This string must be terminated with a matching quote. A quote character may be included with a string by supplying two quote characters. For example: 'This is a valid quoted '' string.' "So's this." 'This string is invalid; quotes mismatch" "This string is not terminated
Separator character ;	Separates multiple commands on one line. The Debugger default is a semicolon.
Escape ^	Always affects the meaning of the character or characters that immediately follow it. Table 4-3 describes the action taken by the Debugger for a number of escape-character combinations.

Note

The escape character referred to in this book is not the ASCII escape character, but rather a logical escape character, which is the up-arrow or circumflex character (^).

The sequence of characters beginning with the escape character and ending with the last character affected by the escape is known as the escape sequence.

Table 4-3 lists the ways that the escape character can affect the meanings of certain characters.

Table 4-3
Effects of Using Escape Character

Character	Escape Sequence	Effect
erase	^"	Enter erase character literally.
kill	^?	Enter kill character literally.
left bracket	^[	Enter left bracket literally.
right bracket	^]	Enter right bracket literally.
escape	^^	Enter escape character literally.
separator	^;	Enter a semicolon literally.
slash	^/	Enter a carriage return literally.
uppercase U lowercase u	^U ^u	Convert all of the following lowercase characters to uppercase.
uppercase L lowercase l	^L ^l	Convert all of the following uppercase characters to lowercase.
three-digit octal number	^nnn	Enter the ASCII character that this number represents.
quote	^' ^"	Enter quote literally.
newline	^ (carriage return)	Continue input on next physical line as part of the same input line.

Examples Using Escape Character: The following list gives examples using the escape character:

- The special meanings of the left bracket and double quote (erase character) are ignored with this command:

```
> SOURCE LOCATE ^[MACRO ^"STRING
```

  Therefore, the text string to be located is '[MACRO "STRING'.

- The following commands demonstrate how the escape, U, and L characters are used to convert uppercase characters to lowercase and vice versa. This command:

```
> : '^LTHIS BECOMES LOWERCASE, ^uthis uppercase'
```

  is interpreted by the Debugger as:

```
> : 'this becomes lowercase, THIS UPPERCASE'
```

- Notice how the escape-slash character sequence generates a literal carriage return:

```
> : ST
ST = 'ABCDEFGHIJ'
> LET ST = 'ABCDE^/FGHIJ'
> : ST
ST = 'ABCDE
          FGHI'
```

- Here is an example of an escape character with a three-digit octal number:

```
> : 'The ^244 symbol is a dollar sign'
```

  This command is interpreted by the Debugger as:

```
> : 'The $ symbol is a dollar sign'
```

- Here is an example of how the escape character followed by a carriage return causes continuation of input on the next line without ending a Debugger command:

```
> BREAKPOINT RESPOND\USERABORT [IF SUBSTR (INPUT,1,5) =^
'ABORT' [WHERE] ELSE [CONTINUE]]
```

SPECIAL SYMBOLS

The Debugger recognizes six special symbols, including the erase, kill, separator, and escape characters. The meanings of all six symbols are listed below:

Symbol	Meaning
erase	Erases the immediately preceding character.
kill	Ignores all characters typed so far on the line.
escape	Used as general escape character.
separator	Used as command separator.
wild	Used as SOURCE command wildcard for FIND and LOCATE operations.
blanks	Used as SOURCE command match for any number of blanks.

Displaying Special Symbols with PSYMBOL

The Debugger's PSYMBOL command displays a list containing the names of special symbols and their current character values. The format of the PSYMBOL command, abbreviated PSYM, is:

```
PSYMBOL
```

Here is an example of how it is used with the Debugger's default symbols:

```
> PSYMBOL
ERASE       "
KILL        ?
ESCAPE      ^
SEPARATOR   ;
WILD        !
BLANKS      #
```

Changing a Symbol Character with SYMBOL

The Debugger's SYMBOL command changes the value of a special symbol. The format of the SYMBOL command, abbreviated SYM, is:

```
SYMBOL symbol-name character-value
```

The symbol-name is the name of the character symbol — ERASE, KILL, ESCAPE, SEPARATOR, WILD, or BLANKS. The character-value is the new character value of the symbol. It may not be a space, alphanumeric, or identical to an existing character symbol value.

Here are two examples of the SYMBOL command:

- To change the separator symbol to an ampersand (&), enter:

```
> SYMBOL SEPARATOR &
```

  Command lines entered hereafter must use an ampersand to separate commands rather than a semicolon (the default).

- To set the erase character to an at-sign (@), enter:

```
> SYMBOL ERASE @
```

5

Breakpoints and Program Control

Commands discussed in this chapter:

RESTART	ACTIONLIST	CLEARALL
CONTINUE	LIST	WHERE
BREAKPOINT	LISTALL	GOTO
IF	CLEAR	MAIN

INTRODUCTION

In Chapter 3, you learned how to activate, suspend, and continue the execution of your program with the RESTART, BREAKPOINT, and CONTINUE commands. These capabilities are some of the Debugger's program control features. Program control is defined as the manipulation of the execution of your program.

This chapter discusses most of the Debugger's program control features, including:

- Activating program execution with the RESTART command.
- Continuing execution with the CONTINUE command.
- Suspending execution with the BREAKPOINT command and all its powerful features. (The IF and ACTIONLIST commands are discussed also.)
- Displaying and deleting breakpoints with the LIST, LISTALL, CLEAR, and CLEARALL commands.
- Transferring program control with the GOTO command.
- Defining the main program with the MAIN command.

This chapter also discusses the WHERE command, which displays the location of the execution environment pointer. Although WHERE is an information request command, it is useful in program control operations.

Note

The remaining program control features are presented in Chapters 7 and 8. Single-stepping and calling program blocks are discussed in Chapter 7, and the UNWIND command is discussed in Chapter 8.

ACTIVATING PROGRAM EXECUTION

The RESTART command, which was introduced and demonstrated in Chapter 3, is used to activate and restart your program execution. (See also Chapter 3.)

The format of the RESTART command, abbreviated RST, is:

```
RESTART [step-command]
```

If no step command is supplied with RESTART, execution continues until control returns to Debugger command level via a breakpoint, completion of execution, or some other circumstance.

The step-command is an optional Debugger single-stepping command. The combined use of RESTART and a step command (STEP, STEPIN, IN, or OUT) causes the program to restart execution, then suspend execution after a specified number of statements has executed.

Single stepping, which is discussed completely in Chapter 7, allows you to execute one or more statements at a time. It also can step across, into, and out of called program blocks.

To restart program execution, enter:

```
> RESTART
```

To restart program execution and suspend execution immediately before the first executable statement in your main program, enter:

```
> RESTART IN
```

For more examples of RESTART, see Chapters 3 and 7.

CONTINUING PROGRAM EXECUTION

The CONTINUE command, which was also introduced and demonstrated in Chapter 3, continues program execution following a breakpoint, a single-step operation, or an error condition. Program execution resumes at the location specified by the execution environment pointer. (See also Chapter 3.)

The format of the CONTINUE command, abbreviated C, is:

```
CONTINUE
```

If the execution environment pointer is undefined — before program execution starts or after program execution completes — then an attempt to continue will fail.

If program execution has stopped at an error condition, you may or may not be able to continue program execution. If not, enter a GOTO command, which is described later in this chapter. Program execution may be restarted with RESTART.

To continue program execution from the last breakpoint, enter:

```
> CONTINUE
```

Other examples of the CONTINUE command are given in Chapter 3.

SETTING BREAKPOINTS

As you recall from Chapter 3, you can suspend the execution of your program using the BREAKPOINT command. (See also Chapter 3.) BREAKPOINT is one of the most fundamental Debugger commands. You can place a breakpoint on any executable statement -- any statement that performs some action. You can also place a breakpoint on an entry or exit to a called program block. Breakpoints allow you to examine your program's data strategically while execution is frozen. A breakpoint also is known as a trap.

The format of the BREAKPOINT command, abbreviated BRK, is:

```
BREAKPOINT [breakpoint-identifier] [action-list]
           [-AFTER value] [-BEFORE value]
           [-EVERY value] [-COUNT value] [-EDIT]

           [ -IGNORE  ]
           [ -NIGNORE ]
```

A breakpoint suspends execution immediately before the statement or labelled statement specified by the breakpoint-identifier.

The breakpoint-identifier identifies the place where you want to suspend program execution. For statement and label breakpoints, the breakpoint identifier is a statement or label identifier, as defined in Chapter 4. For program block entry and exit breakpoints, the breakpoint identifier is defined in the entry/exit breakpoint discussion in the following section.

If the breakpoint identifier is omitted, the value of the execution environment pointer is used.

As you can see from the BREAKPOINT command format, there are many other powerful breakpoint features that were not presented in Chapter 3. These features, which are described and demonstrated in the sections that follow, are:

- Entry/exit breakpoints -- setting a breakpoint at the entry to or exit from a called program block.
- Action lists -- executing one or more Debugger commands whenever a breakpoint trap occurs. (ACTIONLIST command is discussed.)
- Conditional action lists -- executing an action list contingent upon the result of an expression using the IF command.
- Conditional breakpoints -- taking or not taking a breakpoint trap, depending on the conditions you specify. (-AFTER, -BEFORE, and -EVERY options are used.)
- The breakpoint counter -- keeping track of the number of times a breakpoint has been encountered during program execution. (-COUNT option is used.)
- Breakpoint ignore flag -- suppressing a breakpoint so that the breakpoint is never taken. (-IGNORE and -NIGNORE options are used.)

Note

The BREAKPOINT command's -EDIT option is used to edit and modify an existing breakpoint using the Debugger's command line editor. (See Chapter 10.)

Entry/exit Breakpoints

Besides an executable statement, a breakpoint can be set at the entry to or exit from any program block. Entry/exit breakpointing is very useful for examining your program's data while execution is frozen, immediately before or after a program block is called.

An entry trap occurs inside the called program block, immediately before the first executable statement. The exit trap occurs outside the program block, after the block has returned.

An entry or exit is identified by one of the following three formats:

- program-block-name\\breakpoint-type
- \breakpoint-type
- program-block-name\

The breakpoint-type can be either ENTRY (abbreviated EN) or EXIT (abbreviated EX). The program-block-name is the name of the program block at which you want to break.

Suppose you were using the first format and you wanted to break at the entry of a program block named CONVERT. You would enter the following command:

```
> BREAKPOINT CONVERT\\ENTRY
```

Similarly, if you wanted to break at the exit, you would enter:

```
> BREAKPOINT CONVERT\\EXIT
```

The first format is used when you are not in the same evaluation environment as the program-block at which you want to break; that is, it is used when you are debugging in another block.

If you want to break at the entry or exit of the current evaluation environment block — the block in which you are debugging — use the second format:

```
> BREAKPOINT \ENTRY
> BREAKPOINT \EXIT
```

The third format allows you to break at both the entry and exit of any specified program block, in this case, CONVERT:

```
> BREAKPOINT CONVERT\
```

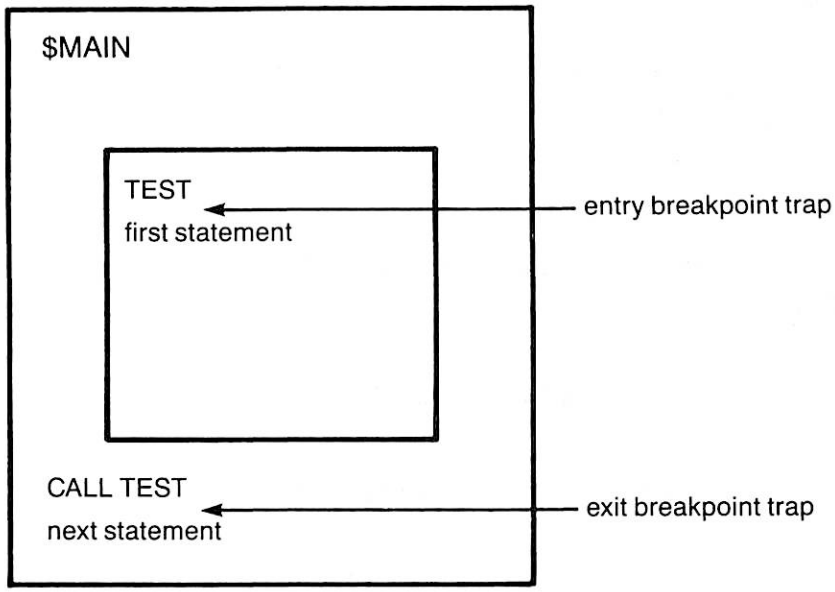
Consider the following PL/I-G program, which has two procedures — the main program block, MULT, which multiplies two numbers, and internal procedure SUBTR, which subtracts two numbers:

```
 1: MULT : PROCEDURE;
 2:   DECLARE (P, Q, R) FIXED BIN(15);
 3:     P = 5;
 4:     Q = 10;
 5:     CALL SUBTR (P, Q);
 6:     R = P * Q;
 7:     PUT SKIP LIST('The product is', R);
 8:     PUT SKIP;
 9: SUBTR : PROCEDURE (X, Y);
10:   DECLARE (X, Y, Z) FIXED BIN(15);
11:     Z = Y - X;
12:     PUT SKIP LIST('The remainder is', Z);
13:     PUT SKIP;
14: END SUBTR;
15: END MULT;
```

The following example uses this program to demonstrate all three entry/exit breakpoint formats:

```
> BREAKPOINT SUBTR\\ENTRY
> RESTART

**** breakpointed at entry to MULT.SUBTR
> BREAKPOINT \EXIT
> CONTINUE

The remainder is            5

**** breakpointed at exit from MULT.SUBTR
> BREAKPOINT MULT\
> RESTART

**** breakpointed at entry to MULT
> CONTINUE

**** breakpointed at entry to MULT.SUBTR
> CONTINUE

The remainder is            5

**** breakpointed at exit from MULT.SUBTR
> CONTINUE

The product is             50

**** breakpointed at exit from MULT
> CONTINUE

**** Program execution complete.
>
```

Figure 5-1 illustrates the exact positions where entry and exit breakpoints are trapped. In this figure, which uses two sample blocks named MAIN and TEST, notice how the entry trap occurs inside the called program block, immediately before the first executable statement. The exit trap occurs outside the called program block, after the call statement but before the next executable statement.



Positions of Entry and Exit Breakpoints
Figure 5-1

Action Lists

An action list allows you to execute one or more Debugger commands whenever a breakpoint trap occurs.

An action list saves you the time and trouble of entering each command, one at a time, after the breakpoint occurs.

To create an action list, enclose the list of Debugger commands in paired square brackets [ ], and separate the commands with semicolons. For example:

```
[: X; TYPE X; TYPE Y]
```

The action list is created with your breakpoint:

```
> BREAKPOINT 5 [: X; TYPE X; TYPE Y]
```

The command shown above suspends execution just before source line number 5 of your program, then immediately executes the commands listed within the square brackets. Execution remains suspended until another command such as CONTINUE or RESTART is given.

To continue execution automatically after the action list has executed, simply include a CONTINUE command as the last command in the action list:

```
> BREAKPOINT 5 [: X; TYPE X; TYPE Y; CONTINUE]
```

The action list is a permanent part of the breakpoint unless it is deleted or modified.

You can delete an action list by:

- Reentering the breakpoint followed by an empty set of brackets:

```
> BREAKPOINT 5 [ ]
```

- Deleting the entire breakpoint with the Debugger's CLEAR or CLEARALL command, discussed later in this chapter.

You can modify an action list by:

- Reentering the breakpoint with the new desired action list.
- Using BREAKPOINT's -EDIT option with the Debugger's command line editor, which is discussed in Chapter 10.

Conditional Action Lists -- the IF Command: The Debugger's IF command executes an action list conditionally, contingent upon the result of an expression.

The format of the IF command is:

```
IF expression action-list [ELSE action-list]
```

The expression is any valid expression in the host language. It can be either true or false. If the expression is true, the first action list immediately following the expression is executed, and the ELSE clause,

if present, is ignored. If the expression is false, the first action list is ignored, but the ELSE action list, if present, is executed. For example:

```
> BREAKPOINT 8 [IF A > B [: A + B] ELSE [CONTINUE]]
```

Caution

When you tell the Debugger to evaluate an expression, the Debugger only understands syntax written in the host language. The expression A > B used above assumes a PL/I-G or Pascal language of evaluation. If you were debugging a FORTRAN program, the Debugger would not understand the > operator. You would have to use a FORTRAN operator:

```
IF A.GT.B
```

In a COBOL 74 environment, you could enter:

```
IF A GREATER THAN B
```

For more information on the language of evaluation, see Chapters 4 and 6.

A conditional action list is also referred to as a nested action list, because action lists contain other action lists. Here is an example of an IF command clause within the action list of another IF command clause:

```
> BREAKPOINT 8 [IF A <= B [IF B <= C [: C; CONTINUE]^
ELSE [: B; CONTINUE]] ELSE [CONTINUE]]
```

The example shown above assumes a Pascal or PL/I-G language of evaluation.

Note

You can use the IF command outside an action list, directly after the Debugger's > command prompt. However, the IF command is most often used for conditional action lists and macros. (Chapter 9 discusses macros.)

Action List Examples: Consider the following Pascal program:

```
 1: PROGRAM Even_Odd;
 2: VAR
 3:   I : INTEGER;
 4: BEGIN
 5:   FOR I := 1 TO 10 DO
 6:       IF I MOD 2 = 0 THEN
 7:         WRITELN ('The number', I:2, ' is even')
 8:       ELSE
 9:         WRITELN ('The number', I:2, ' is odd')
10: END.
```

This program determines whether a number is odd or even. The output looks like this:

```
OK, SEG EVEN_ODD
The number 1 is odd
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
The number10 is even
OK,
```

In the following debugging session, a breakpoint that contains a conditional action list is set. This action list evaluates the value of the number only if it is even. The Pascal expression I MOD 2 yields the remainder of a division operation, in this case, the remainder of I divided by 2. Notice what happens to the output:

```
OK, DBG EVEN_ODD

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> SOURCE PRINT 23
    1: PROGRAM Even_Odd;
    2: VAR
    3:   I : INTEGER;
    4: BEGIN
    5:   FOR I := 1 TO 10 DO
    6:       IF I MOD 2 = 0 THEN
    7:         WRITELN ('The number', I:2, ' is even')
    8:       ELSE
    9:         WRITELN ('The number', I:2, ' is odd');
   10: END.
BOTTOM
> BREAKPOINT 6 [IF I MOD 2 = 0 [: I; CONTINUE] ELSE [CONTINUE]]
> RESTART
The number 1 is odd
I = 2
The number 2 is even
The number 3 is odd
I = 4
The number 4 is even
The number 5 is odd
I = 6
The number 6 is even
The number 7 is odd
I = 8
The number 8 is even
The number 9 is odd
I = 10
The number10 is even

**** Program execution complete.
>
```

In the next example, the same breakpoint action list is modified so that two conditions — specified by nested IF commands — are met. This action list evaluates the number only if it is even and if it is greater than 6:

```
> BREAKPOINT 6 [IF I MOD 2 = 0 [IF I >= 6 [: I; CONTINUE] ELSE^
[CONTINUE]] ELSE [CONTINUE]]
> RESTART
The number 1 is odd
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
I = 6
The number 6 is even
The number 7 is odd
I = 8
The number 8 is even
The number 9 is odd
I = 10
The number10 is even

**** Program execution complete.
>
```

Displaying Action Lists with ACTIONLIST Command: The Debugger's ACTIONLIST command, abbreviated AL, can display the commands of an action list prior to their execution. The format of the ACTIONLIST command is:

```
ACTIONLIST { SUPPRESS }
           { PRINT    }
```

Normally, no action lists are displayed prior to their execution. The PRINT option specifies all action list commands to be displayed. The SUPPRESS option deactivates the PRINT option, causing no action list commands to be displayed once again.

When an action list is displayed on your terminal, a number enclosed in angle brackets < > is printed at the left margin immediately preceding the action list. This number is the action list depth counter, which specifies the nesting level of the action list. This counter is incremented by 1 for each pending action list in which one or more commands remain to be executed. It is decremented following the execution of the last command in the action list.

If you specify the PRINT option, the commands contained within all action lists and macro command lists are displayed immediately before they are executed. (Chapter 9 describes macros.)

Here is an example of the ACTIONLIST PRINT command:

```
> BREAKPOINT 9 [: X; TYPE Y]
> ACTIONLIST PRINT
> RESTART

**** breakpointed at TEST\9:

<1> : X; TYPE Y
X = 5
integer parameter
>
```

If you specify the SUPPRESS option, no information is displayed when an action list or macro is executed:

```
> ACTIONLIST SUPPRESS
```

Conditional Breakpoints

In the previous section, you learned that a certain condition can cause an action list to execute or not execute. A breakpoint itself can also be conditional. That is, a breakpoint trap can be taken or not taken during the execution of your program, depending on the condition you specify.

Three BREAKPOINT command options (-AFTER, -BEFORE, and -EVERY) can be used to trap breakpoints conditionally. These options work in conjunction with the breakpoint counter, which keeps track of the number of times the breakpoint has been encountered. This counter is set to 0 when the breakpoint is created and incremented by one each time the breakpoint trap is taken.

When the -AFTER, -BEFORE, or -EVERY option is specified on a BREAKPOINT command line, the option is followed by a number that is compared to the value of the counter every time the breakpoint is encountered.

-AFTER Option: The -AFTER option causes the breakpoint trap to occur only when the value of the counter exceeds the value of the number following -AFTER.

-BEFORE Option: The -BEFORE option causes the breakpoint trap to occur only when the value of the counter is less than the value following -BEFORE.

-EVERY Option: The -EVERY option causes the breakpoint trap to occur every n iterations through the breakpoint location, where n is the value following -EVERY.

The following example demonstrates the -AFTER and -BEFORE options with the EVEN_ODD Pascal program:

```
> SOURCE PRINT 23
    1: PROGRAM Even_Odd;
    2: VAR
    3:   I : INTEGER;
    4: BEGIN
    5:   FOR I := 1 TO 10 DO
    6:       IF I MOD 2 = 0 THEN
    7:         WRITELN ('The number', I:2, ' is even')
    8:       ELSE
    9:         WRITELN ('The number', I:2, ' is odd')
   10: END.
BOTTOM
> BREAKPOINT 6 -AFTER 3 -BEFORE 5
> RESTART
The number 1 is odd
The number 2 is even
The number 3 is odd

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
The number10 is even

**** Program execution complete.
>
```

Here is an example of the -EVERY option used with the same program:

```
> BREAKPOINT 6 -EVERY 3
> RESTART
The number 1 is odd
The number 2 is even

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 3 is odd
The number 4 is even
The number 5 is odd

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 6 is even
The number 7 is odd
The number 8 is even

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 9 is odd
The number10 is even

**** Program execution complete.
>
```

The Breakpoint Counter

The breakpoint counter, as pointed out in the previous section, keeps track of the number of times that a breakpoint has been encountered. This counter, which is known to the Debugger, but is often invisible to the user, is set to 0 when the breakpoint is created and incremented by 1 each time the breakpointed location is encountered.

Note

The breakpoint counter is visible to the user when you use the LIST and LISTALL commands, which is discussed later in this chapter, or when you reference the Debugger-defined variable, $COUNT, which is discussed in Chapter 6.

The BREAKPOINT command's -COUNT option can be used to reset the breakpoint counter. It is set to the value that you specify after the option. Here is an example that uses the -COUNT option:

```
> BREAKPOINT 6 -BEFORE 3
> RESTART

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 1 is odd

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
The number10 is even

**** Program execution complete.
> BREAKPOINT 6 -COUNT 0
> RESTART

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 1 is odd

**** breakpointed at EVEN_ODD\6
> CONTINUE
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
The number10 is even

**** Program execution complete.
>
```

Breakpoint Ignore Flag

While debugging, you may want to suppress the breakpoints so that the breakpoint trap is never taken. The breakpoint ignore flag, specified by the BREAKPOINT command's -IGNORE option, causes a breakpoint never to be taken. Whenever an ignored breakpoint is encountered, only the counter is incremented; your program execution continues.

Ignoring a breakpoint is useful for suppressing a trap while retaining all its information, especially action lists, for future use. It is also useful for obtaining a count of the number of times the breakpoint is encountered. The -NIGNORE option deactivates the ignore flag so that the breakpoint trap is taken again.

Here is an example that uses an ignore flag:

```
> BREAKPOINT 6 -IGNORE
> RESTART
The number 1 is odd
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
The number10 is even

**** Program execution complete.
> BREAKPOINT 6 -NIGNORE
> RESTART

**** breakpointed at EVEN_ODD\6
>
```

DISPLAYING YOUR BREAKPOINTS

During your debugging sessions, you may want to display the locations, counts, and action lists of your breakpoints. The Debugger's LIST and LISTALL commands display breakpoint attributes for you. (The LIST and LISTALL commands also display your tracepoint attributes. See also Chapter 8.)

The LIST Command

The LIST command displays the attributes of one breakpoint. The format of the LIST command is:

```
LIST [breakpoint-identifier]
```

Used without the breakpoint-identifier, LIST displays the attributes for the breakpoint at the location specified by the execution environment pointer:

```
> LIST
Type  Location
brk   $$MAIN$$\4, count = 1
      [IF A >= 3 [: B] ELSE [CONTINUE]]
```

The breakpoint-identifier can be any valid breakpoint identifier, such as a simple source line number in the current evaluation environment:

```
> LIST 7
Type  Location
brk   $$MAIN$$\7, count = 0
```

To display a breakpoint on source line number 18 in program block TEST, enter:

```
> LIST TEST\18
```

To list the attributes of the entry and exit breakpoints to program block TEST, enter:

```
> LIST TEST\
Type  Location
brk   entry to $$MAIN$$.TEST, count = 2
brk   exit from $$MAIN$$, count = 2
```

The LISTALL Command

The LISTALL command lists the attributes of all the breakpoints you have set. The format of the LISTALL command, abbreviated LSTA, is:

```
LISTALL [program-block-name [-DESCEND]] [ -BREAKPOINTS ]
                                        [ -TRACEPOINTS ]
```

The program-block-name is the name of the program block that contains the breakpoints you want to see.

If no arguments are specified, the Debugger displays a list of all breakpoint and tracepoint attributes that you have set:

```
> LISTALL
Type  Location
brk   entry to $MAIN, count = 1
brk   $MAIN\120+1, count = 0
      [IF I.GT.1000 [: Y] ELSE [CONTINUE]]
brk   $MAIN\134, count = 0, after 2
brk   ADDITM\12, count = 1
```

If you specify the -BREAKPOINTS option, only the breakpoints -- no tracepoints -- are listed:

```
> LISTALL -BREAKPOINTS
```

If you want to list all the breakpoints and tracepoints in a specific program block, enter the block name:

```
> LISTALL TEST
```

If you want to list just the breakpoints in a specific program block, enter the block name followed by the -BREAKPOINTS option:

```
> LISTALL TEST -BREAKPOINTS
Breakpoints at:
      entry to TEST, count = 0
      TEST\7, count = 0
      TEST\9, count = 0
```

The -DESCEND option displays all breakpoints and tracepoints for a specified program block and for all the nested program blocks or "descendants" contained in the specified block. A program block name must be specified with the -DESCEND option. For example:

```
> LISTALL TEST -DESCEND
Type  Location
brk TEST.TEST_TWO\20, count = 0
brk TEST.TEST_THREE\25, count = 0
brk TEST.TEST_FOUR\39, count = 0
> LISTALL -DESCEND
-DSC is allowed only when a procedure name is specified.
```

If you are using the -DESCEND option and want to list only breakpoints, specify both the -DESCEND and -BREAKPOINTS options:

```
> LISTALL TEST -DESCEND -BREAKPOINTS
```

The -TRACEPOINTS option functions like the -BREAKPOINTS option, except that it lists tracepoints. (See also Chapter 8.)

DELETING YOUR BREAKPOINTS

The Debugger's CLEAR and CLEARALL commands delete breakpoints. (These commands can also delete tracepoints. See also Chapter 8.)

The CLEAR Command

The CLEAR command deletes a breakpoint. The format of the CLEAR command, abbreviated CLR, is:

```
CLEAR [breakpoint-identifier]
```

The breakpoint-identifier must be any valid breakpoint identifier, such as a simple source line number, as defined for the BREAKPOINT command.

If you omit the breakpoint identifier, the breakpoint located at the current execution environment pointer is deleted.

To delete a breakpoint on source line number 13 in the current evaluation environment, enter:

```
> CLEAR 13
```

To delete a breakpoint on source line number 22 in program block TEST, enter:

```
> CLEAR TEST\22
```

To delete a breakpoint at the exit of program block TEST, enter:

```
> CLEAR TEST\\EXIT
```

To delete the breakpoints at the entry and exit to program block TEST, enter:

```
> CLEAR TEST\
```

The CLEARALL Command

The CLEARALL command deletes either all breakpoints in the debugging environment or all breakpoints in a specific program block. The format of the CLEARALL command, abbreviated CLRA, is:

```
CLEARALL [program-block-name [-DESCEND]] [ -BREAKPOINTS ]
                                         [ -TRACEPOINTS ]
```

The program-block-name is the name of the program block containing the breakpoints that you want to delete.

If no arguments are specified, all breakpoints and tracepoints are deleted:

```
> CLEARALL
```

To delete all breakpoints and no tracepoints, enter:

```
> CLEARALL -BREAKPOINTS
```

To delete all breakpoints only in a given program block TEST, enter:

```
> CLEARALL TEST -BREAKPOINTS
```

The -DESCEND option functions the same as it does with the LISTALL command. -DESCEND deletes all breakpoints and tracepoints in a specified program block and in all the nested program blocks or "descendants" contained in the specified block. A program block name must be specified with the -DESCEND option:

```
> CLEARALL TEST -DESCEND
```

If you are using the -DESCEND option and want to delete only breakpoints, specify both the -DESCEND and -BREAKPOINTS options:

```
> CLEARALL TEST -DESCEND -BREAKPOINTS
```

The -TRACEPOINTS option functions like the -BREAKPOINTS option, except that it deletes tracepoints. (See also Chapter 8.)

FINDING THE EXECUTION ENVIRONMENT POINTER

Often during your debugging sessions you may want to verify the location of the execution environment pointer -- the position at which execution is to resume. The WHERE command displays the location of the execution environment pointer. The format of the WHERE command, abbreviated WH, is:

```
WHERE [segment-number/address]
```

Entered by itself, WHERE displays the current location of the execution environment pointer.

This command can also find the program location that corresponds to a given segment and halfword memory address. The segment-number is a segment number represented in octal. The address is a halfword address represented in octal. If a segment and address are specified, the Debugger attempts to identify that address in terms of program location (program block name and statement identifier). If successful, this program location is displayed. Otherwise a diagnostic (unidentifiable address) is displayed.

Here is how the WHERE command is used to print the current location of the execution environment pointer:

```
> WHERE
Currently at TEST\25
```

Here is how the WHERE command is used to identify the program location that corresponds to address 4001/1412:

```
> WHERE 4001/1412
$MAIN\102
```

TRANSFERRING PROGRAM CONTROL

With the Debugger's GOTO command, you can change the execution environment pointer to indicate another statement in your program. When you issue this command, the place where execution is suspended changes to the statement you have specified. A CONTINUE command following a GOTO command would, therefore, resume program execution at the statement you have specified with GOTO.

The format of the GOTO command is:

```
GOTO [program-block-name\[activation-number\]]statement-identifier
```

The GOTO command may only be used to transfer control to a statement in an active program block, which means that the block must have been called, but not returned.

The statement-identifier can be a statement source line number, statement label, or any other valid identifier as defined in Chapter 4.

If an activation-number is specified — usually for recursive blocks — then control is transferred to the specified statement in that activation. If no activation number is specified, the most recent activation of the program block is assumed.

After a GOTO, the evaluation environment pointer is set to the new program block. If the specified program block is written in another language, then the language of evaluation is set to the new language.

Consider the following COBOL 74 program named CALLER:

```
 1:        IDENTIFICATION DIVISION.
 2:        PROGRAM-ID. CALLER.
 3:       *
 4:        DATA DIVISION.
 5:       *
 6:        WORKING-STORAGE SECTION.
 7:        01 A1.
 8:            02 A2                      PIC XX VALUE 'A2'.
 9:            02 A3                      PIC XX VALUE 'A3'.
10:            02 A4.
11:               03 A5                   PIC XX VALUE 'A5'.
12:               03 A6                   COMP PIC S9(4) VALUE 0.
13:            02 A7                      PIC XX VALUE 'A7'.
14:       *
15:        PROCEDURE DIVISION.
16:        001-BEGIN.
17:            PERFORM 002-CALL UNTIL A2 = 'B2'.
18:            DISPLAY 'END OF RUN'.
19:            EXHIBIT A6.
20:            STOP RUN.
21:       *
22:        002-CALL.
23:            CALL 'CALLED' USING A1.
```

The program CALLER calls the following program block named CALLED:

```
 1:        IDENTIFICATION DIVISION.
 2:        PROGRAM-ID. CALLED.
 3:       *
 4:        DATA DIVISION.
 5:       *
 6:        LINKAGE SECTION.
 7:        01 ARG1.
 8:            02 B2                  PIC XX.
 9:            02 B3                  PIC XX.
10:            02 B4.
11:               03 B5                   PIC XX.
12:               03 B6                   COMP PIC S9(4).
13:            02 B7                      PIC XX.
14:       *
15:        PROCEDURE DIVISION USING ARG1.
16:            DISPLAY 'ENTERING CALLED'.
17:            MOVE 'B2' TO B2.
18:            MOVE 'B3' TO B3.
19:            MOVE 'B5' TO B5.
20:            ADD 9999 TO B6.
21:            MOVE 'B7' TO B7.
22:            GOBACK.
```

The example that follows uses the GOTO command to skip over the statement in CALLED that causes the program to end (line 17). Thus, the program keeps looping back to line 16, increasing the initial value of A6 from 9999 to 29997. Notice the use of the WHERE command in this example:

```
> BREAKPOINT CALLED\16
> RESTART

**** breakpointed at CALLED\16
> WHERE
Currently at CALLED\16.
> GOTO 18
> WHERE
Currently at CALLED\18.
> CONTINUE

**** breakpointed at CALLED\16
> GOTO 18
> CONTINUE

**** breakpointed at CALLED\16
> CONTINUE
ENTERING CALLED
END OF RUN
A6 =     29997

EXIT.  Program exit from CALLER\20 ($001-BEGIN+4).
>
```

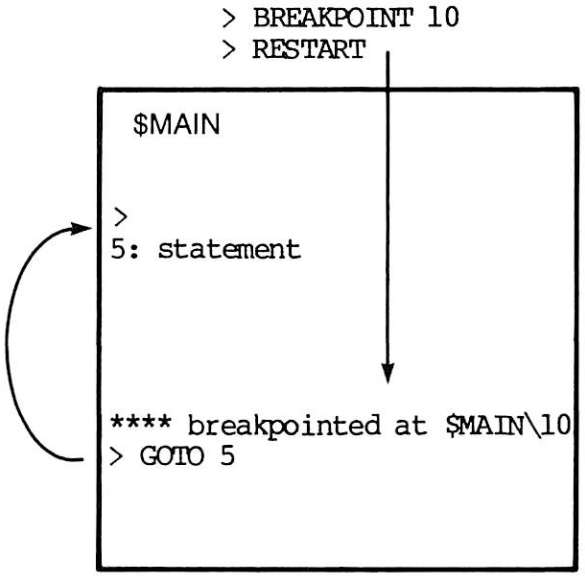
Figure 5-2 illustrates the GOTO command using one sample program block named MAIN.

DEFINING THE MAIN PROGRAM

Usually, the Debugger recognizes the program block that you have intended to be your main program -- the block that the Debugger calls when you enter a RESTART command. However, if for some reason you want the Debugger to recognize some other program block as your main program, use the MAIN command to tell the Debugger what your main program block should be.

The format of the MAIN command is:

```
MAIN [program-block-name]
```



Transferring Control with the GOTO Command
Figure 5-2

The program-block-name is the name of the program block that you want the Debugger to call when a RESTART command is given. For example, to set the main program to block TEST, enter:

```
> MAIN TEST
```

If you want to find out the main program that the Debugger currently recognizes, enter the MAIN command by itself:

```
> MAIN
Main program is TEST
```

Note

The program block that the Debugger recognizes as the main program by default is the first program block that you have loaded.

6
Data Manipulation

Commands discussed in this chapter:

:	ENVIRONMENT
TYPE	ENVLIST
LET	LANGUAGE
ARGUMENTS	

INTRODUCTION

In nearly all your debugging sessions, it becomes necessary to examine your program's variables and expressions while execution is frozen. Data manipulation is vital for determining why your program failed.

Chapter 3 introduced three data manipulation commands (:, TYPE, and LET). This chapter explains more about these three commands and describes these other features:

- Displaying the values of arguments passed to a program block with the ARGUMENTS command.
- Changing the evaluation environment with the ENVIRONMENT command.
- Changing the language of evaluation with the LANGUAGE command.
- Referencing Debugger-defined variables.
- Referencing external variables with $EXTERNAL.

THE EVALUATION COMMAND (:)

The Debugger command that examines or evaluates a simple variable or more complex expression is a colon (:). (See also Chapter 3.) The format of the evaluation command is:

```
:[ language-name[,print-mode] ] expression
 [ print-mode                 ]
```

The language-name specifies the language of evaluation, which tells the Debugger what language syntax rules to use when it evaluates an expression. If you specify no language name, then the expression is evaluated according to the syntax rules of the host language -- the language used for the program block corresponding to the current evaluation environment. You may specify one of the following language names. Abbreviations are underlined:

- FORTRAN
- F77
- PL1G
- PASCAL
- COBOL
- VRPG
- C

The language you give in the : command line overrides the language of the current block or the language set by the LANGUAGE command for the duration of the : command only.

Note

You can change the language of evaluation explicitly with the Debugger's LANGUAGE command, explained later in this chapter.

The print-mode specifies the format in which the result is printed. You may specify one of the following print modes. Abbreviations are underlined:

- ASCII
- BIT
- DECIMAL

- FLOAT
- HEX
- OCTAL

If you specify no print mode, the default print mode, which corresponds to the declared type of the variable or the resultant type of the expression, is used. The print mode you give in the : command line overrides the print mode of the given expression for the duration of the command.

Note

The PMODE command explicitly sets the print mode for a variable so that the print mode will be used whenever that variable is evaluated. The PMODE command is fully explained in Chapter 13.

If you specify the language name and/or print mode, they must immediately follow the colon. There must be no space between the colon and language name or print mode. If both language name and print mode are supplied, a comma must separate them. The expression must be preceded by a space. For example:

```
> :PASCAL, OCTAL X
```

Evaluating Variables

You may evaluate any variable according to the rules of the host language. Where applicable, you may evaluate:

- Simple variables
- Arrays
- Array elements
- Array cross sections
- Structures (records)
- Structure members (fields)
- Pointer-referenced data

Here are some examples of variable evaluation:

```
> : KEYS
KEYS = 14577

> :PL1G, BIT KEYS
KEYS = 0011100011110001 (b)

> :OCTAL SUBR\ADDRESS
ADDRESS = 64017 120270 (o)

> :FORTRAN ITEMNO
ITEMNO = 1374
```

The lowercase (b) and (o) stand for BIT and OCTAL respectively.

As you recall from Chapter 4, to reference a variable in a different program block, you must give the name of the block as well as the variable. The block must also be active. For example, to evaluate the variable X in block TEST you would enter:

```
> : TEST\X
```

Similarly, if you want to reference a variable in a particular activation of a program block, enter the activation number as well. For example:

```
> : TEST\3\X
```

In the example shown above, the variable X is being evaluated in the third activation of program block TEST.

Array References: You can reference a portion of an array by specifying a star extent or a "bound pair", which designates the range of a given dimension using a special Debugger notation (...).

A star extent works as follows. Substituting a star (*) for a subscript indicates that the full range of that dimension will be displayed or operated upon. For example, to reference the one-dimensional array formed by the elements (1, 2), (2, 2), and (3, 2) of the 3-by-3 array TICTACTOE, enter:

```
> : TICTACTOE (*, 2)
TICTACTOE(1) = 'X'
TICTACTOE(2) = 'X'
TICTACTOE(3) = 'O'
>
```

Here is an example using a COBOL 74 record structure that contains a two-dimensional array:

```
> : BUDGETED-AMT (*, 1)
THEDEPT(1).BUDGETED-AMT = 3002.00
THEDEPT(2).BUDGETED-AMT = 0.00
THEDEPT(3).BUDGETED-AMT = 0.00
THEDEPT(4).BUDGETED-AMT = 0.00
THEDEPT(5).BUDGETED-AMT = 0.00
THEDEPT(6).BUDGETED-AMT = 0.00
THEDEPT(7).BUDGETED-AMT = 0.00
THEDEPT(8).BUDGETED-AMT = 0.00
>
```

You can also limit the range of a dimension by specifying a bound pair of the form:

```
lower-bound ... upper-bound
```

lower-bound and upper-bound are any valid expressions that reduce to integer values. For example, given a three-dimensional array, CALENDAR, declared as "CALENDAR (12, 31, 1950:2000) CHARACTER (10) VARYING", you may enter the following:

```
> : CALENDAR (DECEMBER, 25, 1979...1985)
CALENDAR(1979) = 'Tuesday'
CALENDAR(1980) = 'Thursday'
CALENDAR(1981) = 'Friday'
CALENDAR(1982) = 'Saturday'
CALENDAR(1983) = 'Sunday'
CALENDAR(1984) = 'Tuesday'
CALENDAR(1985) = 'Wednesday'
>
```

Note that the resultant type of an array cross section is an array whose number of dimensions is equal to the number of subscripts with either a star or a bound pair. The range of such a dimension is the declared range if the subscript is a star, or the specified range if the subscript is a bound pair. From the example above, therefore, you can enter:

```
> TYPE CALENDAR (DECEMBER, 25, 1979...1985)
character(10) varying automatic 1 dimensional array: (1979:1985)
```

Here is another example of range used with a COBOL 74 record structure:

```
> : THEDEPT (7...8)
THEDEPT(7).THEREST(1).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(1).AMT-SPENT = 0.00
THEDEPT(7).THEREST(2).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(2).AMT-SPENT = 0.00
THEDEPT(7).THEREST(3).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(3).AMT-SPENT = 0.00
THEDEPT(7).THEREST(4).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(4).AMT-SPENT = 0.00
THEDEPT(7).THEREST(5).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(5).AMT-SPENT = 0.00
THEDEPT(7).THEREST(6).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(6).AMT-SPENT = 0.00
THEDEPT(7).THEREST(7).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(7).AMT-SPENT = 0.00
THEDEPT(7).THEREST(8).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(8).AMT-SPENT = 0.00
THEDEPT(7).THEREST(9).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(9).AMT-SPENT = 0.00
THEDEPT(7).THEREST(10).BUDGETED-AMT = 0.00
THEDEPT(7).THEREST(10).AMT-SPENT = 0.00
THEDEPT(8).THEREST(1).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(1).AMT-SPENT = 0.00
THEDEPT(8).THEREST(2).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(2).AMT-SPENT = 0.00
THEDEPT(8).THEREST(3).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(3).AMT-SPENT = 0.00
THEDEPT(8).THEREST(4).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(4).AMT-SPENT = 0.00
THEDEPT(8).THEREST(5).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(5).AMT-SPENT = 0.00
THEDEPT(8).THEREST(6).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(6).AMT-SPENT = 0.00
THEDEPT(8).THEREST(7).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(7).AMT-SPENT = 0.00
THEDEPT(8).THEREST(8).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(8).AMT-SPENT = 0.00
THEDEPT(8).THEREST(9).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(9).AMT-SPENT = 0.00
THEDEPT(8).THEREST(10).BUDGETED-AMT = 0.00
THEDEPT(8).THEREST(10).AMT-SPENT = 0.00
>
```

The Debugger can also compare arrays and structures. Consider the following two PL/I-G arrays:

```
COUNTS(1) = 6          LIMITS(1) = 10
COUNTS(2) = 3          LIMITS(2) = 5
COUNTS(3) = 6          LIMITS(3) = 5
COUNTS(4) = 11         LIMITS(4) = 12
COUNTS(5) = 2          LIMITS(5) = 2
```

You can compare these arrays with the following Debugger command:

```
> : COUNTS <= LIMITS
(1) = '1'b
(2) = '1'b
(3) = '0'b
(4) = '1'b
(5) = '1'b
```

The first '1'b means that COUNTS(1) is less than or equal to LIMITS(1) -- that 6 is less than 10 -- and so on.

Similarly, you can compare the array COUNTS this way:

```
> : COUNTS = 6
(1) = '1'b
(2) = '0'b
(3) = '1'b
(4) = '0'b
(5) = '0'b
```

Using Pascal character arrays, you have to compare an entire array to see if the condition is either TRUE or FALSE for every element in the array. For example, consider these two Pascal character arrays:

```
ARRAY_ONE [1] = 'h'          ARRAY_TWO [1] = 'o'
ARRAY_ONE [2] = 'e'          ARRAY_TWO [2] = 'l'
ARRAY_ONE [3] = 'l'          ARRAY_TWO [3] = 'l'
ARRAY_ONE [4] = 'l'          ARRAY_TWO [4] = 'e'
ARRAY_ONE [5] = 'o'          ARRAY_TWO [5] = 'h'
```

Watch what happens when these arrays are compared:

```
> : ARRAY_ONE = ARRAY_TWO
false
> : ARRAY_ONE <> ARRAY_TWO
true
> : ARRAY_ONE [3] = ARRAY_TWO [3]
true
>
```

Suppose you were debugging a Pascal program containing one character array declared as ARRAY OF CHAR, and one character string declared as type STRING, which is a Prime extension data type similar to the CHARACTER VARYING type in PL/I-G. In the following example, notice the differences in the ways the array and string are evaluated:

```
> BREAKPOINT 15
> RESTART

**** breakpointed at PREZ\15
> : BEST_PRESIDENT
BEST_PRESIDENT [1] = 'N'
BEST_PRESIDENT [2] = 'i'
BEST_PRESIDENT [3] = 'x'
BEST_PRESIDENT [4] = 'o'
BEST_PRESIDENT [5] = 'n'
> TYPE BEST_PRESIDENT
char static  1 dimensional array: [1..5]
> : CURRENT_PRESIDENT
CURRENT_PRESIDENT = 'Reagan'
> TYPE CURRENT_PRESIDENT
string[9] static
>
```

In the example shown above, the variables are fields in a Pascal record structure. Notice that you do not need to specify the record name when using the evaluation or TYPE command. The field name suffices.

Evaluating Expressions

The Debugger can evaluate any expression permitted by the source language. This includes the ability to evaluate expressions using:

- PL/I-G built-in functions and replaced symbols
- Pascal standard functions and CONSTANTS
- FORTRAN intrinsic functions and PARAMETERS
- One C function, SIZEOF

Table 6-1 lists the available PL/I-G, Pascal, FORTRAN, and C functions.

Here are some examples of how these functions would be used to evaluate data:

```
> : SQRT(X)

> : CHR(X+Y)

> LET X = SUCC(X)
```

Note

These functions are defined in the invisible $DBG block, and they are always known to the Debugger. (For more information on the $DBG block, see Chapter 4.)

Here are some more examples of expression evaluation:

```
> : BALANCE - (CHECKTOTAL + SVCCHARGE) + DEPOSITS
597.98

> : A .GT. 5
.TRUE.

> : LOC (GAMMA)
4002(0)/32

> : 'Hello' || ' there'
'Hello there'

> : COLOR
COLOR = BLUE
```

Table 6-1
PL/I-G, Pascal, FORTRAN, and C Supported Functions

ABS	COMPLEX	EXP	MAX1	QTANH
ACOS	CONJG	FAULT	MIN	QUAD
ADD	COPY	FIXED	MIN0	RANK
ADDR	COS	FLOAT	MIN1	REAL
ADDREL	COSD	FLOOR	MOD	REL
AFTER	COSH	HBOUND	MULTIPLY	REVERSE
AIMAG	CSIN	HIGH	NINT	RING
AINT	CSQRT	IABS	NOT	RND
ALOG	DABS	ICHAR	NULL	ROUND
ALOG10	DACOS	IDIM	ODD	RS
AMAX0	DASIN	IDINT	OFFSET	RT
AMAX1	DATAN	IDNINT	ONCODE	SEARCH
AMIN0	DATAN2	IFIX	OR	SEGNO
AMIN1	DATE	IMAG	ORD	SHFT
AMOD	DBLE	INDEX	POINTER	SIGN
AND	DBLEQ	INSERT	PRED	SIN
ANINT	DCMPLX	INT	PTR	SIND
ARCTAN	DCOS	INTL	QABS	SINH
ASIN	DCOSH	INTS	QACOS	SIZEOF
ATAN	DDIM	IQINT	QASIN	SNGL
ATAN2	DEC	IQNINT	QATAN	SQR
ATAND	DECAT	IRND	QATAN2	SQRT
ATANH	DECIMAL	ISIGN	QCOS	STACKBASE
BASEPTR	DELETE	LBOUND	QCOSH	STACKPTR
BASEREL	DEXP	LEN	QDIM	STR
BEFORE	DIM	LENGTH	QEXP	STRING
BIN	DIMENSION	LGE	QEXT	SUBSTR
BINARY	DINT	LGT	QEXTD	SUBTRACT
BIT	DIVIDE	LINKPTR	QINT	SUCC
BOOL	DLOG	LLE	QLOG	TAN
BYTE	DLOG10	LLT	QLOG10	TAND
CABS	DMAX1	LN	QMAX1	TANH
CCOS	DMIN1	LOC	QMIN1	TIME
CEIL	DMOD	LOG	QMIN1	TRANSLATE
CEXP	DNINT	LOG10	QMOD	TRIM
CHAR	DPROD	LOG2	QNINT	TRUNC
CHARACTER	DSIGN	LOW	QPROD	UNSPEC
CHR	DSIN	LS	QSIGN	UNSTR
CLOG	DSINH	LT	QSIN	VERIFY
CMPLX	DSQRT	LTRIM	QSINH	XOR
CMPX	DTAN	MAX	QSQRT	
COLLATE	DTANH	MAX0	QTAN	

Note

These functions may be used to evaluate expressions in any language, including COBOL 74 and RPG. However, the data type of the expression — or your host language's equivalent of the data type — must be the type that the function expects.

Evaluating Pointer Type Data

PL/I-G, Pascal, and C pointer variables and expressions can also be evaluated. Not only can you evaluate the pointer (the address) but you can also evaluate the value being pointed to (the value located at the address). For example, consider the following piece of Pascal code:

```
 3: TYPE
 4:   POINTER = ^ELEMENT;
 5:   ELEMENT = RECORD
 6:               VALUE : INTEGER;
 7:               NEXT : POINTER
 8:             END;
 9: VAR
10:  NEWVALUE, HEAD, TAIL : POINTER;
11: BEGIN
12:   RESET (INPUT, 'INPUT');
13:   NEW(NEWVALUE);
14:   READ(INPUT, NEWVALUE^.VALUE);
15:   NEWVALUE^.NEXT := NIL;
16:   HEAD := NEWVALUE;
17:   TAIL := NEWVALUE;
```

Based on the Pascal code shown above, you can use the evaluation command as shown in the following example:

```
**** breakpointed at LINK\17
> : NEWVALUE^.VALUE
VALUE = 7
> : NEWVALUE
NEWVALUE = 4027(3)/14
> : NEWVALUE^
NEWVALUE^.VALUE = 7
NEWVALUE^.NEXT = 7777(0)/0
> : HEAD^
HEAD^.VALUE = 7
HEAD^.NEXT = 7777(0)/0
>
```

In the example shown above, notice the difference between the address and the value contained at that address.

Note

When evaluating Pascal pointer expressions, you must enter two consecutive escape characters (^^) to negate the escape character's special meaning. (For your convenience, only one escape character is shown in these examples.)

In the next example, the TYPE and LET commands, which you learned in Chapter 3, are used with the pointer type data:

```
> TYPE NEWVALUE^.VALUE
integer based
> TYPE NEWVALUE
pointer static
> TYPE NEWVALUE^
record based
> LET NEWVALUE^.VALUE = 9
> : NEWVALUE^.VALUE
VALUE = 9
> LET HEAD^.NEXT = NEWVALUE
> : HEAD^.NEXT
NEXT = 4027(3)/14
>
```

The TYPE and LET commands are fully discussed in the following sections.

THE TYPE COMMAND

You can determine the data type and other attributes of a variable or expression by using the TYPE command. When the TYPE command is given, the Debugger evaluates the expression that follows, then prints the data type and other attributes of the resultant expression. (See also Chapter 3.)

The following information is printed, if applicable:

- Data type
- Precision
- Scale factor
- Storage class
- Array dimensions and bounds
- Values of enumeration constants
- Base type
- End points of subrange

The format of the TYPE command is:

```
TYPE expression
```

The expression is any expression permitted by the host language.

Here are some examples using the TYPE command:

```
> TYPE BUFFER
integer*2 common /BUFFER/

> TYPE TOTAL * 5
real*4

> TYPE TAXFNC
entry constant external (real*4 function)

> TYPE TITLE
character(25) varying automatic

> TYPE LINK
pointer automatic  1 dimensional array: (1:50)

> TYPE COLOR
enumerated static
    0:  RED
    1:  BLUE
    2:  GREEN

> TYPE NS
subrange static (THIRD..SIXTH)
```

THE LET COMMAND

The LET command allows you to assign a new value to any variable defined by the program. (See also Chapter 3.)

The format of the LET command is:

```
LET variable = expression
```

The variable is a user variable name, as defined in Chapter 4, and expression is any expression permitted by the host language whose resultant value is convertible to the data type of the variable.

When the LET command is given, the expression on the right side of the assignment statement is evaluated, and the resulting value is assigned to the variable. Type conversions allowed by the host language are performed by the Debugger before the assignment takes place. An error message is displayed if the conversion requested is illegal.

Here are some examples using the LET command:

```
> LET MAXENTRIES = 1000

> LET INDEX = FIRST + FREELIST (I)

> LET FLAG = .FALSE.
```

The Debugger supports assignment to structures and arrays. To assign each element of the one-dimensional array, LIST, to the corresponding element of the array cross-section referenced by TABLE (10, *), enter:

```
> LET TABLE (10, *) = LIST
```

To set each element of TABLE to zero, enter:

```
> LET TABLE = 0
```

To assign each member of the PL/I-G based structure STATUS to the corresponding member of the structure CURRENTSTATUS, enter:

```
> LET CURRENTSTATUS = LOCATOR -> STATUS
```

THE ARGUMENTS COMMAND

You can display the values of all arguments supplied to a given program block using the ARGUMENTS command. Seeing the values of arguments that are passed to called program blocks and how those values change is a handy tool to have in your debugging expeditions.

The format of the ARGUMENTS command, abbreviated ARGS, is:

```
ARGUMENTS program-block-name [\activation-number]
```

Given by itself, the ARGUMENTS command displays the values of the arguments passed to the program block defined by the evaluation environment pointer — the current program block.

When a program-block-name is given, the arguments passed to that block are displayed.

You can designate an activation-number to display the arguments passed to a particular activation of a program block, if more than one activation exists. If the activation number is omitted, the most recent activation is assumed. The activation of the program block must be active. (Activation is defined in Chapter 4.)

You can display the arguments to all called program blocks during entry tracing by specifying the ETRACE command with its ARGS option. (See Chapter 8.)

To display the arguments to program block SEARCH, enter:

```
> ARGUMENTS SEARCH
GROUP = 12
NAME = 'TAYLOR'
RTN_INDEX = 0
```

To obtain the arguments of the current program block, enter:

```
> ARGUMENTS
INTRAT = 0.8E-01
BALNC = 315.79
```

Consider the COBOL 74 programs, CALLER and CALLED, which were shown in Chapter 5. The following debugging example illustrates the ARGUMENTS command used on those COBOL 74 programs:

```
> BREAKPOINT CALLED\18
> RESTART
ENTERING CALLED

**** breakpointed at CALLED\18
> ARGUMENTS
ARG1.B2 = 'B2'
ARG1.B3 = '__'
ARG1.B4.B5 = '__'
ARG1.B4.B6 = 0
ARG1.B7 = '__'
>
```

CHANGING THE EVALUATION ENVIRONMENT

You can change the evaluation environment with the ENVIRONMENT command. The evaluation environment is the program block that the Debugger considers current and uses to identify statements and evaluate variables and expressions. Changing the evaluation environment in effect changes the evaluation environment pointer. (The evaluation environment pointer is defined in Chapter 4.)

Changing the evaluation environment is useful for evaluating data in a program block that is outside your current environment. If the block to which you set your evaluation environment is not active, you cannot evaluate automatic variables and expressions. Other types may be evaluated.

The ENVIRONMENT Command

The format of the ENVIRONMENT command, abbreviated ENV, is:

```
ENVIRONMENT [ program-block-name [\activation-number] ]
            [ -POP                                    ]
```

The program-block-name is the name of the program block that you want as the new evaluation environment. The activation-number is a particular activation of program-block-name. If no activation is specified, the most recent activation is assumed. (Activation is defined in Chapter 4.)

The -POP option, which is used to remove an environment from the evaluation environment stack, is described later in this section.

To display the value of the current evaluation environment, enter the ENVIRONMENT command by itself:

```
> ENVIRONMENT
Current evaluation environment is TEST.
>
```

If the block corresponding to the current evaluation environment is not active, the Debugger tells you so.

To set the evaluation environment to the block CONVERT, enter:

```
> ENVIRONMENT CONVERT
>
```

The ENVIRONMENT command is very useful for debugging recursive programs. You can change the evaluation environment to any particular activation of a recursive block. Then you can evaluate data in that activation. For example:

```
> ENVIRONMENT
Current evaluation environment is FROG.JUMP\3.
> : LEAP
LEAP = 3
> ENVIRONMENT JUMP\1
> ENVIRONMENT
Current evaluation environment is FROG.JUMP\1.
> : LEAP
LEAP = 1
>
```

In the example shown above, the variable LEAP is evaluated in activations 3 and 1 of the Pascal procedure JUMP, contained in the main program FROG.

The ENVIRONMENT command is very useful for evaluating a lot of data in any given block — recursive or otherwise -- because you only have to issue the name of the block once. For example:

```
> ENVIRONMENT JUMP\1
> : LEAP
LEAP = 1
> TYPE LEAP
integer automatic
```

Without the ENVIRONMENT command, you would have to issue the same commands this way:

```
> : JUMP\1\LEAP
LEAP = 1
> TYPE JUMP\1\LEAP
integer automatic
```

In other words, you would have to enter the name of the block, and if necessary, the activation number. This causes extra keystrokes and is time consuming, especially if you have a lot of data to examine.

Changing the evaluation environment is also useful if you have to set a lot of breakpoints in a block outside the current evaluation environment. This saves you the trouble of entering the program block name for each breakpoint. For example:

```
> BREAKPOINT 7
No such statement.
> ENVIRONMENT TEST
> BREAKPOINT 7
> BREAKPOINT 6
>
```

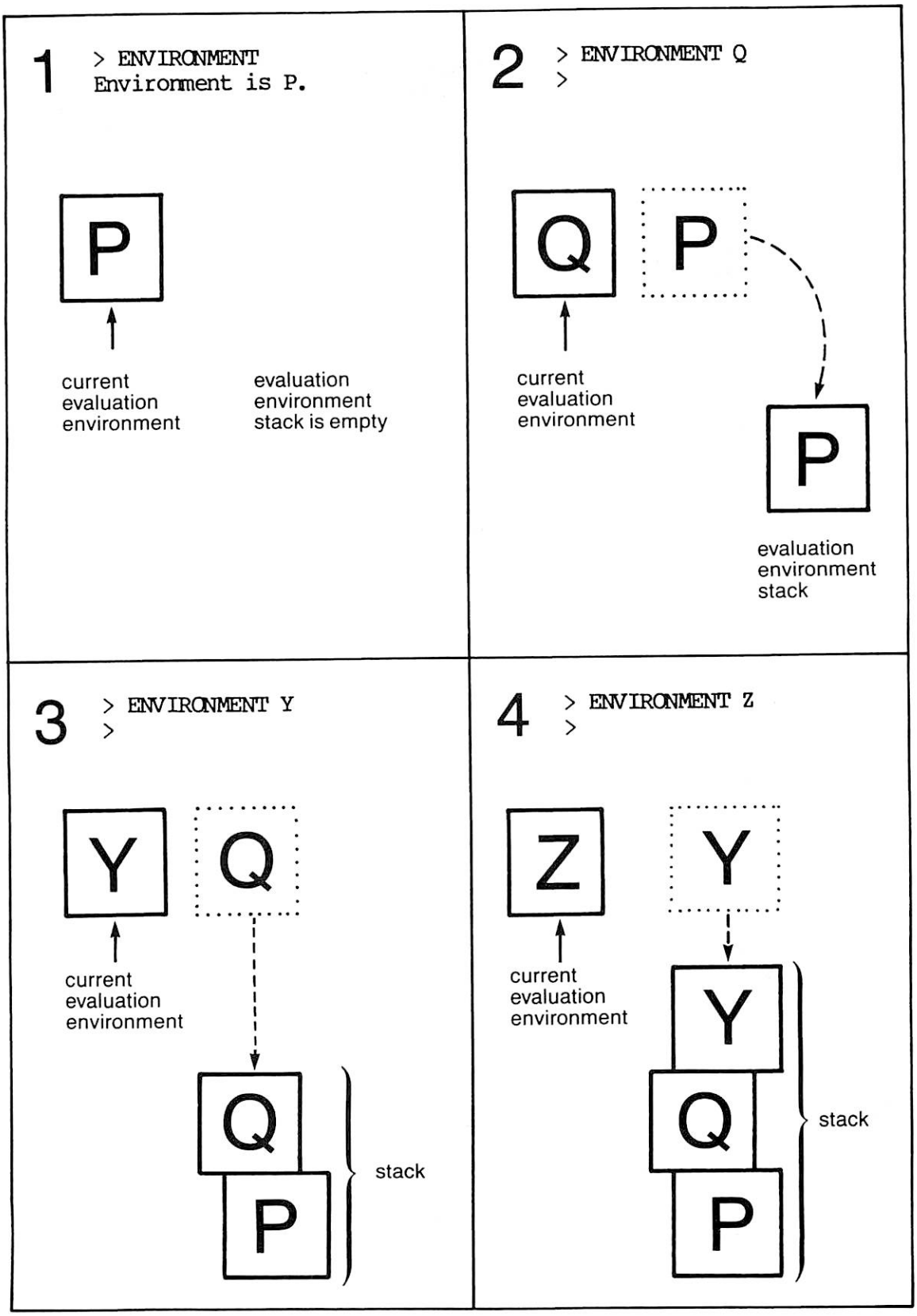
When you use the ENVIRONMENT command to set the evaluation environment to a block that is written in another language, the language of evaluation is automatically set to the language of the new block. The Debugger even tells you what the new language is. For example, if you set the evaluation environment to a FORTRAN subroutine named GETR, the Debugger gives you this response:

```
> ENVIRONMENT GETR
New language is Fortran.
```

The Evaluation Environment Stack: Each time you change the evaluation environment, the previous environment is pushed onto an internal stack known as the evaluation environment stack. This stack can contain up to 10 most recent environments that you have set with ENVIRONMENT. It can provide you with a way of returning to a previous environment without reentering the name of the program block and activation.

Figure 6-1 illustrates how evaluation environments are pushed onto the evaluation environment stack in a four-step sequence.

> **Caution**
>
> The evaluation environment stack is destroyed whenever you restart or continue program execution or issue a GOTO, CALL, or single-step command.



Pushing Environments onto Evaluation Environment Stack
Figure 6-1

You can remove or "pop" an environment off of the evaluation environment stack with the ENVIRONMENT command's -POP option. When you enter the -POP option, the evaluation environment at the top of the stack is removed and becomes the current evaluation environment. For example, to set the evaluation environment to the program block named SORT at the top of the stack, enter:

```
> ENVIRONMENT -POP
Environment is SORT.
```

Figure 6-2 illustrates how evaluation environments are popped off the stack in a four-step sequence with the -POP option.

Displaying the Stack with ENVLIST

The ENVLIST command displays the current evaluation environment and the contents of the evaluation environment stack. (The evaluation environment stack is described in the previous section.)

The format of the ENVLIST command, abbreviated EL, is:

```
ENVLIST
```

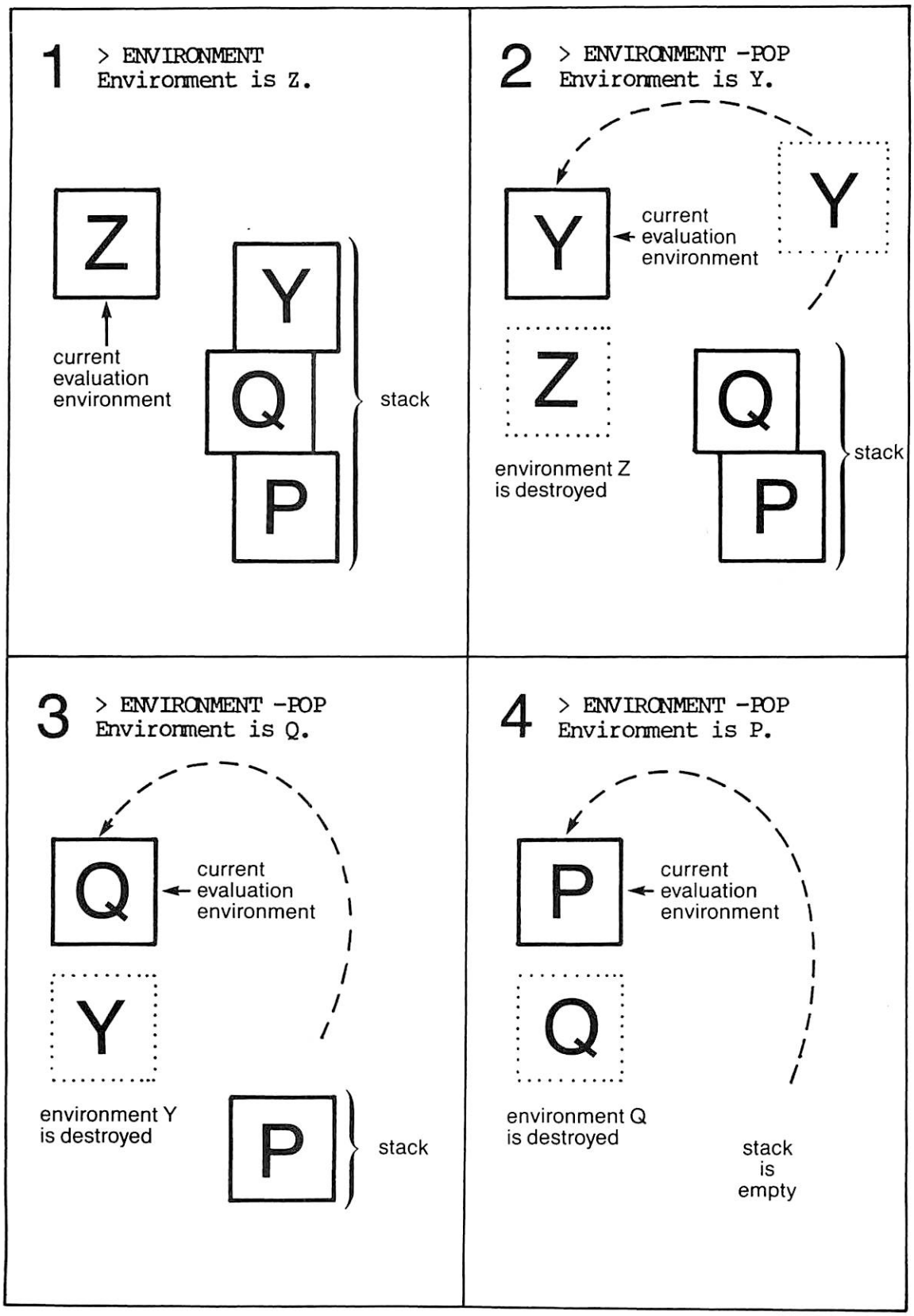
Here is an example of the ENVLIST command:

```
> ENVLIST
Current evaluation environment is PLOT_LINE.
The evaluation environment stack contains:

    PLOT_VECTOR\2
    PLOT_VECTOR\1
    PLOT_SETUP
```

CHANGING THE LANGUAGE OF EVALUATION

As you learned in Chapter 1, the Debugger is multilingual. This means the Debugger can communicate and evaluate expressions in any of the seven supported languages. The Debugger can switch from one language to another when you debug a program that contains several program blocks written in several languages.

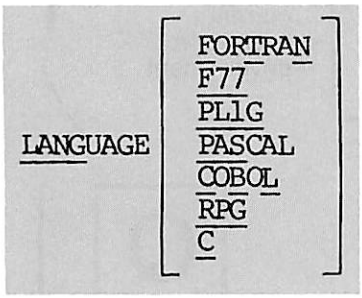
The language that the Debugger uses at any given time to evaluate data is called the language of evaluation. The language of evaluation tells the Debugger which language syntax rules to use when expressions are evaluated.



Popping Environments off the Evaluation Environment Stack
Figure 6-2

The default language used for evaluation is set to the source language of the program block containing the evaluation environment pointer. You can change the language of evaluation to evaluate expressions in any desired language with the LANGUAGE command.

The format of the LANGUAGE command, abbreviated LANG, is:



Used without an argument, the LANGUAGE command displays the name of the current host language. For example:

```
> LANGUAGE
Language is RPG.
```

If you want to change the current language to another language, enter the name of the desired language:

```
> LANGUAGE FORTRAN
```

With the command shown above, you can evaluate expressions using FORTRAN syntax.

Note

Changing the language of evaluation does not interfere with the execution of your program when a RESTART or CONTINUE command is given.

The following example uses the LANGUAGE command to change the language of evaluation from FORTRAN to COBOL. Notice how the Debugger reacts to the evaluation syntax in each language:

```
> LANGUAGE
Language is FORTRAN.
> IF ARG1 LESS THAN 0 [: ARG1]
Unrecognized name - "LESS"
ARG1 LESS THAN 0
     ^

> LANGUAGE COBOL
> IF ARG1 LESS THAN 0 [: ARG1]
ARG1 = -1
>
```

In the example shown above, the Debugger does not recognize the operator LESS THAN in FORTRAN, but does in COBOL.

Here is another example that evaluates a Boolean (true/false) value in Pascal and PL/I-G:

```
> LANGUAGE
Language is PASCAL.
> : A
A = false
> LET A = '1'B
Unable to convert character value to Boolean.
A = '1'B
    ^

> LANGUAGE PL1G
> : A
A = '0'b
> LET A = '1'B
> : A
A = '1'b
> LANGUAGE PASCAL
> : A
A = true
>
```

The LANGUAGE command is useful if you are debugging a program that contains a lot of program blocks written in other languages. You may not be familiar with the syntax rules of those languages and may want to evaluate data using the syntax of the language you are most familiar with. In this way, the LANGUAGE command makes it easier to talk to the Debugger in your native tongue. This command is also very useful if you want to take advantage of a particular language's powerful operators and expression evaluation syntax.

REFERENCING DEBUGGER-DEFINED VARIABLES

In Chapter 4, you learned about the invisible $DBG outer block. The $DBG block contains three Debugger-defined variables -- variables that are created by and always known to the Debugger. These variables are:

- $MR
- $COUNT
- $COUNTERS

You can reference the values of these variables during your debugging sessions to provide you with special information.

Note

The $DBG block, which defines these three variables, also defines and knows about all of the PL/I-G, Pascal, and FORTRAN built-in functions that were listed earlier in this chapter. (For more information on the $DBG block, see Chapter 4.)

The $MR Variable

The $MR variable yields the values of the machine registers. Table 6-2 lists the special information that is contained within the variable $MR.

Table 6-2
Machine Registers

Register Category	Description
SAVE-MASK	Bit string indicating which registers have been saved
V	V-mode registers (A, B, L, X, Y, E)
I	I-mode registers (general registers 0 through 7)
BR	Base registers (PB, SB, LB, XB)
KEYS	Process keys

Each time the Debugger is reentered from the user program, the values of the machine registers are saved and stored in $MR.

For example, to display the current saved machine state, enter:

```
> : $MR
$MR.SAVE_MASK = '111111111111'b
$MR.V.A = 3
$MR.V.B = 0
$MR.V.L = 196608
$MR.V.X = 17975
$MR.V.Y = 17424
$MR.V.E = 14
$MR.I.GR0 = 1679948336
$MR.I.GR1 = 0
$MR.I.GR2 = 196608
$MR.I.GR3 = 14
$MR.I.GR4 = 14
$MR.I.GR5 = 1141922692
$MR.I.GR6 = 0
$MR.I.GR7 = 1394231
$MR.BR.PB = 4001(3)/1046
$MR.BR.SB = 4037(3)120244
$MR.BR.LB = 4002(0)/177404
$MR.BR.XB = 4037(3)6734
$MR.BR.KEYS = '0001100000000000'b
```

The $COUNT Variable

The value of the breakpoint counter for the most recent breakpoint or tracepoint is stored in the Debugger-defined variable $COUNT. This variable is most useful in IF expressions in conditional breakpoint action lists. (The breakpoint counter and action lists are defined in Chapter 5.)

Here is an example using $COUNT in a nested breakpoint action list:

```
> BREAKPOINT A\24 [IF $COUNT > 6 [GOTO 26] ELSE [CONTINUE]]
```

The $COUNTERS Variable

The variable $COUNTERS contains counts made during Debugger initialization relating to program size and symbols. These counts are valid only if the -FULL_INIT option is specified on the DBG command line. (See Chapter 13 for information on DBG command line options.) Table 6-3 lists the meanings for the individual counts.

Table 6-3
Meanings of Counts Specified by $COUNTERS

Count	Meaning
STATEMENTS	Number of statements in procedures compiled in debug mode
OUTER_BLOCKS	Number of external program blocks compiled in debug and production modes
TOTAL_BLOCKS	Number of external and internal program blocks compiled in debug and production modes
TOP_LEVEL_SYMBOLS	Number of declared symbols, not including structure members
NON_TOP_LEVEL_SYMBOLS	Total number of structure members
PERMANENT_STORAGE	Number of words allocated by the Debugger for the user program's symbol table
DATA_FILE_SIZE	Size in words of the Debugger data file contained in the program's SEG file

The following example displays the values contained in $COUNTERS:

```
> : $COUNTERS
$COUNTERS.STATEMENTS = 42
$COUNTERS.OUTER_BLOCKS = 1
$COUNTERS.TOTAL_BLOCKS = 2
$COUNTERS.OBJECT_GROUPS = 81
$COUNTERS.DISK_READS = 2
$COUNTERS.TOP_LEVEL_SYMBOLS = 15
$COUNTERS.NON_TOP_LEVEL_SYMBOLS = 10
$COUNTERS.ALLOCATE_CALLS = 96
$COUNTERS.PERMANENT_STORAGE = 1303
$COUNTERS.DATA_FILE_SIZE = 1370
```

The OBJECT_GROUPS, DISK_READS, and ALLOCATE_CALLS counts are not provided for the user, and they should, therefore, be ignored.

REFERENCING EXTERNAL VARIABLES

The Debugger provides an alternative for referencing PL/I-G and Pascal external variables, that is, variables declared to be global to external program blocks. The format of this alternative reference is:

```
$EXTERNAL\variable
```

The variable is a PL/I-G variable declared as STATIC EXTERNAL or a Pascal external variable declared with surrounding {$E+} and {$E-} compiler switches.

Here is an example:

```
> : $EXTERNAL\TIMEDATE.DATE
DATE.DAY = '20'
DATE.MONTH = '08'
DATE.YEAR = '79'
```

Variables must be qualified at the top level when used with $EXTERNAL. This means that $EXTERNAL/DATE would not work. You must enter TIMEDATE.DATE.

You can produce the same result by specifying the program block that corresponds to the variable or else change the evaluation environment with the ENVIRONMENT command before referencing the variable.

7

Single Stepping and Calling Program Blocks

Commands discussed in this chapter:

STEP	OUT
STEPIN	CALL
IN	

INTRODUCTION

This chapter describes the following special program control features:

- Single stepping — executing one or more statements at a time and stepping across, into, or out of a called program block with the STEP, STEPIN, IN, and OUT commands.
- Calling a program block from Debugger command level with the CALL command.

SINGLE STEPPING

Single stepping allows you to execute a program one or more statements at a time. It also can step across, into, and out of called blocks.

The four single stepping commands, which are described in the following sections, are:

- STEP -- executes a given number of statements at a time and steps across a program block.
- STEPIN -- also executes a given number of statements but steps into a program block.

- IN -- continues execution until the next program block is called.
- OUT -- continues execution until the current program block returns.

The STEP Command

The STEP command executes one or more statements at a time. STEP suspends program execution after a given number of statements in the current program block without referencing statement numbers or labels. STEP requires only the number of statements or "steps" to execute before execution suspends. BREAKPOINT would require a statement identifier and a CONTINUE command to do the same thing.

The format of the STEP command, abbreviated S, is:

```
STEP [value]
```

The value is the number of statements you want to execute before suspending execution. If no value is specified, one statement is executed by default.

Consider the following FORTRAN IV program, which has one subroutine named WRITEN:

```
 1:       CALL WRITEN (3)
 2:       CALL WRITEN (4)
 3:       CALL WRITEN (5)
 4:       CALL EXIT
 5:       END
 6:
 7:       SUBROUTINE WRITEN (N)
 8:       WRITE (1, 10) N
 9: 10    FORMAT ('ARGUMENT IS ', I3)
10:       RETURN
11:       END
```

You can restart program execution with RESTART and suspend execution prior to the first executable statement by entering a STEP command on the same line:

```
> RESTART STEP

**** "step" completion at $MAIN\1
>
```

The step completion message indicates that the next statement to be executed is on source line number 1 in the main program.

If you have just entered the Debugger from PRIMOS command level, you can accomplish the same thing by entering just the STEP command without RESTART:

```
> STEP

**** "step" completion at $MAIN\1
>
```

To execute the first statement in the program, stepping across the called block, enter:

```
> STEP
ARGUMENT IS   3

**** "step" completion at $MAIN\2
>
```

The ARGUMENT IS 3 message was displayed by subroutine WRITEN, which was called from the main program. The STEP command, therefore, neither prevents nor interrupts the execution of a called program block; execution continues until the block returns to the calling program -- in this case, main program. Execution is then suspended again at the next executable statement, which is line 2 in $MAIN. Hence, the term across simply means that when a call statement is encountered, the entire called program block is executed without interruption until the called block returns.

Here is an example of a STEP command that uses the argument 2; that is, two statements are executed at a time instead of one:

```
> RESTART STEP 2
ARGUMENT IS   3

**** "step" completion at $MAIN\2
> STEP 2
ARGUMENT IS   4
ARGUMENT IS   5

**** "step" completion at $MAIN\4
>
```

There are some minor differences in the interpretation of value, depending on where the STEP or STEPIN command is given:

- If the execution environment pointer represents a statement, that is, if execution is suspended at a statement, the value is the number of statements that will be executed.

- If the execution environment pointer represents an entry to a program block, the value 1 means execution suspends prior to the first statement, the value 2 means the first statement is executed and suspension occurs prior to the second statement, and so on.

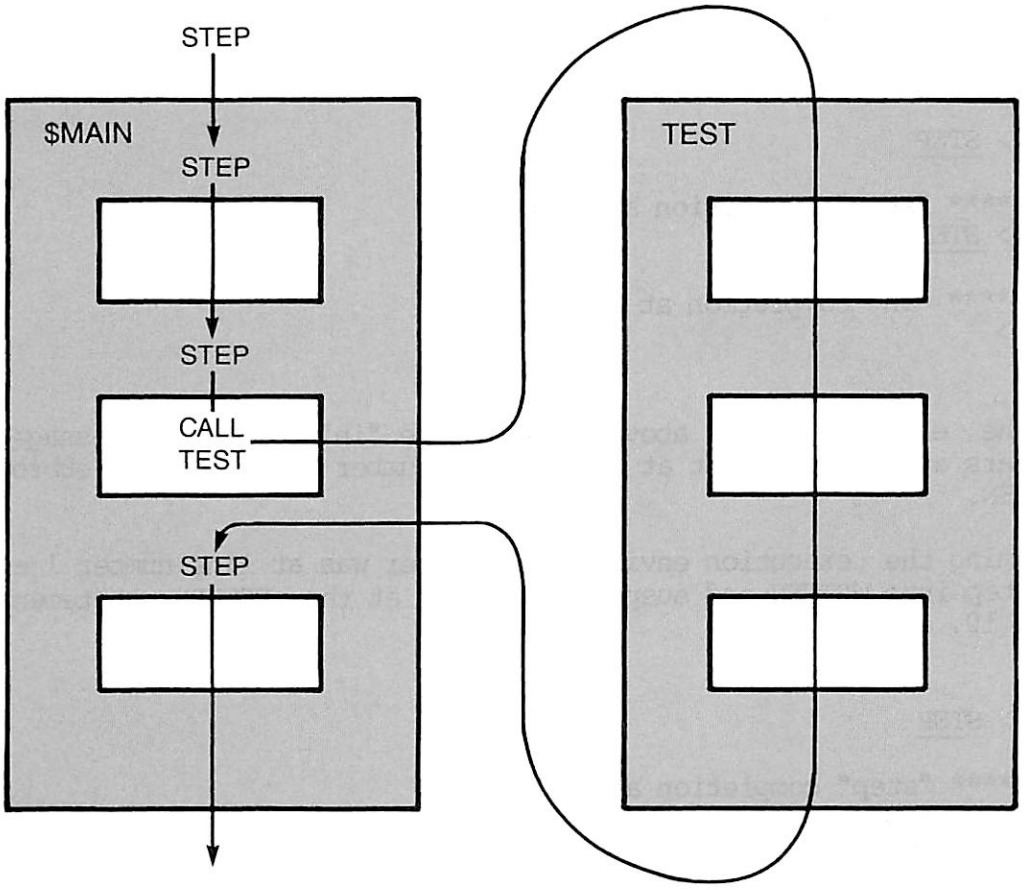
- If the execution environment pointer represents an exit from a program block, a value of 1 signifies that the remainder of the statement (if any) that caused entry to the just-returned program block is to be executed and that execution is to suspend prior to the next statement. A value of 2 indicates that, in addition to the above, the next statement is to be executed, and so on.

- If the execution environment pointer represents the Debugger default on-unit for a particular condition, a value of 1 indicates that the on-unit is to return and that execution is to suspend prior to the next statement. A value of 2 indicates that the on-unit is to return and the next statement is to be executed, and so on.

- If the execution environment pointer is undefined, a RESTART STEP or a RESTART STEPIN command is implied.

Figure 7-1 illustrates the STEP command. In this figure, the word STEP indicates that execution has suspended at a step completion. Each white box represents one executable statement. Notice that execution does not stop within the called block.

The Step Counter: Invisible to the user is the step counter. This counter, which is internal to the Debugger, contains the number of statements left to be executed before execution is suspended again and control returns to Debugger command level. This number is the value that you specify whenever you use a STEP or STEPIN command. If you do not specify a value, the counter is set to 1 by default. When you enter a STEP or STEPIN command, the counter is decremented by 1 until the counter reaches 0. When the count becomes 0, execution is suspended again and control returns to Debugger command level.



The STEP Command
(Each white box represents one executable statement)
Figure 7-1

The STEPIN Command

STEPIN works like the STEP command, except that when STEPIN encounters a call statement, STEPIN steps into the called block and continues single stepping — suspending execution at statements within the called block.

STEPIN executes and single steps every statement, whether or not the statement is contained within the current block or the called block.

The format of the STEPIN command, abbreviated SI, is:

```
STEPIN [value]
```

The value is the number of statements you want to execute before suspending execution. If no value is specified, one statement is executed by default.

Here is an example of the STEPIN command used with the sample FORTRAN program shown in the previous section:

```
> STEP

**** "step" completion at $MAIN\1
> STEPIN

**** "in" completion at WRITEN\8
>
```

In the example shown above, notice the "in" completion message that appears at the statement at source line number 8 within the subroutine WRITEN.

Assuming the execution environment pointer was at line number 1 again, to step into WRITEN and suspend execution at the RETURN statement on line 10, enter:

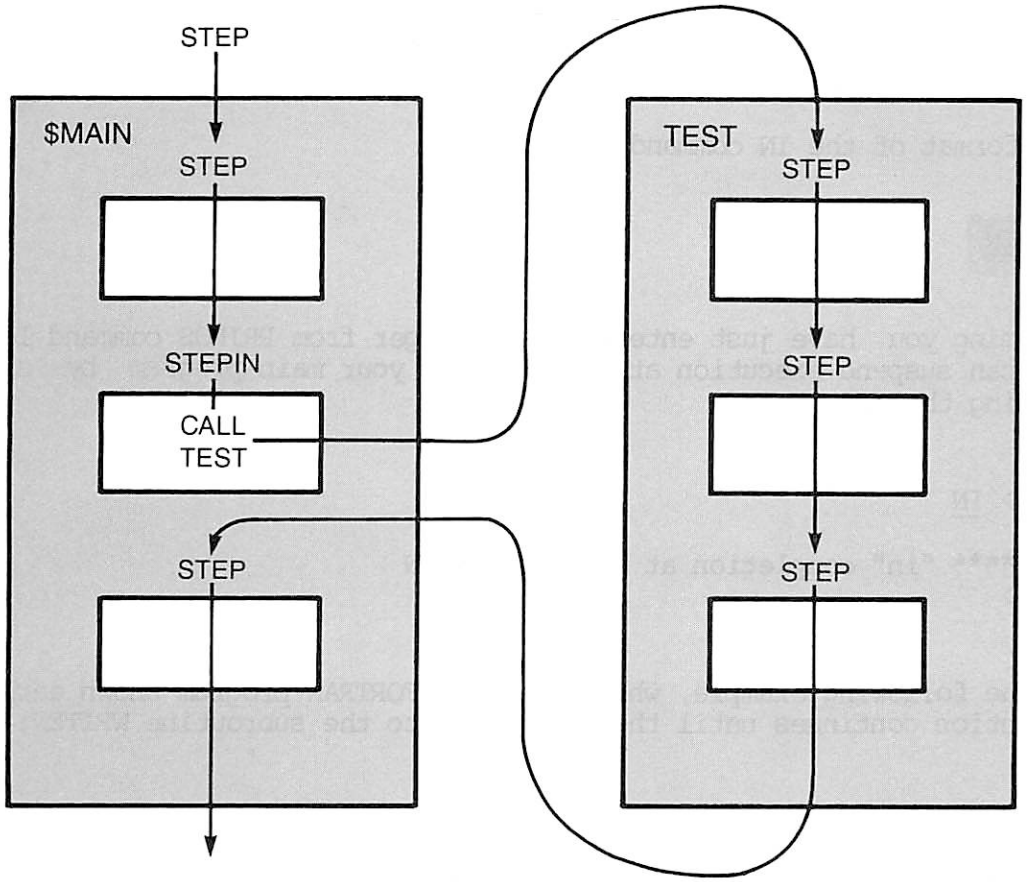
```
> STEP

**** "step" completion at $MAIN\1
> STEPIN 3
ARGUMENT IS   3

**** "in" completion at WRITEN\10 ($10+1)
>
```

The three statements that are executed with this command are on source line numbers 1, 8, and 9.

Note

If the program block specified by the execution environment pointer at the time the STEP or STEPIN command is given returns, single stepping is terminated and control returns to Debugger command level.

Figure 7-2 illustrates the STEPIN command. In this figure, the word STEP or STEPIN indicates that execution has been suspended. Each white box represents one executable statement.



The STEPIN Command
(Each white box represents one executable statement)
Figure 7-2

The IN Command

The IN command continues program execution until the next program block is called, then suspends execution at the entry to that block, immediately before the first executable statement.

IN is similar to an entry breakpoint, except that you specify a particular block for an entry breakpoint. IN simply suspends execution at the next program block that is called, whatever it happens to be. (For information on entry breakpoints, see Chapter 5.)

If you enter an IN command when execution is suspended inside a program block that returns before another program block is called or before program execution ends, then execution will suspend at the exit of the current program block, immediately before the next executable statement. The effect is similar to an exit breakpoint. (See Chapter 5.)

If you enter an IN command when no more blocks are to be called or returned, execution continues until it is suspended for some other reason or until program execution completes.

If the execution environment is undefined, a RESTART IN command sequence is implied, that is, an IN command will produce the same result as RESTART IN.

The format of the IN command is:

```
IN
```

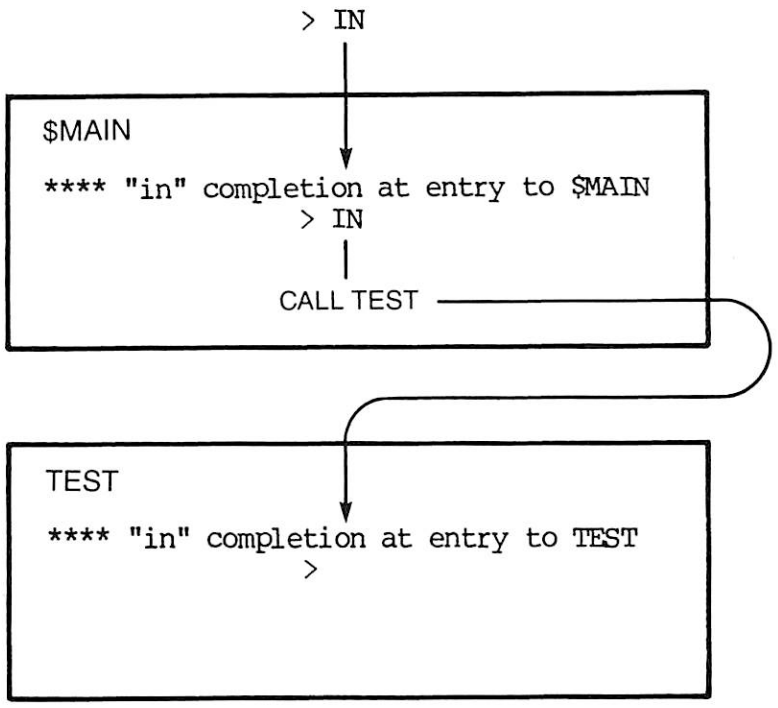
Assuming you have just entered the Debugger from PRIMOS command level, you can suspend execution at the entry of your main program by simply issuing the IN command:

```
> IN

**** "in" completion at entry to $MAIN
>
```

In the following example, which uses the FORTRAN program shown earlier, execution continues until the first call to the subroutine WRITEN:

```
> IN

**** "in" completion at entry to $MAIN
> IN

**** "in" completion at entry to WRITEN
>
```

Figure 7-3 illustrates the IN command.



The IN Command
Figure 7-3

The OUT Command

The OUT command continues program execution until the current program block, defined by the execution environment pointer, returns and execution suspends at the exit of that block.

OUT is similar to an exit breakpoint, except that you specify a particular block for the exit breakpoint. OUT simply suspends execution at the exit of the current block, whatever it happens to be.

The format of the OUT command is:

```
OUT
```

Assume you are suspended at the entry to WRITEN in the sample FORTRAN program used earlier. To continue program execution until the block returns, enter:

```
**** breakpointed at entry to WRITEN
> OUT
ARGUMENT IS   3

**** "out" completion at exit from WRITEN into $MAIN\2
>
```

Figure 7-4 illustrates the OUT command.

CALLING PROGRAM BLOCKS

With the Debugger's CALL command, you can call a program block from Debugger command level. A program block could be, for example, a FORTRAN function, Pascal procedure, COBOL program, or RPG subroutine. (Program blocks are defined in Chapter 4.)

The CALL command is useful when you are having problems with a certain program block and want to call that block again or "replay" it to discover those problems. When you replay your block, you can pass correct or incorrect arguments to it, suspend execution within it, evaluate an expression in it, trace a value through it, or use other Debugger features while you study the output. CALL moves the evaluation environment pointer to the block that you call.

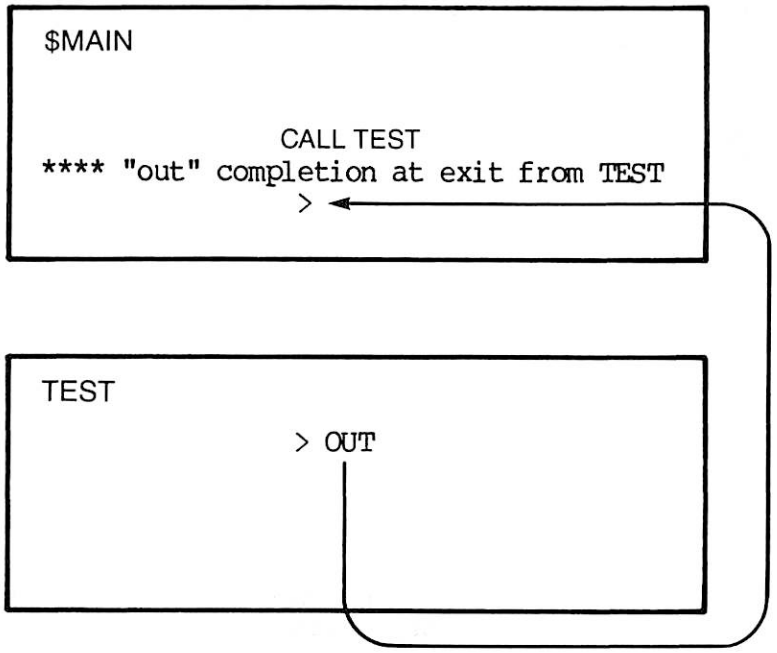
CALL is useful because it can test a single subroutine before the rest of the program is finished. CALL can also call utility routines, which might help during debugging — for example, displaying an English interpretation of some internal data structure.

The format of the CALL command is:

```
CALL variable [(argument-list)]
```

The variable is the name of the program block you want to call, as described in Chapter 4.

The argument-list is a list of expressions or "parameters" that are supplied or "passed" to the program block according to the rules of the host language.



The OUT Command
Figure 7-4

When the CALL command is given, the Debugger evaluates the value of each argument, then calls the block, supplying the resulting values as arguments. For example:

```
> CALL STRSORT (LIST, KEY1 + 2)
```

If the user block is a function, the Debugger displays the returned value on the terminal:

```
> CALL GETIDX
Returned value: 11
```

If you want to call a block that is not declared within the block corresponding to the evaluation environment pointer, you must specify the desired block name. For example, suppose you wanted to call a block named MULT, which was not defined in the current evaluation environment. You would enter:

```
> CALL MULT\MULT (2, 5)
The product is      10
```

If you had not entered the MULT\, you would have received an error message:

```
> CALL MULT
No applicable declaration found for MULT
MULT
     ^
```

If you had not supplied the correct number of arguments to the block, you would have received an error message:

```
> CALL MULT\MULT (5)
Too few arguments to "MULT":  2 expected, 1 supplied.
MULT\MULT (5)
          ^
```

An alternative to specifying the same block name (MULT) is specifying the name of the next outer block in which MULT is declared. For example, if MULT were declared within another block named CALC, you could enter:

```
> CALL CALC\MULT (2, 5)
The product is      10
```

If execution is suspended with the called block, the Debugger displays a special command prompt, which has the following format:

```
(call-level:block-name) >
```

The call-level indicates the level of invocation of the Debugger. The block-name is the name of the called user block. For example:

```
> BREAKPOINT INSERT\47
> BREAKPOINT RELINK\\ENTRY
> CALL INSERT (NEWITEM)

**** breakpointed at INSERT\47
(1:INSERT) > CALL RELINK (PTR)

**** breakpointed at entry to INSERT.RELINK
(2:RELINK) >
```

In effect, this prompt is telling you that you are suspended within a program block that has been executed with the CALL command.

CALL and the Call/Return Stack

Each CALL command causes a new invocation of the Debugger. Other calls not related to Debugger CALL invocations include user block calls, such as your main program's call to another block, and calls to PRIMOS error-handling conditions when the Debugger encounters serious runtime errors. Each call and invocation is placed on a stack known as the call/return stack.

The call/return stack, therefore, is a list of currently active program blocks. Each call or invocation that is put on the stack is known as a call/return stack frame. A frame that is created by a CALL command invocation is known as a Debugger CALL frame.

Looking at the call/return stack is useful because it lets you see how program control passed around during the execution of your program. The Debugger's TRACEBACK command can display the contents of the call/return stack. The UNWIND command erases all frames from the call/return stack. (For complete information on the call/return stack and the TRACEBACK and UNWIND commands, see Chapter 8.)

8
Tracing

Commands discussed in this chapter:

TRACEPOINT	WATCH	STRACE
LIST	WATCHLIST	TRACEBACK
LISTALL	UNWATCH	UNWIND
CLEAR	VTRACE	
CLEARALL	ETRACE	

INTRODUCTION

This chapter describes and demonstrates all of the Debugger's tracing features, which include:

- Setting trace messages, known as tracepoints, at a statement or at the entry to or exit from a program block with the TRACEPOINT command.
- Displaying and deleting tracepoints with the LIST, LISTALL, CLEAR, and CLEARALL commands. (See also Chapter 5.)
- Value tracing — tracing the value of a variable and displaying a message any time the value changes with the WATCH, UNWATCH, WATCHLIST, and VTRACE commands.
- Entry tracing — setting a trace message at the entry to or exit from a program block with the ETRACE command.
- Statement tracing — setting a trace message at every statement or every labelled statement with the STRACE command.
- Tracing the currently active program blocks (the call/return stack) with the TRACEBACK command.

This chapter also describes the program control command, UNWIND, which erases the call/return stack.

SETTING TRACEPOINTS

A tracepoint is like a breakpoint. However, instead of suspending execution on a statement, entry to, or exit from a program block, a tracepoint just displays a message each time the statement, entry, or exit is encountered. Execution does not stop.

Setting a tracepoint is useful for calling attention to a particular spot in your program execution.

The format of the TRACEPOINT command, abbreviated TRA, is:

```
TRACEPOINT [breakpoint-identifier] [-AFTER value]
           [-BEFORE value] [-EVERY value] [-COUNT value]

           [-IGNORE  ]
           [-NIGNORE ]
```

The breakpoint-identifier is the statement, entry, or exit on which you want to set a tracepoint. (The breakpoint-identifier is explained in more detail in Chapter 5.)

As you can see, the syntax and features of TRACEPOINT are very much like BREAKPOINT, except that TRACEPOINT does not allow action lists or the -EDIT option.

The following example demonstrates the TRACEPOINT command and the resulting tracepoint messages:

```
> TRACEPOINT 24
> TRACEPOINT DAYS_IN_MONTH\10
> RESTART
**** $$MAIN$$\24
**** $$MAIN$$.DAYS_IN_MONTH\10

**** Program execution complete.
>
```

Note

For a complete explanation and demonstration of the tracepoint options, see the section on setting breakpoints in Chapter 5. Just substitute a tracepoint wherever a breakpoint is used.

DISPLAYING AND DELETING TRACEPOINTS

The Debugger's LIST and LISTALL commands display the attributes of tracepoints as well as breakpoints. The LIST command displays the attributes of one tracepoint:

```
> LIST 14
Type  Location
tra   $$MAIN$$\14, count = 0
```

The LISTALL command, abbreviated LSTA, can list the attributes of all tracepoints:

```
> LISTALL -TRACEPOINTS
Tracepoints at:
      $$MAIN$$\24, count = 0
      $$MAIN$$.DAYS_IN_MONTH\17, count = 0
      $$MAIN$$.DAYS_IN_MONTH\10, count = 0
>
```

The Debugger's CLEAR and CLEARALL commands delete tracepoints as well as breakpoints. The CLEAR command, abbreviated CLR, deletes a tracepoint:

```
> CLEAR 14
```

The CLEARALL command, abbreviated CLRA, can delete all tracepoints:

```
> CLEARALL -TRACEPOINTS
```

Note

For a complete explanation and demonstration of the LIST, LISTALL, CLEAR, and CLEARALL commands, see Chapter 5. These commands function the same with tracepoints as they do with breakpoints.

VALUE TRACING

In Chapter 3, you learned how to trace the changing value of a variable through the execution of your program with the WATCH command. This process is called value tracing. Value tracing is useful for finding out when and how unusual values are assigned to variables during execution. Pinpointing the spot where a certain value has been assigned or not assigned can help you correct a program error.

During value tracing, a message is displayed each time the value of a specified variable changes. The Debugger displays the former value, the new value, and the program location where the change was detected. The WATCH command specifies which variables are to be traced.

The WATCH Command

You must specify which variables you want to trace during the execution of your program with the WATCH command. The format of the WATCH command, abbreviated WA, is:

```
WATCH variable-1 [, variable-2 ... ]
```

where variable-1, variable-2, and so on, are the variables that you want to trace or "watch."

In order to indicate that an automatic variable should be watched only in a specific activation of a program block, you may give an activation number. If the activation number is omitted, the variable will be watched in all activations.

When you use the WATCH command to trace variables, these variables are placed in an internal Debugger table known as the watch list. You can add variables to the watch list any time. They remain in the watch list until you remove them with the UNWATCH command, described later in this chapter. The WATCHLIST command, also described later in this chapter, displays the names of variables in the watch list.

Consider the following FORTRAN program:

```
1:        INTEGER A
2:        DIMENSION A (3)
3:
4:        N = 0
5:        DO 10 I = 1, 3
6:        N = N + 1
7: 10     A(I) = N
8:        STOP
9:        END
```

Here is an example of the WATCH command used to trace the changing values of the variables N and A in this program:

```
> WATCH N, A
> RESTART
The value of $MAIN\N has been changed at $MAIN\7 ($10)
  from 0
  to   1
The following values in $MAIN\A have been changed at $MAIN\6
  A(1) from 0
       to   1
The value of $MAIN\N has been changed at $MAIN\7 ($10)
  from 1
  to   2
The following values in $MAIN\A have been changed at $MAIN\6
  A(2) from 0
       to   2
The value of $MAIN\N has been changed at $MAIN\7 ($10)
  from 2
  to   3
The following values in $MAIN\A have been changed at $MAIN\8 ($10+1)
  A(3) from 0
       to   3

**** STOP

Program stop at $MAIN\8 ($10+1).
>
```

In the example shown above, notice how a message is printed whenever an element in the array A changes. In general, a message is printed only for those elements in arrays or structures that change.

In this next example, the WATCH command is used to trace the changing value of the variable PAY in an RPG program:

```
> WATCH PAY
> RESTART
The value of RPG$MAIN\PAY has been changed at RPG$MAIN\11
  from 0.00
  to   146.80
The value of RPG$MAIN\PAY has been changed at RPG$MAIN.SUBRU\16
  from 146.80
  to   180.00
The value of RPG$MAIN\PAY has been changed at RPG$MAIN.SUBRU\16
  from 180.00
  to   200.00
The value of RPG$MAIN\PAY has been changed at RPG$MAIN\11
  from 200.00
  to   119.60
The value of RPG$MAIN\PAY has been changed at RPG$MAIN.SUBRU\16
  from 119.60
  to   270.00
```

You can watch any portion of an array or structure. For example, to watch the array cross section specified by TABLE (2, *), you would enter:

```
> WATCH TABLE (2, *)
```

To watch variables in another block, enter the desired block name:

```
> WATCH SUBR1\Y, SUBR2\TEST(8)
```

Value tracing is useful in recursive program blocks. For example, to watch the variable N in only the third activation of a block named FACTORIAL, enter:

```
> WATCH FACTORIAL\3\N
```

The way in which watched variables are monitored differs for each storage class. Specifically:

- The value of a static variable is saved when the WATCH command is given and is watched throughout the debugging session unless it is removed by the UNWATCH command. (All COBOL variables are static.)

- Each generation of an automatic variable is watched unless a specific generation is designated by the user. The value is saved upon program block entry and monitored until the program block becomes inactive.

- A PL/I-G based variable or Pascal dynamic variable is saved and watched according to the storage class of its locator (pointer). That is, qualification by a static or "constant" locator causes the based or dynamic variable to be watched as though its storage class were static, by an automatic locator as though its storage class were automatic.

- PL/I-G controlled variables cannot be watched.

Watching PL/I-G, Pascal, and C Pointer Variables: When a PL/I-G based, Pascal dynamic, or a C pointer variable is added to the watch list, the locator or pointer expression is evaluated at the time the WATCH command is given, and the resulting address is used as the locator or pointer as long as the variable remains in the watch list. Therefore, at the time a variable is added to the watch list:

- The block containing the locator/pointer must be active if the locator or pointer is automatic.

- The locator/pointer must be initialized to the desired address. (This applies even if the variable is not automatic.)

For example, consider the following PL/I-G program:

```
 1: P1 : PROCEDURE;
 2:
 3:       DECLARE LAST_NAME CHARACTER (15);
 4:       DECLARE LOCATOR   POINTER;
 5:       DECLARE STRING8   BASED CHARACTER (8);
 6:
 7:
 8:       LAST_NAME = 'Washington     ';
 9:       LOCATOR = ADDR (LAST_NAME);
10:       LAST_NAME = 'Jefferson      ';
11: END; /* P1 */
```

In order to watch LOCATOR -> STRING8, P1 must be active, and LOCATOR must be initialized. Thus, the WATCH command cannot be given until after execution of the statement on source line number 9 as follows:

```
**** breakpointed at P1\10
> WATCH LOCATOR -> STRING8
> CONTINUE
The value of P1\LOCATOR -> STRING8 has been changed at P1\11
  from 'Washingt'
  to   'Jefferso'
```

Alternatively, you may watch STRING8 any time after entry to P1 by specifying a constant pointer:

```
> WATCH ADDR (LAST_NAME) -> STRING8
```

Even if the value of the locator or pointer in the user program changes value, the address computed when the based, dynamic, or pointer variable was added to the watch list will continue to be used for value tracing.

In the sections that follow, you will learn how to display the variables in the watch list, remove variables from the watch list, turn off value tracing without disturbing the contents of the watch list, and use value tracing only on the entries and exits to program blocks. These features involve the Debugger's WATCHLIST, UNWATCH, and VTRACE commands.

The WATCHLIST Command

The WATCHLIST command displays the names of the variables currently in the watch list. The format of the WATCHLIST command, abbreviated WL, is:

```
WATCHLIST
```

Here is an example of the WATCHLIST command:

```
> WATCHLIST
MAIN\X
MAIN\LIST
MAIN\TABLE(2,*)
SUBR1\Y
SUBR2\TEST(8)
FACTORIAL\3\N
```

The UNWATCH Command

The UNWATCH command removes one or more variables from the watch list. The format of the UNWATCH command, abbreviated UW, is:

```
UNWATCH { variable-1 [, variable-2 ... ] }
        { -ALL                           }
```

where variable-1, variable-2, and so on, are variables you want to remove from the watch list.

If you specify the -ALL option, all variables are removed from the watch list.

As in the WATCH command, if the program block name is omitted, the block corresponding to the current evaluation environment is used.

To remove the variables LIST, Y, and TEST(8) from the watch list, enter:

```
> UNWATCH MAIN\LIST, SUBR1\Y, SUBR2\TEST(8)
```

To remove all variables from the watch list, enter:

```
> UNWATCH -ALL
```

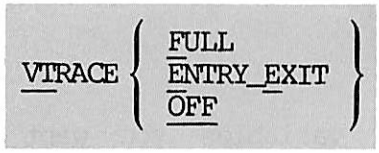
You can discontinue value tracing without "unwatching" all of the variables in the watch list by entering the VTRACE OFF command, which is explained in the next section.

Note

The UNWATCH command does not disable value tracing unless the watch list becomes empty.

The VTRACE Command

You can disable, enable, or change the frequency of value tracing with the VTRACE command. The format of the VTRACE command, abbreviated VT, is:



If you specify the ENTRY_EXIT option, abbreviated EE, the values of variables in the watch list are compared only at the entry to and exit from each program block. This is useful for narrowing down the area in which a variable is modified, since entry/exit value tracing is substantially faster than full value tracing. For example:

```
> WATCH BEST_PRESIDENT
> VTRACE ENTRY_EXIT
> RESTART
The value of A\BEST_PRESIDENT has been changed at entry to A.B
  from ''
  to   'Truman'
The value of A\BEST_PRESIDENT has been changed at entry to A.B.C
  from 'Truman'
  to   'Nixon '

**** Program execution complete.
>
```

If you specify the OFF option, value tracing does not occur, but the contents of the watch list are undisturbed. For example:

```
> WATCH N, A
> VTRACE OFF
> RESTART

**** STOP

Program stop at $MAIN\8 ($10+1).
> WATCHLIST
$MAIN\A
$MAIN\N
>
```

If you specify the FULL option, abbreviated F, value tracing is enabled once again at every statement. That is, the values of variables in the watch list are compared at every executable statement.

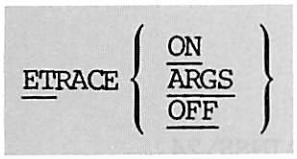
Note

When the WATCH command is entered, the variables specified are watched at every statement unless you use the VTRACE command to enable entry/exit value tracing or to turn off value tracing completely. If value tracing is turned on again after being turned off, the saved value of each watched variable is immediately updated to its current value.

ENTRY TRACING

During your debugging sessions, you may want to display a trace message each time a program block is called or returned. This process is known as entry tracing, which is enabled with the ETRACE command.

The format of the ETRACE command, abbreviated ET, is:



If you specify the ON option, a trace message is displayed when each program block is called and returned. Here is an example of the ETRACE ON command used with a Pascal program that has one function named DAYS_IN_MONTH:

```
> ETRACE ON
> RESTART
**** entry to $$MAIN$$
**** entry to $$MAIN$$.DAYS_IN_MONTH
**** exit from $$MAIN$$.DAYS_IN_MONTH into $$MAIN$$\24
**** exit from $$MAIN$$

**** Program execution complete.
>
```

If you specify the ARGS option, not only do trace messages appear at the entry and exits to called program blocks, but the values of arguments passed to each called block are also displayed at each entry. The argument values are not displayed at the exits. Here is an example of the ETRACE ARGS command:

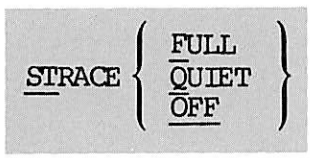
```
> ETRACE ARGS
> RESTART
**** entry to $$MAIN$$
**** entry to $$MAIN$$.DAYS_IN_MONTH
MONTH_IN = SEPTEMBER
YEAR_IN = 1983
**** exit from $$MAIN$$.DAYS_IN_MONTH into $$MAIN$$\24
**** exit from $$MAIN$$

**** Program execution complete.
>
```

If you specify the OFF option, entry tracing is turned off. For example:

```
> ETRACE ON
> RESTART
**** entry to $$MAIN$$
**** entry to $$MAIN$$.DAYS_IN_MONTH
**** exit from $$MAIN$$.DAYS_IN_MONTH into $$MAIN$$\24
**** exit from $$MAIN$$

**** Program execution complete.
> ETRACE OFF
> RESTART

**** Program execution complete.
>
```

STATEMENT TRACING

At some point you might want to display a trace message prior to the execution of every statement in your program. This feature is known as statement tracing, which is invoked with the STRACE command. The format of the STRACE command, abbreviated ST, is:



If you specify the FULL option, abbreviated F, the Debugger displays a trace message prior to the execution of every statement in your program. For example:

```
> STRACE FULL
> RESTART
**** $MAIN\4
**** $MAIN\5
**** $MAIN\6
**** $MAIN\7 ($10)
**** $MAIN\6
**** $MAIN\7 ($10)
**** $MAIN\6
**** $MAIN\7 ($10)
**** $MAIN\8 ($10+1)

**** STOP

Program stop at $MAIN\8 ($10+1).
>
```

If you specify the QUIET option, abbreviated Q, the Debugger displays a trace message only prior to the execution of each labelled statement. (Chapter 4 defines labels for each language.) For example:

```
> STRACE QUIET
> RESTART
**** $MAIN\7 ($10)
**** $MAIN\7 ($10)
**** $MAIN\7 ($10)

**** STOP

Program stop at $MAIN\8 ($10+1).
>
```

The OFF option turns off statement tracing.

Note

Entry tracing and statement tracing are independent of one another and may be enabled or disabled at any point during the debugging session. Furthermore, use of these tracing features is applicable to the entire debugging environment. You cannot, strictly speaking, enable statement or entry tracing for one particular routine. This effect can be achieved using breakpoints. You may set a breakpoint at the entry of a program block specifying an action list that contains the appropriate trace commands. In order to continue program execution, the trace commands should be followed by a CONTINUE command. Similarly, tracing can then be disabled by setting a corresponding exit breakpoint with the appropriate trace disable commands contained within an action list.

TRACING YOUR ACTIVE PROGRAM BLOCKS

While your program executes, PRIMOS keeps a list of program blocks that are currently active in your program's execution. The list could include, for example, the Debugger's call to your main program when you enter a RESTART command, or your main program's call to a procedure, function, or subroutine.

When such a call is encountered, the call is placed on an internal stack known as the call/return stack. Each call that is placed onto the stack is known as a call/return stack frame. You can look at the contents of this stack with the TRACEBACK command, which is described later in this section.

Looking at the contents of the call/return stack, frame by frame, is useful for seeing the sequence of your program block calls. For instance, you might see that one of your subroutines or functions was called at the wrong place at the wrong time.

When you look at the call/return stack with the TRACEBACK command, each frame can provide you with the following information:

- The name of the block that was called, known as the owner block
- The name of the calling block and the source line number on which the call was made
- The name of the block and the source line number on which the call returns
- The internal addresses of the call and the return
- On-unit information
- The stack frame number

The TRACEBACK command also gives you the current location of the execution environment pointer.

By now you are wondering what the stack looks like. Assume you just placed a breakpoint at the entry to a subroutine named TEST. When execution is suspended at that entry, you want to look at the call/return stack. The stack would look something like this:

```
**** breakpointed at entry to $MAIN.TEST
> TRACEBACK
Currently at entry to TEST.
Stack contains 3 frames.

  3: Owner is "TEST".
     Called from $MAIN\3, returns to $MAIN\4.

  2: Owner is "$MAIN".
     Called from debugger, returns to debugger.

  1: Debugger-owned frame.
```

In the example shown above, notice how each frame is listed from the most recent frame (3) to the oldest frame (1). In other words, when you read the stack from top to bottom, you are going back in time to the beginning of execution. The Debugger assigns a number to each frame. The least recent is frame 1, and the most recent is the number of frames on the stack. As you can see, the frame number is printed in the lefthand margin.

Types of Frames

The call/return stack can provide various types of information in each of its frames. This information may or may not be useful to you, depending on your needs and depending on how experienced a programmer you are. A call/return stack frame provides one of five types of information and has one of the following formats:

- User-owned call frame. For example:

```
  3: Owner is "TEST".
     Called from $MAIN\43, returns to $MAIN\45.
```

  User-owned frames are identified by the program block name. For each user-owned frame, the frame owner (name of the program block that owns the frame), call location, and return location are displayed. You may request that memory addresses that

correspond to these program locations also be displayed. Also, you can request that the names of the on-units declared within the frames be displayed.

- Condition frame. For example:

```
4: CONDITION FRAME for "ARITH$",
   returns to location 13(3)/1720.
   Condition initiated by hardware fault,
   detected at $MAIN\3.
   Corrective action by on-unit is required.
```

For each condition frame, the information displayed includes the condition name, program location of where it was raised, program location of where it returns, signal source (software or hardware related), and crawlout information, where applicable.

Note

COBOL does not allow on-units, which are routines to handle error conditions such as division by zero and arithmetic overflow. The COBOL 74 Reference Guide discusses how to handle some common error conditions within COBOL. For other languages, see the chapter on the condition mechanism in the Subroutines Reference Guide.

- Fault frame. For example:

```
3: FAULT FRAME; fault type "ARITH" (50)
   Fault returns to $MAIN\3.
   Fault code 403, fault address 0(0)/0.
```

For each fault frame, information displayed includes the fault type, code, and address, and the program location of where the fault returns.

- Debugger-owned frame. For example:

```
14: Debugger-owned frames, through frame 4.
```

Debugger-owned frames are normally listed in "compressed" form, meaning that the traceback will indicate only that several debugger-owned frames exist between two user-owned frames or at the root of the stack. Debugger-owned frames at the top of the stack are never displayed. You may request that these debugger

frames be displayed, listing the linkbase addresses of each frame owner and procedure base called-from and returns-to addresses. (This information, which is meaningless to most users, is discussed in the SEG and LOAD Reference Guide (Rev. 19.2) and in the System Architecture Reference Guide.)

- Debugger CALL frame. For example:

```
16: Debugger CALL frame.
    Called from debugger, returns to debugger.
```

A Debugger CALL frame represents a program block invoked with the Debugger's CALL command. (See Chapter 7.) CALL frames are generated immediately prior to user-owned frames as a result of calling a user program block from Debugger command level with the CALL command.

Condition frames and fault frames are generated by error conditions occurring in your program execution, and they provide internal information on the operating system's attempts to deal with the errors encountered. (For more information on conditions and faults, see the Prime User's Guide and the Subroutines Reference Guide.)

The TRACEBACK command, which allows you to see the contents of the call/return stack, is fully explained in the following section.

The TRACEBACK Command

The TRACEBACK command allows you to look at the contents of the call/return stack and the useful information that the stack provides. (See the previous section.)

The format of the TRACEBACK command, abbreviated TB, is:

```
TRACEBACK [-FRAMES value [-LEAST_RECENT]] [-FROM value] [-TO value]
          [-REVERSE] [-DBG] [-ONUNITS] [-ADDRESSES]
```

The value is a positive non-zero integer.

If you use the TRACEBACK command by itself, with no arguments, all frames on the stack are printed in order from the most recent frame to the least recent frame. For example:

```
> TRACEBACK
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  6: Owner is "TEST.SWITCH".
     Called from TEST.SORT\69, returns to TEST.SORT\69.

  5: Owner is "TEST.SORT".
     Called from TEST\77, returns to TEST\78.

  4: Owner is "TEST".
     Called from debugger, returns to debugger.

  3: Debugger-owned frames, thru frame 1.
>
```

If you specify the -FRAMES option, the number of frames displayed is limited to the specified value. These will be the most recent frames, unless you specify the -LEAST_RECENT option, in which case they will be the least recent frames. For example:

```
> TRACEBACK -FRAMES 2
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  6: Owner is "TEST.SWITCH".
     Called from TEST.SORT\69, returns to TEST.SORT\69.

  5: Owner is "TEST.SORT".
     Called from TEST\77, returns to TEST\78.
>
```

If you specify the -FROM option, the traceback starts from the frame number (the value) that follows -FROM. If you specify the -TO option, the last frame that is displayed is the frame number represented by value. Here is an example of both the -FROM and -TO options:

```
> TRACEBACK -FROM 5 -TO 4
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  5: Owner is "TEST.SORT".
     Called from TEST\77, returns to TEST\78.

  4: Owner is "TEST".
     Called from debugger, returns to debugger.
>
```

If you specify the -REVERSE option, the frames are listed in reverse order from least recent to most recent. For example:

```
> TRACEBACK -REVERSE
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  1: Debugger-owned frames, thru frame 3.

  4: Owner is "TEST".
     Called from debugger, returns to debugger.

  5: Owner is "TEST.SORT".
     Called from TEST\77, returns to TEST\78.

  6: Owner is "TEST.SWITCH".
     Called from TEST.SORT\69, returns to TEST.SORT\69.
>
```

If you specify the -DBG option, Debugger-owned frames are displayed in expanded form. For example:

```
> TRACEBACK -DBG
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  6: Owner is "TEST.SWITCH".
     Called from TEST.SORT\69, returns to TEST.SORT\69.

  5: Owner is "TEST.SORT".
     Called from TEST\77, returns to TEST\78.

  4: Owner is "TEST".
     Called from debugger, returns to debugger.

  3: Debugger-owned frame.
     Called from debugger, returns to debugger.

  2: Debugger-owned frame.
     Called from debugger, returns to debugger.

  1: Debugger-owned frame.
     Called from location 4000(0)/50, returns to location 4000(3)/52.
>
```

If you specify the -ONUNITS option, the names of all on-units and their addresses are displayed for each frame. The address format is:

```
segment-number(ring-number)/address
```

For example:

```
> TRACEBACK -ONUNITS -DBG -FROM 3 -TO 1
Currently at TEST.SWITCH\47.
Stack contains 6 frames.


  3: Debugger-owned frame.
     Called from debugger, returns to debugger.
     Onunit for "STACK_OVF$" is 4000(3)/152574.
     Onunit for "NONLOCAL_GOTO$" is 4000(3)/152734.
     Onunit for "BAD_NONLOCAL_GOTO$" is 4000(3)/153010.
     Onunit for "ARITH$" is 4000(3)/153272.
     Onunit for "ILLEGAL_INST$" is 4000(3)/153072.
     Onunit for "REENTER$" is 4000(3)/153142.
     Onunit for "ANY$" is 4000(3)/152540.
     Onunit for "QUIT$" is 4000(3)/153214.
```

```
   2: Debugger-owned frame.
      Called from debugger, returns to debugger.

   1: Debugger-owned frame.
      Called from location 4000(0)/50, returns to location 4000(3)/52.
>
```

If you specify the -ADDRESSES option, the following internal address information is displayed:

- The stack and stack root segment numbers. (See the SEG and LOAD Reference Guide for explanation.)
- The stack frame address within the stack segment of each program block or frame.
- The linkbase address of the frame owner (segment number, ring number, and address within segment).
- The called-from and returns-to addresses.
- The on-unit ECB address if the -ONUNITS option is specified. (See the System Architecture Reference Guide for a discussion of ECBs.)

Here is an example of the -ADDRESSES option:

```
> TRACEBACK -ADDRESSES
Currently at TEST.SWITCH\47.
Stack contains 6 frames.

Stack segment is 4037, root segment is 4037.

  6, 146070: Owner is "TEST.SWITCH", owner LB is 4002(0)/177400.
     Called from TEST.SORT\69 (location 4001(0)/2112),
     returns to TEST.SORT\69 (location 4001(3)/2120).

  5, 145762: Owner is "TEST.SORT", owner LB is 4002(0)/177400.
     Called from TEST\77 (location 4001(0)/2163),
     returns to TEST\78 (location 4001(3)/2171).

  4, 145704: Owner is "TEST", owner LB is 4002(0)/177400.
     Called from debugger (location 2040(0)/101064),
     returns to debugger (location 2040(3)/101066).

  3, 145210: Debugger-owned frames, thru frame 1.
>
```

TRACEBACK and Recursive Blocks: The TRACEBACK command is a very useful tool to have when you are debugging recursive program blocks. The call/return stack shows you the currently active recursive calls, where each frame represents a block calling itself -- an activation of that block. Here is an example of the TRACEBACK command used with a recursive Pascal procedure named JUMP contained within its main program named FROG:

```
> TRACEBACK
Currently at FROG.JUMP\42.
Stack contains 8 frames.


  8: Owner is "FROG.JUMP", activation 4.
     Called from FROG.JUMP\59, returns to FROG.JUMP\61.

  7: Owner is "FROG.JUMP", activation 3.
     Called from FROG.JUMP\59, returns to FROG.JUMP\61.

  6: Owner is "FROG.JUMP", activation 2.
     Called from FROG.JUMP\59, returns to FROG.JUMP\61.

  5: Owner is "FROG.JUMP", activation 1.
     Called from FROG\73, returns to FROG\74.

  4: Owner is "FROG".
     Called from debugger, returns to debugger.

  3: Debugger-owned frames, thru frame 1.
>
```

In the example shown above, notice how each frame displays the activation number of each recursive call, keeping track of how many times JUMP calls itself. If you want to evaluate a variable or other expression in any particular activation of JUMP, you can look at the call/return stack, decide which activation you need, then evaluate the variable.

ERASING THE CALL/RETURN STACK WITH UNWIND

You can erase the call/return stack and cause the execution environment pointer to become undefined with the UNWIND command. This command is especially useful for eliminating multiple invocations of the Debugger resulting from the CALL command.

The format of the UNWIND command is:

```
UNWIND
```

Here is an example of the UNWIND command:

```
(8:TEST) > UNWIND
> WHERE
Execution environment is undefined.
```

(The call/return stack is discussed earlier in this chapter. The CALL command is discussed in Chapter 7.)

9

Customizing Your Debugger Commands — Macros

Commands discussed in this chapter:

MACRO ACTIONLIST
MACROLIST

INTRODUCTION

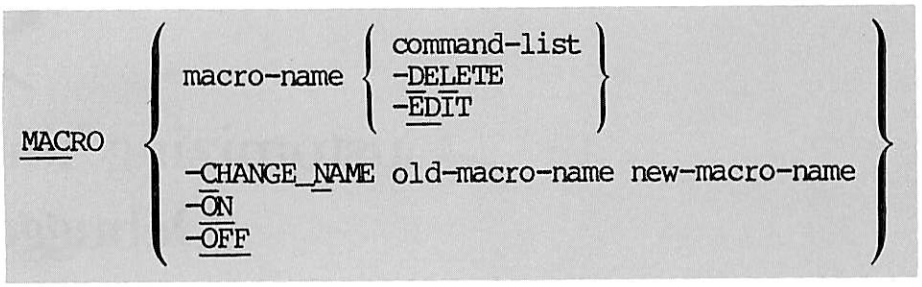
This chapter shows you how to customize your Debugger commands -- how to create a new command name that can be used in place of one or more Debugger commands. These new command names, known as macros, are created with the MACRO command.

If you do a lot of debugging, you will find that macros are wonderful time savers. Instead of entering the same sequence of Debugger commands over and over again, you can take a shortcut by creating macros to do the dirty work for you.

The Debugger's powerful features that create and manipulate macros are discussed and demonstrated in the sections that follow. (See also Chapter 12, ADVANCED MACROS.)

CREATING AND USING MACROS

All macros are created with the MACRO command. The format of the MACRO command, abbreviated MAC, is:



The macro-name is the name of the macro that you want to create. The command-list is the list of one or more Debugger commands that you want your macro name to stand for. When you create a macro, the command list must be enclosed in square brackets, and the commands must be separated by semicolons. Here is an example of how a macro is created:

```
> MACRO LOOK [WHERE; SOURCE PRINT]
```

Based on the example shown above, whenever you enter the LOOK command, the Debugger will interpret it as WHERE; SOURCE PRINT.

The -DELETE option, abbreviated -DL, deletes a specified macro. For example:

```
> MACRO LOOK -DELETE
```

The -EDIT option, abbreviated -ED, invokes the Debugger's command line editor so that you can modify the macro specified by macro-name. (See Chapter 10.)

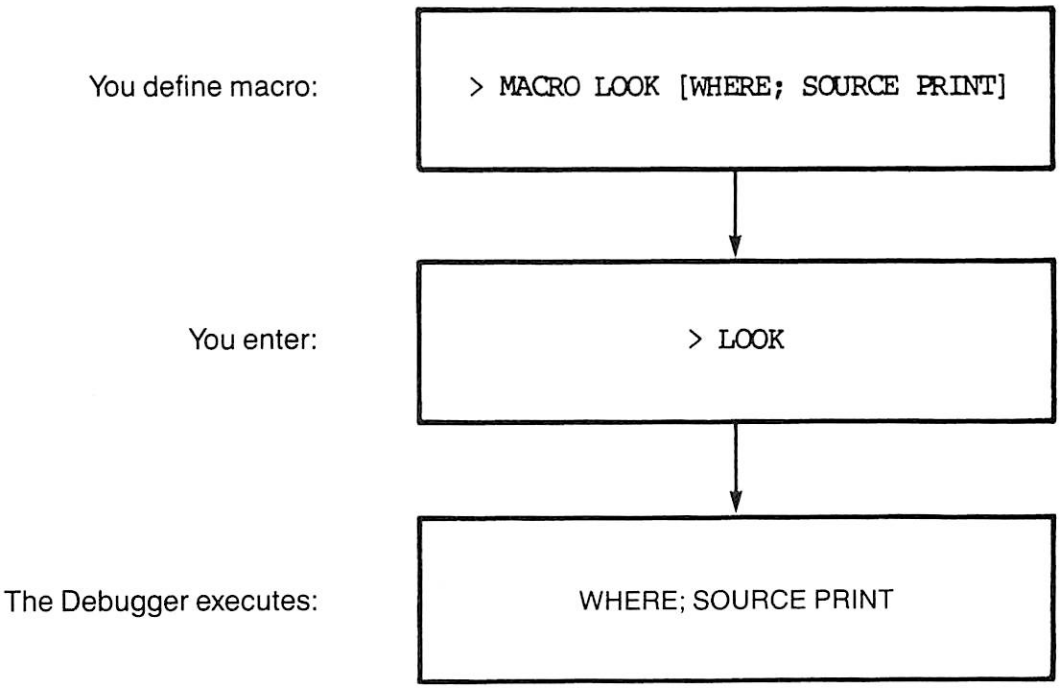
The -CHANGE_NAME option, abbreviated -CN, changes the name of a macro from old-macro-name to new-macro-name. For example:

```
> MACRO -CHANGE_NAME LOOK SEE
```

The -OFF option turns off the use of macros without destroying your current macros. The -ON option enables the use of macros once again. For example:

```
> MACRO -OFF
> MACRO -ON
```

Figure 9-1 illustrates the execution of a macro.



Execution of a Macro
Figure 9-1

Using Parameters with Macros

Your macros can accept command line parameters whenever you need different variations of a macro. To create a macro so that you can optionally use one or more parameters, enclose a positive integer within percent signs (%) in the command list for every parameter you intend to use.

For example, consider the following macro definition:

```
> MACRO RS [RESTART %1%; SOURCE PRINT]
```

In the example shown above, the 1 enclosed in percent signs means that the first parameter you use after the RS command will be plugged into that spot in the macro list command sequence. In other words, if you enter the parameter STEP as the first parameter after the RS command, then:

```
> RS STEP
```

will be interpreted by the Debugger as:

```
RESTART STEP; SOURCE PRINT
```

Similarly, if you want to use two parameters with your macro, define the position of both the first and second parameters in the macro list command sequence. For example:

```
> MACRO RS [RESTART %1%; SOURCE PRINT %2%]
```

In the example shown above, the 1 marks the position where the first parameter will be used and the 2 marks the position where the second parameter will be used. Therefore, if you use the RS macro this way:

```
> RS STEP 5
```

the Debugger will interpret it as:

```
RESTART STEP; SOURCE PRINT 5
```

where STEP is the first parameter, which replaces %1%, and 5 is the second parameter, which replaces %2%.

In the next example, the same parameter is used twice in the same command list:

```
> MACRO AR [: AR1[%1%]; : AR2[%1%]]
```

Given the macro command list shown above, if you enter:

```
> AR 5
```

the Debugger will interpret it as:

```
: AR1[5]; : AR2[5]
```

Note

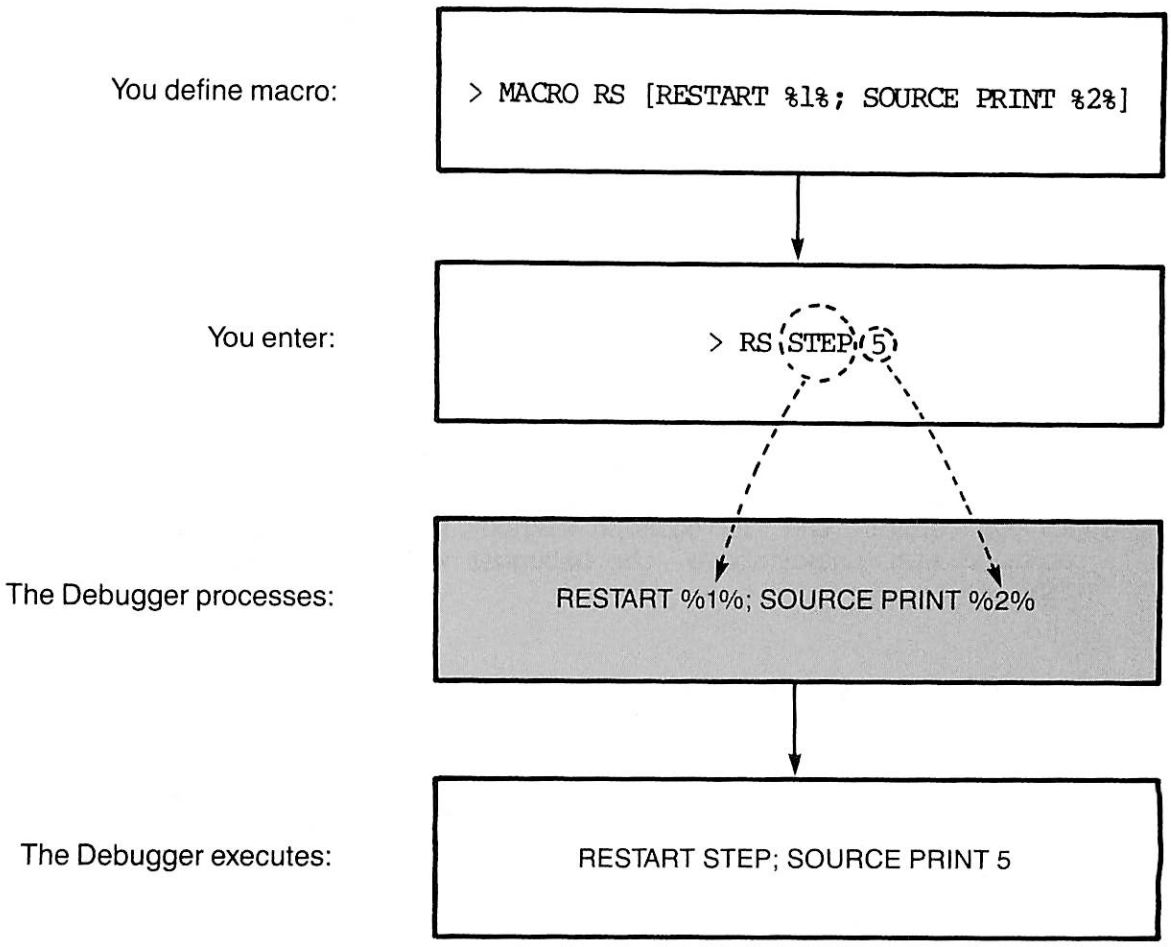
Use of macro command line parameters is optional. Even if your command list expects parameters, you do not have to use them. If you enter the RS command without parameters, based on its command list shown above, the Debugger will simply interpret it as:

```
RESTART; SOURCE PRINT
```

If you enter a parameter when your macro expects no parameters, or if you enter a parameter beyond the highest numbered parameter in the macro list, then that parameter is appended to the command list. For example, suppose your macro RS expects no parameters. If you enter RS 5, the Debugger will interpret it as:

```
RESTART; SOURCE PRINT 5
```

Figure 9-2 illustrates the execution of a macro that contains parameters.



Execution of a Macro Containing Parameters
Figure 9-2

Using Parameters that Contain Spaces: Often you might want to use a parameter that contains one or more spaces. For example, you might want to enter STEP 4 as one parameter. To use a parameter that contains one or more spaces, enclose the parameter in single quotes on the macro command line. For example:

```
> RS 'STEP 4'
```

This macro command will be interpreted by the Debugger as:

```
RESTART STEP 4; SOURCE PRINT
```

Using Literal Single Quotes in Parameters: If you want to use one or more literal single quotes in a parameter, as they are used in character strings, each quote in the parameter must be doubled, then the entire parameter must be enclosed in quotes. For example, suppose you defined this macro named V:

```
> MACRO V [: 'VALUE = ' || %1%]
```

(The || operator is the PL/I-G character string concatenation operator.)

Based on the macro definition shown above, if you enter the following macro command:

```
> V '''string'''
```

the Debugger interprets it as:

```
: 'VALUE = ' || 'STRING'
```

Similarly, if you enter this macro command:

```
> V '''string of A''''S'''
```

the Debugger interprets it as:

```
: 'VALUE = ' || 'STRING OF A''S'
```

Similarly, if you enter:

```
> V 'STRING'
```

the Debugger interprets it as:

```
: 'VALUE = ' || STRING
```

Using Literal Percent Signs: If you want to use a literal percent sign in a parameter, so that is not interpreted as a special symbol, enter two consecutive percent signs.

Some Macro Examples

The following are examples of macros that can simplify your debugging sessions. (See also Chapter 12, ADVANCED MACROS.)

- To display five lines of source code, enter:

```
> MACRO P5 [SOURCE PRINT 5]
```

- To display the next line of source code, enter:

```
> MACRO N [SOURCE NEXT]
```

- To step into a called program block and display the first executable statement, enter:

```
> MACRO SP [STEPIN; SOURCE PRINT]
```

- To use the source PRINT subcommand without using SOURCE, enter:

```
> MACRO P [SOURCE PRINT %1%]
```

- To display a particular line of code in a particular evaluation environment, enter:

```
> MACRO PT [ENVIRONMENT %1%; SOURCE POINT %2%]
```

Note

You can use a macro name to define another macro. For example, consider these three macro definitions:

```
> MACRO LOOK [WHERE; SOURCE PRINT]
> MACRO SEE [STEP; SOURCE PRINT]
> MACRO GAWK [LOOK; SEE]
```

DISPLAYING ALL YOUR MACROS

You can display one or all of your currently defined macros and their command lists with the MACROLIST command.

The format of the MACROLIST command, abbreviated ML, is:

```
MACROLIST [macro-name]
```

The macro-name is the name of a specific macro that you want to display.

When macros are created, they are placed into an internal Debugger table, known as the macro list. If you enter the MACROLIST command by itself, with no macro name, all macros in the macro list and their corresponding command lists are displayed. For example:

```
> MACROLIST
BALW        [BREAKPOINT %1% [%2%; WHERE]]
BCC         [BREAKPOINT %1%; CLEAR; CONTINUE]
DEC         [:FORTRAN, DECIMAL :%1%]
RS          [RESTART %1%; SOURCE PRINT]
STS         [STEP; SOURCE EX]
```

If you want to display the command list for only one macro, enter the macro name following the MACROLIST command:

```
> MACROLIST BCC
BCC         [BREAKPOINT %1%; CLEAR; CONTINUE]
```

Caution

Once you leave the Debugger and return to PRIMOS command level, the macros in your macro list are destroyed, just as all breakpoints and tracepoints are destroyed. If you want to save your macros as well as breakpoints and tracepoints for future debugging sessions, see the discussion on the SAVESTATE and LOADSTATE commands in Chapter 10.

Displaying Macros with ACTIONLIST Command

The ACTIONLIST command, which was introduced in Chapter 5, can display the contents of a macro command list prior to its execution. The format of the ACTIONLIST command, abbreviated AL, is:

```
ACTIONLIST { SUPPRESS }
           { PRINT    }
```

Normally, no lists are displayed prior to their execution. The PRINT option specifies all lists to be displayed. The SUPPRESS option deactivates the PRINT option, causing no lists to be displayed.

For example, to cause a macro command list to be displayed prior to execution, enter:

```
> ACTIONLIST PRINT
> MACRO E [: TS.ARR1(%1%).VALUE - TS.ARR2(%2%, %1%).VALUE]
> E 3 2

<1> : TS.ARR1(3).VALUE - TS.ARR2(2, 3).VALUE
2
```

The ACTIONLIST command also can display the contents of breakpoint action lists. (For more information on the ACTIONLIST command, see Chapter 5.)

10

Modifying and Saving Debugger Commands

Commands discussed in this chapter:

RESUBMIT	SAVESTATE
BREAKPOINT -EDIT	LOADSTATE
MACRO -EDIT	

INTRODUCTION

The Debugger's command line editor allows you to edit the most recent command line entered and edit breakpoint action lists and macro command lists. This editor is very useful when you have to fix mistakes in or change command lines. It saves you the time of reentering the entire command to fix a mistake or to make a desired change.

Besides being able to modify and edit your command lines, you can also save all of your breakpoints, tracepoints, and macros in PRIMOS files. Saving your breakpoints, tracepoints, and macros means that you can pull them out of a file whenever you need them in future debugging sessions. If you have a couple of favorite breakpoints or macros, or if you use so many of them that they're hard to keep track of, this Debugger feature is especially useful.

Specifically, this chapter discusses:

- Using the Debugger's command line editor.
- Modifying and editing the most recent command line entered with the RESUBMIT command and the command line editor.
- Modifying and editing breakpoint action lists and macro command lists with the command line editor.
- Saving your breakpoints, tracepoints, and macros by putting them in a PRIMOS file with the SAVESTATE command.

- Pulling your breakpoints, tracepoints, and macros out of the PRIMOS file to use during future debugging sessions with the LOADSTATE command.

USING THE COMMAND LINE EDITOR

The Debugger's command line editor has eight editing subcommands that perform the following functions:

- Delete a character using the D subcommand.
- Define the first character of the command line using the F subcommand.
- Define the last character of the command line using the L subcommand.
- Append text to the end of the command line using the A subcommand.
- Insert text using the I subcommand.
- Overlay text using the O subcommand.
- Replace the original command line and return to Debugger command level by entering a carriage return.
- Quit editing without replacing original command line and return to Debugger command level using the Q subcommand.

When you enter the RESUBMIT, the BREAKPOINT -EDIT, or the MACRO -EDIT command, the most recent command line or the specified breakpoint or macro displays on your terminal. A colon (:) then appears on the next line at the left margin. This colon is the command line editor prompt, which waits for subcommand input.

The editor subcommands are described and demonstrated in the paragraphs that follow.

Deleting a Character (D): The character under which the D is positioned is deleted. The line is shifted to the left to fill in the vacant character position. Further edit operations are allowed following a D. Here is an example of the D subcommand:

```
   Now is the time for all good good women...
:                            DDDDD     DD
   Now is the time for all good men...
:
```

Defining the First Character (F): The character under which the F is positioned becomes the first character of the line. The portion of the line appearing to the left of the F is truncated. Further edit operations are allowed following the F. Here is an example of the F subcommand:

```
   These are the times which try men's souls...
:                           F
   try men's souls...
:
```

Defining the Last Character (L): The character under which the L is positioned becomes the last character of the line. The portion of the line to the right of L is truncated. Further edit operations are allowed following the L. Here is an example of the L subcommand:

```
   If anything can go wrong, it will.
:                                  L
   If anything can go wrong, it wil
```

Append Text to End of Line (A): The A subcommand causes the text that follows the A to be appended to the end of the line. All text that follows the A is interpreted literally. Here is an example of the A subcommand:

```
   Four scor
: Ae and seven years
   Four score and seven years
:
```

Insert Text (I): The I subcommand causes the text that follows it to be inserted into the line following the character under which the I is positioned. No further edit subcommands are allowed on the command line after the I command is entered. Here is an example of the I subcommand:

```
   Let me make one thing clear.
:                       Iperfectly
   Let me make one thing perfectly clear.
:
```

Overlay Text (O): The O subcommand causes the characters that follow it to be overlayed onto the line, starting at the character under which the O appears. No further edit commands are allowed on the command line. Here is an example of the O subcommand:

```
   My kingdom for a horse!
:     DDDOlife
   My life for a horse!
```

Finish Editing (carriage return): When you enter a carriage return, it indicates that editing is finished. The edited command line replaces the original command line.

Quit Editing (Q): The Q subcommand aborts the editing session and returns you to Debugger command level without replacing the original command line. Here is an example of the Q subcommand:

```
: Q
>
```

All of these command line editor subcommands are summarized in Table 10-1.

MODIFYING THE MOST RECENT COMMAND

Using the Debugger's command line editor and the RESUBMIT command, you can modify the most recent command you entered and resubmit that command for execution. Modifying the most recent command is useful for correcting mistakes.

The format of the RESUBMIT command, abbreviated RSU, is:

```
RESUBMIT
```

Assume that you have just entered the following Debugger command line:

```
> CLEAR $MAIN\6; BREAKPOINT $MAIN\20; ETRACE ON; CONTINUE

CLEAR $MAIN\6
No such breakpoint.
```

Table 10-1
Command Line Editor Subcommands

Subcommand	Function
D	Deletes the character under which the D is positioned.
F	Makes the character under which the F is positioned the first character of the command line.
L	Makes the character under which the L is positioned the last character of the command line.
A	Appends the text that follows A to the end of the line.
I	Inserts the text that follows I into the line following the character under which the I is positioned.
O	Overlays the characters that follow O onto the line starting at the character under which the O appears.
carriage return	Finishes the editing session and replaces original command line.
Q	Aborts the editing session and returns to Debugger command level, but does not replace original command line.

Seeing this error message, you realize the breakpoint is at line 7, not line 6. You also realize that entry tracing should be turned off, not on. You then enter the following:

```
> RESUBMIT
  CLEAR $MAIN\6; BREAKPOINT $MAIN\20; ETRACE ON; CONTINUE
:           07
  CLEAR $MAIN\7; BREAKPOINT $MAIN\20; ETRACE ON; CONTINUE
:                                              DDDDIOFF;
  CLEAR $MAIN\7; BREAKPOINT $MAIN\20; ETRACE OFF; CONTINUE
: (carriage return)
>
```

MODIFYING BREAKPOINTS AND MACROS

If you use breakpoint action lists and macros regularly, you will need to change the contents of, or fix mistakes in, action lists and macro command lists. The command line editor, combined with the -EDIT option of the BREAKPOINT or MACRO command, is also used to modify breakpoints and macros.

When you want to modify or edit an action list or macro, use the -EDIT option of the BREAKPOINT or MACRO command respectively. For example, enter:

```
> BREAKPOINT breakpoint-identifier -EDIT
```

or enter:

```
> MACRO macro-name -EDIT
```

The breakpoint-identifier, usually a source line number or statement label, identifies the breakpoint as defined in Chapters 4 and 5.

The macro-name is the name of the macro you want to modify. (See also Chapter 9.)

When you enter either of these commands, the specified breakpoint action list or macro command list displays on your terminal. A colon (:) appears on the next line at the left margin. This colon is the command line editor prompt, which waits for subcommand input. Here is an example of the -EDIT option:

```
> BREAKPOINT 30 -EDIT
  LET N = 'C'; : B; : UCOUNT; WHERE; CONTINUE
:
```

In the example shown above, the command line editor waits for you to use its subcommands to edit the action list. (The editor subcommands are described and demonstrated earlier in this chapter.)

Example Using Four Subcommands: The following example illustrates how four command line editor subcommands are used to edit a breakpoint action list. In this editing session, the LET command is replaced by the WHERE command:

```
> BREAKPOINT 30 -EDIT
   LET N = 'C'; : B; : UCOUNT; WHERE; CONTINUE
:                F
   : B; : UCOUNT; WHERE; CONTINUE
: IWHERE;
   WHERE; : B; : UCOUNT; WHERE; CONTINUE
:                        DDDDDDD
   WHERE; : B; : UCOUNT; CONTINUE
: (carriage return)
> LIST 30
Type  Location
brk   A\30, count = 0
      [WHERE; : B; : UCOUNT; CONTINUE]
```

SAVING YOUR BREAKPOINTS, TRACEPOINTS, AND MACROS

If you create some useful breakpoint action lists, tracepoints, or macros during your debugging sessions, you will want to save them to use over again the next time you enter the Debugger. The SAVESTATE command allows you to save your favorite breakpoints, tracepoints, and macros in PRIMOS files that reside in your user file directory. Later on, when you use the Debugger again, you can simply pull your breakpoints, tracepoints, or macros out of these files and into the debugging session. (See the LOADSTATE command later in this chapter.)

The SAVESTATE Command

The SAVESTATE command saves your breakpoints, tracepoints, and macros and places them into a PRIMOS file in your directory for future use. These files are also known as SAVESTATE files. The format of the SAVESTATE command, abbreviated SS, is:

```
SAVESTATE filename [-MACROS] [-BREAKPOINTS] [-TRACEPOINTS]
```

The filename is the pathname of the PRIMOS file where you want your breakpoints, tracepoints, and macros placed. If you do not specify a pathname, the file is placed into the directory to which you are attached.

If you specify just a filename without an option, then, by default, all of your breakpoints, breakpoint action lists, tracepoints, and macros are placed into the file specified by filename. For example:

```
> SAVESTATE DBGSTUFF
```

In the example shown above, all of the breakpoints, tracepoints, and macros that you are using at the time this command is given are placed into a PRIMOS file named DBGSTUFF in the directory to which you are attached.

The -MACROS option, abbreviated -MAC, causes only your macros to be placed into the file specified by filename. For example:

```
> SAVESTATE MYMACROS -MACROS
```

The -BREAKPOINTS option, abbreviated -BRK, causes only your breakpoints and their action lists to be placed into the file. For example:

```
> SAVESTATE MYBREAKS -BREAKPOINTS
```

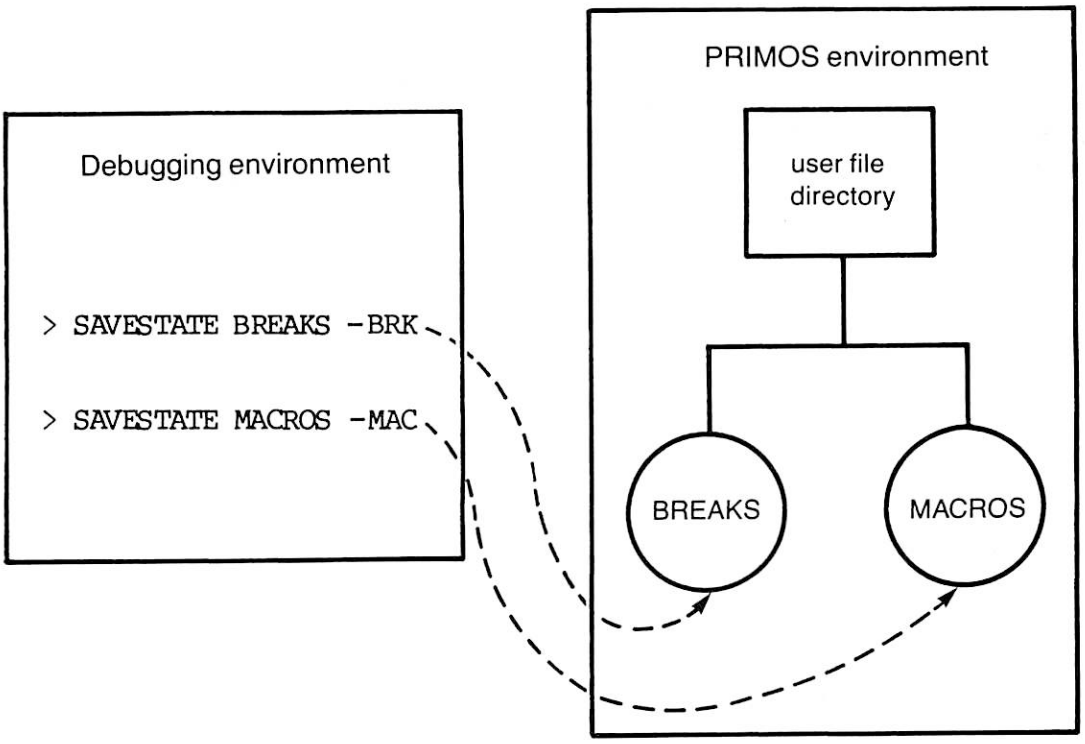
The -TRACEPOINTS option, abbreviated -TRA, causes only your tracepoints to be placed into the file. For example:

```
> SAVESTATE MYTRACES -TRACEPOINTS
```

Caution

While you are debugging, if you want to add new breakpoints, tracepoints, and macros to an existing file, first use the LOADSTATE command to pull the contents of the existing file into your debugging session. Then use SAVESTATE to add the new information to that file. Do not do it the other way around. If you save new information in a file that already exists before using LOADSTATE, the contents of that file will be destroyed and overwritten. (The LOADSTATE command is discussed later in this chapter.)

Figure 10-1 illustrates the use of the SAVESTATE command.



The SAVESTATE Command
Figure 10-1

Manipulating SAVESTATE Files from PRIMOS

Your SAVESTATE files are PRIMOS files, similar to the other PRIMOS files that reside in your directory. Therefore, you can edit, look at, and create these files from PRIMOS command level using Prime's line EDITOR or EMACS screen editor.

Note

If you create your own SAVESTATE file using an editor, you should add the word END-SAVE at the bottom of the file. The END-SAVE must be the last line in your file so that the LOADSTATE process terminates properly. When you create the file with the SAVESTATE command from the Debugger, the END-SAVE is automatically put at the bottom of the file for you.

RESTORING SAVED BREAKPOINTS, TRACEPOINTS, AND MACROS

If you have saved your breakpoints, tracepoints, and macros in a SAVESTATE file during a previous debugging session, you can use them again in later debugging sessions by restoring them with the LOADSTATE command. LOADSTATE pulls the contents of your SAVESTATE file back into your debugging session. (Saving breakpoints, tracepoints, and macros with SAVESTATE is discussed in the previous section.)

The format of the LOADSTATE command, abbreviated LS, is:

```
LOADSTATE filename
```

The filename is the pathname of the SAVESTATE file that contains the breakpoints, tracepoints, and macros you want to use. For example, if you wanted to use macros that you had previously saved in a file named MYMACROS, you would enter:

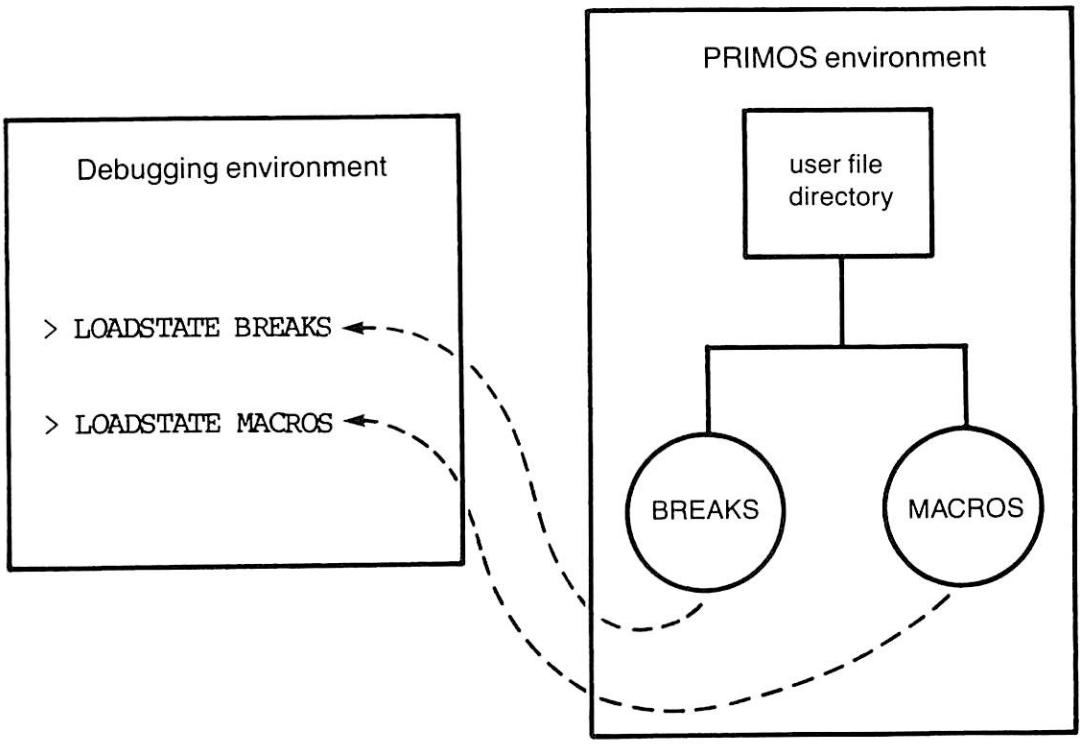
```
> LOADSTATE MYMACROS
```

The commands in the file execute without terminal output unless an error occurs.

Note

If any of your breakpoints, tracepoints, or macros in the SAVESTATE file contains a mistake, such as a misspelled argument, or if your END-SAVE is missing from the bottom of the file, you will receive an error message.

Figure 10-2 illustrates the use of the LOADSTATE command.



The LOADSTATE Command
Figure 10-2

You can restore the contents of any SAVESTATE file when invoking the Debugger using the DBG command's -LOADSTATE option, abbreviated -LS. For example:

```
OK, DBG TEST -LOADSTATE MYMACROS
```

The -LOADSTATE option is further discussed in Chapter 13.

11
Other Features

Commands discussed in this chapter:	
SOURCE EX	PAUSE
SOURCE NAME	*
SOURCE RENAME	AGAIN
!	

INTRODUCTION

This chapter describes some miscellaneous Debugger features that supplement the other features you learned about in Chapters 3 through 10. Specifically, this chapter covers:

- Using multiple commands on the SOURCE command line.
- Using three special SOURCE subcommands (EX, NAME, and RENAME).
- Executing PRIMOS commands from within the Debugger with the ! command.
- Suspending your Debugging session with the PAUSE command.
- Repeating Debugger commands with the * and AGAIN commands.

USING MULTIPLE COMMANDS WITH SOURCE

When you use the SOURCE command, all commands that follow it on the command line are interpreted to be source EDITOR subcommands. The commands are separated by semicolons, which are Debugger command separators:

```
> SOURCE TOP; PRINT 23
```

When you want to use other Debugger commands on the same command line that contains the SOURCE command, enter two consecutive separator characters:

```
> SOURCE TOP; PRINT 5;; RESTART
```

(Chapter 3 discusses the general use of the SOURCE command.)

USING SOURCE SUBCOMMANDS EX, NAME, AND RENAME

Three additional source subcommands that provide special functions in the Debugger environment are also available:

- EX
- NAME
- RENAME

The Source EX Subcommand

The source EX subcommand sets the source file and EDITOR line pointer to correspond to the source line where execution is to resume (the execution environment pointer) then displays that line. This is illegal when the execution environment pointer describes the exit from a program block.

Suppose execution were suspended with a breakpoint on line 7. The execution environment pointer would be at line 7. Here is an example of how the EX subcommand moves the EDITOR line pointer to source line 7 and displays that line in a PL/I-G program:

```
> BREAKPOINT 7
> RESTART

**** breakpointed at TEST\7
> SOURCE PRINT 5
    7:      PUT SKIP LIST('The integer sum is', Z);
    8:      PUT SKIP;
    9:      A = 5.5;
   10:      B = 4.3;
   11:      C = A + B;
> SOURCE EX
    7:      PUT SKIP LIST('The integer sum is', Z);
```

The Source NAME Subcommand

The source NAME subcommand allows you to look at the contents of another file from within the Debugger. The format of the NAME subcommand is:

```
SOURCE NAME [ filename   ]
            [ -DEFAULT   ]
```

The filename is the name of the other file you want to look at. The -DEFAULT option, abbreviated -DF, brings you back to looking at the file corresponding to the evaluation environment.

Used by itself, with no argument, the NAME subcommand simply gives you the current source pathname. For example:

```
> SOURCE NAME
Source file is "<PRIME>PAUL>TEST.PL1G", based on evaluation
environment.
```

In the example shown above, the NAME subcommand tells you that the current source file is called TEST.PL1G, which is located in the directory PAUL, within the disk volume PRIME.

Suppose you wanted to look at a second file called MULT.PL1G while you were debugging TEST.PL1G. You could specify MULT with the NAME subcommand:

```
> SOURCE PRINT 10
    1: TEST : PROCEDURE;
    2:   DECLARE (X, Y, Z) FIXED BIN(15);
    3:   DECLARE (A, B, C) FIXED DEC(4, 2);
    4:     X = 5;
    5:     Y = 3;
    6:     Z = X + Y;
    7:     PUT SKIP LIST('The integer sum is', Z);
    8:     PUT SKIP;
    9:     A = 5.5;
   10:     B = 4.3;
> SOURCE NAME
Source file is "<PRIME>PAUL>TEST.PL1G", based on evaluation
environment.
> SOURCE NAME MULT.PL1G
> SOURCE NAME
Source file is "MULT.PL1G", user specified.
> SOURCE PRINT 23
.NULL.
    1: MULT : PROCEDURE;
    2:   DECLARE (P, Q, R) FIXED BIN(15);
    3:   P = 5;
    4:   Q = 10;
    5:   R = P * Q;
    6:   PUT SKIP LIST('The product is', R);
    7:   PUT SKIP;
    8: END MULT;
BOTTOM
>
```

In the example shown above, the current source filename is temporarily changed from TEST.PL1G to MULT.PL1G. If you want to go back to looking at TEST.PL1G, issue the NAME -DEFAULT subcommand:

```
> SOURCE NAME MULT.PL1G
> SOURCE PRINT 23
.NULL.
    1: MULT : PROCEDURE;
    2:   DECLARE (P, Q, R) FIXED BIN(15);
    3:   P = 5;
    4:   Q = 10;
    5:   R = P * Q;
    6:   PUT SKIP LIST('The product is', R);
    7:   PUT SKIP;
    8: END MULT;
BOTTOM
> SOURCE NAME -DEFAULT
> SOURCE PRINT 10
.NULL.
    1: TEST : PROCEDURE;
    2:   DECLARE (X, Y, Z) FIXED BIN(15);
    3:   DECLARE (A, B, C) FIXED DEC(4, 2);
    4:     X = 5;
    5:     Y = 3;
    6:     Z = X + Y;
    7:     PUT SKIP LIST('The integer sum is', Z);
    8:     PUT SKIP;
    9:     A = 5.5;
>
```

Note

Use of the source NAME subcommand to look at a second file does not interfere with the debugging of the first (default) file.

The Debugger selects the file to be operated upon by SOURCE commands using the following rules:

1. When the Debugger is entered from PRIMOS command level, the source file is that which contains the main program. The line pointer is positioned to the first line in the program.

2. When the Debugger is reentered from the user program as a result of a breakpoint or condition, the source file name is set to that containing the program block represented by the execution environment pointer. The line number is that of the breakpointed statement (if a statement breakpoint), first line in the program block (if an entry to a program block), or next statement to be executed (if exiting from a program block). It is set to the statement that caused the condition to be raised if a condition caused Debugger reentry.

3. When you execute an ENVIRONMENT command, the source file is set to that which contains the program block described by the new evaluation environment pointer. The line number is set to the first line in the program block.

4. When the SOURCE EX command is executed, the source file name is set to correspond to the execution environment pointer. The line number is set as described in rule 2, above.

The SOURCE RENAME Subcommand

The source RENAME subcommand resets the default source filename for a specified program block. In other words, the default source file -- the filename defined in the symbol table -- is reset for the rest of the debugging session until another SOURCE RENAME command is entered. The format of the RENAME subcommand is:

```
SOURCE RENAME filename [-BLOCK program-block-name]
```

The filename is the name of the file that you want as your default source file. The program-block-name is the name of the program block in which the default source file will be the specified filename. If you do not specify program-block-name, the program block corresponding to the current evaluation environment is assumed. If the indicated program block is the same as the current block, the current source file is changed to filename.

EXECUTING PRIMOS COMMANDS FROM DEBUGGER

You can execute certain PRIMOS commands from Debugger command level using the ! command, which is an exclamation point on your keyboard. This command is useful if you want to perform PRIMOS operations, such as attaching to another directory, listing your directory's files, and checking the time, without leaving the Debugger.

The format of the ! command is:

```
! primos-command-line
```

The primos-command-line is a PRIMOS command that you want to execute from the Debugger.

> **Caution**
>
> In order for this command to work properly you must use internal PRIMOS commands, not external commands. (Internal and external commands are summarized in the PRIMOS Commands Reference Guide.) Internal commands are part of PRIMOS itself. External commands are programs that are stored in a special directory on your system. External commands execute, but they interfere with the memory image of the Debugger or your program. Therefore, you could not return to Debugger command level.

Here is an example of the ! command:

```
> ! ATTACH PAUL>MEMOS
>
```

In the example shown above, the PRIMOS command ATTACH is executed and control returns to Debugger command level.

SUSPENDING YOUR DEBUGGER SESSIONS

You can temporarily suspend your Debugging session and return to PRIMOS command level with the PAUSE command. PAUSE is useful when you have several PRIMOS commands to enter instead of just one or two. The format of the PAUSE command, abbreviated PA, is:

```
PAUSE
```

> **Caution**
>
> As explained under the Caution in the previous discussion of the ! command, you must use internal PRIMOS commands with PAUSE, not external commands. (Internal and external PRIMOS commands are summarized in the PRIMOS Commands Reference Guide. See also the Caution in the previous section.)

Here is an example of the PAUSE command:

```
> PAUSE
To resume debugging, type 'START'.

OK, ATTACH PAUL>MEMOS
OK, START
>
```

As shown in the example above, whenever you are ready to return to Debugger command level, type START. The OK, prompt is the PRIMOS command prompt.

REPEATING DEBUGGER COMMANDS

Two Debugger commands (* and AGAIN) allow you to repeat the execution of Debugger commands. The * command, which is an asterisk on your keyboard, repeats the current command line as many times as you wish. The AGAIN command repeats the Debugger command most recently executed. These two commands are discussed in the paragraphs that follow.

The * Command

The * command, which is an asterisk, executes the current command line for a specific number of times or forever. The format of the * command is:

```
* [value]
```

The optional value is the number of times you want the command line to be repeated.

The * command must be the last command on your command line and separated from the preceding commands by a semicolon (command separator).

If you do not supply a value following * on the command line, then that command line will be executed forever or until an error is encountered or until you hit the break (CONTROL-P) key. For example, to locate all occurrences of the character & in the current source file, enter:

```
> SOURCE TOP
> SOURCE LOCATE &; *
BOTTOM
>
```

If you want to repeat the command line a specific number of times, specify a value immediately following the * command. For example, to print the values of A(I) squared, incrementing I from 1 to 10, enter:

```
> LET I = 0
> LET I = I + 1; : A(I) ** 2.; *10
```

In the example shown above, note that ** is the operator that squares A(I).

The AGAIN Command

The AGAIN command repeats the Debugger command line that has just been executed. This command is useful if you have to execute the same command more than once but don't want to type the command over again. The format of the AGAIN command, abbreviated A, is:

```
AGAIN
```

Here is an example of how AGAIN might be used:

```
> SOURCE LOCATE YEAR
   4:      YEAR  = INTEGER;
> AGAIN
   7: FUNCTION CALC (MONTH_IN : MONTH; YEAR_IN : YEAR) : DATE;
> AGAIN
  16:             IF YEAR_IN MOD 4 = 0 THEN
>
```

Note

You must enter the AGAIN command by itself on the command line.

PART III

Advanced Techniques and Features

12
Advanced Macros

Commands discussed in this chapter:
MACRO MACROLIST

INTRODUCTION

This chapter lists some more complex examples of how macros can be used in your debugging sessions. Each example is accompanied by an explanation of what function the macro is performing. In future revisions of this book, more macro examples may be added to this chapter. (See also Chapter 9 for information on how to create and use macros.)

As you recall from Chapter 9, all macros are created with the MACRO command, abbreviated MAC. The contents of macro lists can be displayed with the MACROLIST command. (The formats of these two commands are given in Chapter 9.)

Some examples of more complex macros follow:

- To examine 10 source lines at a time without disturbing the source line pointer:

```
> MACRO PT [SOURCE BRIEF; NEXT -5; PRINT 10; NEXT -4; VERIFY]
```

- To display a particular name and home state of a U.S. President contained in a PL/I-G linked list structure:

```
> MACRO PREZ [: ADR(%1%)->PRES(%2%).NAME; : STATE(ADR(%1%)->^
PRES(%2%).SN)]
```

- To save all your macros for future use in a PRIMOS file, MYMACROS, and leave the Debugger:

```
> MACRO SSM [SAVESTATE PAUL>DBG>MYMACROS -MACROS; QUIT]
```

- To set a new breakpoint, clear the breakpoint you are suspended on, and continue execution to your new breakpoint, enter:

```
> MACRO PR [BREAKPOINT %1%; CLEAR; CONTINUE]
```

- To close a file in any particular directory, enter:

```
> MACRO CLS [! CLOSE <Prime>%1%>%2%>%3%]
```

- To check the date and the status of the network, enter:

```
> MACRO DNE [! DATE; ! STAT NETWORK]
```

- To be able to look at any RPG source file in any particular directory, enter:

```
> MACRO NAM [SOURCE NAME <Prime>%1%>%2%.rpg]
```

13
Other Advanced Features

Commands discussed in this chapter:

DBG	STATUS
CMDLINE	PMODE
INFO	VPSD
SEGMENTS	

INTRODUCTION

This chapter describes some Debugger features that, in general, you might use in more advanced applications. Specifically, this chapter covers:

- Using DBG command line options when you invoke the Debugger with the DBG command. These options tell the Debugger to do or not do certain things during its operations.
- Using the compiler options -DEBUG, -PRODUCTION and -NODEBUG.
- Entering command line arguments with the CMDLINE command.
- Using the Debugger's advanced information request commands (INFO, SEGMENTS, and STATUS).
- Setting the print mode with the PMODE command.
- Entering the 64V Mode Prime Symbolic Debugger (VPSD) with the VPSD command.

USING DBG COMMAND LINE OPTIONS

There are several DBG command line options available to you that make the Debugger do some useful things during its operations. (The DBG command, which invokes the Debugger from PRIMOS command level, is discussed in Chapter 3.)

The command line option follows the name of the program file on the DBG command line. For example:

```
OK, DBG MYPROG -COMINPUT
```

Table 13-1 lists the DBG command line options and their functions.

Table 13-1

DBG Command Line Options
(Abbreviations are underlined.)

Option	Function
-LOADSTATE pathname	-LOADSTATE allows you to restore the contents of a SAVESTATE file -- your saved breakpoints, tracepoints, and macros -- upon invoking the Debugger. The pathname is the pathname of the file that you want to restore.
-VERIFY_SYMBOLS and -NO_VERIFY_SYMBOLS	-VERIFY_SYMBOLS checks all external symbol declarations for consistency in all program blocks contained within the executable file. The Debugger displays a warning message during initialization if it encounters external symbol declarations that differ. -NO_VERIFY_SYMBOLS, which is the opposite of -VERIFY_SYMBOLS, suppresses external symbol checking, thereby speeding up initialization. (-VERIFY_SYMBOLS is the default.)

Table 13-1 (continued)
DBG Command Line Options

-VERIFY_PROC and -NO_VERIFY_PROC	-VERIFY_PROC specifies that the procedure text is to be verified to ensure that statement breakpoints may be set where applicable. A warning message is displayed if the Debugger encounters a statement for which the procedure text is unsuitable for placing a breakpoint. -NO_VERIFY_PROC, the opposite of _VERIFY_PROC, specifies that the procedure text is not to be inspected for improper format regarding placement of breakpoints. (-VERIFY_PROC is the default.)
-COMINPUT and -NO_COMINPUT	-COMINPUT specifies that the Debugger accept input from a command input file or CPL program. -NO_COMINPUT, specifies that the Debugger accept input only from the terminal, not from a command input file. (-NO_COMINPUT is the default.)
-FULL_INIT and -QUICK_INIT	-FULL_INIT causes the Debugger to read and process the entire symbol table from the specified executable file prior to entering command mode. Normally, information is read from the symbol table only when required. This option is useful for obtaining a complete external symbol mismatch summary at initialization time. Use of -FULL_INIT will approximately triple initialization time. -QUICK_INIT specifies that only information required to identify each block is to be loaded at initialization time. The remainder of the symbol table is loaded as required during the debugging session. (-QUICK_INIT is the default.)

USING COMPILER OPTIONS

Normally, when you want to use the Source Level Debugger to debug your program, you specify the -DEBUG option during compilation. There are two other compiler options that are related to the Debugger (-PRODUCTION and -NODEBUG).

The -PRODUCTION option, which can be used instead of the -DEBUG option, also produces code for the Debugger. However, its code is limited and less powerful than the code generated by -DEBUG. The -NODEBUG option, which is one of your compiler's default options, causes no Debugger information to be generated.

When you use the -DEBUG option during compilation, the compiler is said to be in debug mode. Similarly, when the -PRODUCTION or -NODEBUG option is used, the compiler is said to be in production mode or nodebug mode respectively.

The following paragraphs provide more information on the use of debug, production, and nodebug modes.

Debug Mode -- Produce Full Debugger Information

The debug compilation mode causes the compiler to produce symbol and statement information for the Debugger.

You may set breakpoints at any statement in, entry to or exit from a program compiled with the -DEBUG option. You may also reference all symbols declared within the program.

Programs compiled in debug mode are not optimized. The amount of space occupied by the procedure text of debug mode programs is 10-20 percent larger than that occupied by production or nodebug mode programs. Furthermore, the amount of space occupied by the link frame for each debug mode program increases two halfwords for each common block (external static variable) and external entry declared.

Production Mode -- Produce Limited Debugger Information

Production mode is the hybrid of debug and nodebug modes. Information about each program block and symbol is produced by the compiler, but no statement information is produced.

You may place breakpoints at the entry to and exit from programs, but may not breakpoint at individual statements. Program locations are identified by program block name plus numerical offset expressed in octal. (No statement information is produced by the compilers.)

Code produced during a production mode compilation is optimized. Unlike debug mode, no extra space is used by the procedure text. However, the amount of space used in the linkage frame increases two halfwords for each common block (external static variable) and external entry declared.

Nodebug Mode -- Produce No Debugger Information

The nodebug compilation mode causes the compiler to output no symbol and statement information for later digestion by the Debugger. Programs compiled in this mode will not be recognized by the Debugger, in that no statement, entry, or exit breakpoints may be set, nor may any variables be referenced. The only way the Debugger can identify a program compiled in nodebug mode (in a traceback listing, for example) is using segment and halfword memory addresses. Programs compiled in nodebug mode may be optimized by the compiler.

ENTERING COMMAND LINE ARGUMENTS WITH CMDLINE

The CMDLINE command may be used to enter your program's command line arguments.

The format of the CMDLINE command, abbreviated CL, is:

```
CMDLINE
```

The command line must be reentered each time the program is restarted.

You should be careful not to enter the PRIMOS command SEG, DBG, or RESUME following the CMDLINE command. The first argument entered will be the first read by the user program.

Here is an example of the CMDLINE command:

```
> CMDLINE
Enter command line:
-FREQ 12 -BRIEF
```

Here is another example that includes the invocation of the Debugger:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> CMDLINE
Enter command line:
MYFILE
```

USING ADVANCED INFORMATION REQUEST COMMANDS

In addition to the WHERE command, which tells you the location of the execution environment pointer, the Debugger has three other commands that can provide you with useful information. These commands are:

- The INFO command, which displays certain information about a program block.
- The SEGMENTS command, which displays a list of segments that are in use.
- The STATUS command, which displays information pertaining to the status of the debugging environment and debugging operations.

(For information on the WHERE command, see Chapter 5.)

The INFO Command

The INFO command displays information about a program block or statement.

For a program block, the Debugger displays the following information, which is explained in the System Architecture Reference Guide (Rev. 19.2 and higher):

- Procedure base start and end addresses.
- Link base start address, if a procedure was called.
- ECB address, if a procedure was called.
- Most recent invocation's stack frame address, if a procedure was called.
- Starting and ending source line numbers.
- Source filename.
- Mechanism by which the block is invoked — procedure call or short-call. (Short-calls are possible in Prime Macro Assembler (PMA) language with the JSXB instruction and in PL/I Subset G with the OPTIONS SHORT-CALL feature, described in PTU84, PL/I Subset G, Rev. 19.0.)
- Compilation mode (debug or production).

For a statement, the memory address of the first instruction is displayed.

The format of the INFO command is:

```
INFO { program-block-name\   }
     { statement-identifier  }
```

The program-block-name is the name of the program block about which you want information. The statement-identifier, which is defined in Chapter 4, is the executable statement about which you want information.

In the command format, note that a back slash follows the program block name to distinguish this name from a statement label.

To display the information for a program block named AINIT, enter:

```
> INFO AINIT\
Pb start 4001(0)/1000, pb end 4001(0)/1120,
linkframe start 4002(0)/177404,
ecb 4002(0)/11, currently no activations.
Start line # 3, end line # 47 in source file <XYZ>DPR>AINIT.FTN.
Block invoked via procedure call mechanism;
compiled in debug mode.
```

To display the address of the first instruction generated for the statement on source line number 37 in subroutine COMPAD, enter:

```
> INFO COMPAD\37
First instruction at 4001(0)/1061.
```

The SEGMENTS Command

Using the SEGMENTS command, you can display a list of segments in memory that are currently in use. (For more information about default segment use, see the SEG and LOAD Reference Guide, Rev. 19.2.)

The segments are classified by usage, as follows:

- User procedure text, linkage text, and data
- Debugger procedure text
- Debugger linkage text, data, and symbol table
- Procedure call/return stack

The format of the SEGMENTS command, abbreviated SEGS, is:

```
SEGMENTS
```

Here is an example of the SEGMENTS command:

```
> SEGMENTS
User procedure text, linkage text, and data:
   4001, 4002.
Debugger procedure text:
   2040, 2041, 2042, 2115.
Debugger linkage text, data, and symbol table:
   4036, 4037.
Stack:
   4037, 4035.
```

The STATUS Command

Using the STATUS command, you can display the following information about your current debugging environment:

- Statement, entry/exit, and value trace status (on/off)
- Macro execution status (on/off)
- Action list output on breakpoint (print/suppress)
- Execution environment
- Evaluation environment
- Language of evaluation
- Source filename and how it was derived from evaluation or execution environment, or explicitly set by the user

The format of the STATUS command is:

```
STATUS
```

Here is an example of the STATUS command:

```
> STATUS

tracing:
   statement                off
   entry/exit               off
   value                    off
macro execution             on
breakpoint action list      suppress
execution environment       entry to TEST.SORT
evaluation environment      TEST.SORT
evaluation language         PASCAL
source file name            <PRIME>PAUL>TEST.PASCAL (evaluation env)
```

SETTING THE PRINT MODE WITH PMODE

Each user variable has associated with it a print mode, which specifies the format in which the value of the variable will be printed upon evaluation. Initially, the default print mode for each variable corresponds to its declared type. You may override this default in two ways:

- Designate a print mode when using the evaluation (:) command. (See Chapter 6.)
- Set the print mode explicitly with the PMODE command, which is described in the following paragraphs.

Whenever you set a particular print mode for a variable using the PMODE command, that print mode will be used thereafter during your debugging session whenever the variable is evaluated, unless it is overridden by the presence of a print mode modifier in an evaluation command (:).

The format of the PMODE command, abbreviated PM, is:

```
PMODE print-mode variable-1 [,variable-2...]
```

The print-mode is the print mode you want to specify. It can be one of the following: ASCII, BIT, DECIMAL, FLOAT, HEX, OCTAL, or DEFAULT. variable-1, variable-2, etc., are the variables you want evaluated with print-mode.

Table 13-2 lists the results that are printed for each print mode.

Table 13-2
Results of Using Print Modes

Print Mode	Result
ASCII	Each group of eight bits is printed as an ASCII character.
BIT	Each bit is printed as a binary digit.
DECIMAL	Each group of 16 bits is printed as a signed single-precision decimal number.
FLOAT	Each group of 32 bits is printed as a single-precision floating point number.
HEX	Each group of four bits is printed as a hexadecimal digit.
OCTAL	Each group of 16 bits is printed as an unsigned octal number.
DEFAULT	The print mode is set back to the default mode corresponding to the declared type of the variable.

Given the variable OPCODE declared as INTEGER*2, the result of an evaluation would normally be printed in decimal:

```
> : OPCODE
OPCODE = 2471
```

To indicate that OPCODE should be printed in octal, enter:

```
> PMODE OCTAL OPCODE
> : OPCODE
OPCODE = 4647 (o)
```

The lowercase o in parentheses indicates the print mode is octal.

To return to the default print mode (decimal), enter:

```
> PMODE DEFAULT OPCODE
```

To set the print modes of the variables OPCODE, ADDR (contained in block TRANSL), and INFO to bit, enter:

```
> PMODE BIT OPCODE, TRANSL\ADDR, INFO
```

Note

If a specified variable is a PL/I-G or Pascal structure, the print mode of each member of that structure is set to what you specify.

ENTERING PRIME'S V-MODE SYMBOLIC DEBUGGER

You can enter the 64V mode Prime Symbolic Debugger (VPSD) using the VPSD command. The 64V mode Prime Symbolic Debugger, commonly known as VPSD, is a machine-level debugger, which is most helpful to users familiar with the machine architecture. (For a description of the 64V mode Prime Symbolic Debugger, see the Assembly Language Programmer's Guide).

The format of the VPSD command, abbreviated PSD, is:

```
VPSD
```

The VPSD base registers and general machine registers are set up to their values at the time DBG was reentered from the user program. Although you may modify these registers while within VPSD, the updated values are not returned to DBG and therefore not placed in the register set when program execution is continued.

To return to Source Level Debugger command level from VPSD, enter the QUIT command at the VPSD $ prompt. For example:

```
> VPSD

$QUIT
>
```

Caution

Do not set VPSD breakpoints when you enter VPSD from the Debugger. Setting these breakpoints while the Debugger is running could produce unpredictable and undesirable results.

APPENDIXES

A

Sample Sessions with FORTRAN IV

This appendix offers some sample debugging sessions using Prime FORTRAN IV. Other FORTRAN IV debugging examples are given in Chapter 3 and throughout the book. (For more information about FORTRAN IV, see the FORTRAN Reference Guide.)

There are a few things you should keep in mind when using the Debugger with FORTRAN IV. First, you must use the -64V option along with the -DEBUG option when compiling your program. For example, suppose you had a FORTRAN IV program named TEST.FTN. You would enter:

```
OK, DBG TEST -64V -DEBUG
```

Another thing to remember is that, instead of a "program execution complete" message, your program will receive a "program stop at (statement-id)" message if a STOP statement is executed or a "program exit from (statement-id)" if a program block calls EXIT.

Exit breakpoints may not be set in FORTRAN IV program blocks that execute alternate returns; that is, blocks that execute a GOTO statement to a label value supplied as an argument to that block. The following Debugger features will not work if an alternate return is executed: exit breakpoints, exit tracepoints, the OUT command, the CALL command entry/exit, statement, or value tracing. This restriction does not apply to GOTOs executed using the PL1$NL subroutine.

SAMPLE SESSIONS

In the first debugging session, a program that is supposed to take two numbers, add them, and display the results, is used. The program compiles and loads successfully, but when you try to execute it, you get an incorrect answer:

```
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD ADD
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
ENTER AN INTEGER NUMBER FROM 1 TO 10
3
ENTER AN INTEGER NUMBER FROM 1 TO 10
5
   0

**** STOP
```

You enter the Debugger and look at the contents of the source program using the SOURCE command:

```
OK, DBG ADD

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> SOURCE PRINT 23
    1: C TAKE 2 NUMBERS, ADD THEM, AND DISPLAY THE RESULTS
    2: C
    3:       WRITE(1,5)
    4: 5     FORMAT('ENTER AN INTEGER NUMBER FROM 1 TO 10')
    5:       READ(1,7)I
    6: 7     FORMAT(I2)
    7:       WRITE (1,5)
    8:       READ(1,7)J
    9:       CALL ADD(I,J,K)
   10:       WRITE(1,20)K
   11: 20    FORMAT(I4)
   12:       STOP
   13:       END
   14: C
   15:       SUBROUTINE ADD(I,J)
   16:       K=I+J
   17:       RETURN
   18:       END
   19:
BOTTOM
```

After looking at the source code, you set a breakpoint on source line number 9 to try to locate where the error occurred. After setting the breakpoint, you use the RESTART command to start execution:

```
> BREAKPOINT 9
> RESTART

ENTER AN INTEGER NUMBER FROM 1 TO 10
3
ENTER AN INTEGER NUMBER FROM 1 TO 10
5

**** breakpointed at $MAIN\9 ($7+3)
```

You set another breakpoint on source line 10 and check the values of I and J, which were just assigned:

```
> BREAKPOINT 10
> CONTINUE

**** breakpointed at $MAIN\10 ($7+4)
> : I; : J
I = 3
J = 5
```

The values of I and J have been assigned correctly, so you set another breakpoint, this time at the entry of the subroutine ADD. You delete the two previous breakpoints and restart program execution:

```
> BREAKPOINT ADD\\ENTRY
> CLEAR 9
> CLEAR 10
> RESTART

ENTER AN INTEGER NUMBER FROM 1 TO 10
3
ENTER AN INTEGER NUMBER FROM 1 TO 10
5

**** breakpointed at entry to ADD
```

At this entry, you use the ARGUMENTS command to see if the values of I and J were passed to the subroutine correctly:

```
> ARGUMENTS
I = 3
J = 5
```

You see that I and J were passed correctly, but, to your amazement, argument K is not displayed. You wonder what happened to K. Using the SOURCE command, you look at the argument list at the entry to the subroutine:

```
> SOURCE PRINT
  15:       SUBROUTINE ADD(I,J)
```

You have now discovered the problem. The argument K is missing from the argument list. Without this argument, the program does not add I and J correctly. After supplying the argument in the source program, you compile, load, and execute the program again:

```
OK, FTN ADD -64V -DEBUG
0000 ERRORS [<.MAIN.>FTN-REV19.2]
0000 ERRORS [<ADD   >FTN-REV19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD ADD
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
ENTER AN INTEGER NUMBER FROM 1 TO 10
3
ENTER AN INTEGER NUMBER FROM 1 TO 10
5
   8

**** STOP
```

Sure enough, the numbers add correctly. Notice the **** STOP message signifying the completion of program execution.

In the next sample Debugger session, the following FORTRAN IV program is used. Debugger source line numbers are added for convenience:

```
 1:        INTEGER*2 ARRAY(5), TOTAL
 2:        DATA ARRAY/10,200,40,55,78/
 3:        TOTAL=0
 4:        DO 100 J=1,4
 5:        TOTAL=TOTAL+ARRAY(J)
 6:        I=J
 7:  100   CONTINUE
 8:        WRITE(1,200) TOTAL
 9:  200   FORMAT('THE TOTAL OF ARRAY = ',I4)
10:        STOP
11:        END
```

This program is supposed to total the values of all the elements of a five-element array. You compile and load the program successfully, but when you execute it, the output is suspicious. The total isn't as large as you think it is supposed to be:

```
OK, FTN TOTAL -64V -DEBUG
0000 ERRORS [<.MAIN.>FTN-REV19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD TOTAL
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
THE TOTAL OF ARRAY = 305

**** STOP
```

Because the value 305 looks suspicious, you enter the Debugger:

```
OK, DBG TOTAL

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

>
```

To see if the elements of the array were assigned correctly, you place a breakpoint on source line number 4 just before the DO loop, restart program execution, and check the values of the array elements:

```
> BREAKPOINT 4
> RESTART

**** breakpointed at $MAIN\4
> : ARRAY
ARRAY(1) = 10
ARRAY(2) = 200
ARRAY(3) = 40
ARRAY(4) = 55
ARRAY(5) = 78
```

You see that the values were assigned to the array correctly. Now, since the program is so small, you decide to use the WATCH command to trace the changing values of variables J and TOTAL to see where and how new values are assigned throughout execution. You also place a breakpoint on source line 8 to suspend execution at the exit of the DO loop, after all values have been assigned:

```
> WATCH J, TOTAL
> CLEAR 4
> BREAKPOINT 8
> RESTART
The value of $MAIN\TOTAL has been changed at $MAIN\4
  from 305
  to   0
The value of $MAIN\J has been changed at $MAIN\5
  from 4
  to   1
The value of $MAIN\TOTAL has been changed at $MAIN\6
  from 0
  to   10
The value of $MAIN\J has been changed at $MAIN\5
  from 1
  to   2
The value of $MAIN\TOTAL has been changed at $MAIN\6
  from 10
  to   210
The value of $MAIN\J has been changed at $MAIN\5
  from 2
  to   3
The value of $MAIN\TOTAL has been changed at $MAIN\6
  from 210
  to   250
The value of $MAIN\J has been changed at $MAIN\5
  from 3
  to   4
The value of $MAIN\TOTAL has been changed at $MAIN\6
  from 250
  to   305

**** breakpointed at $MAIN\8 ($100+1)
```

Now you have discovered the problem. The array index value of 5 is never assigned to J, and the value of ARRAY(5) is never added to TOTAL. Using the SOURCE command, you look at the DO LOOP index on line 4:

```
> SOURCE POINT 4
   4:       DO 100 J=1,4
```

As you had suspected, a 4 was assigned as the maximum iteration instead of a 5. So you change 4 to 5 in your source program and compile, load, and execute your program again:

```
OK, FTN TOTAL -64V -DEBUG
0000 ERRORS [<.MAIN.>FTN-REV19.2]
OK, SEG -LOAD
[SEG REV 19.2]
$ LOAD TOTAL
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
THE TOTAL OF ARRAY =  383

**** STOP

> QUIT
OK,
```

Your program now executes correctly.

B

Sample Sessions with FORTRAN 77

This appendix offers a sample debugging session with Prime FORTRAN 77. Other FORTRAN debugging sessions are given in Appendix A, SAMPLE SESSIONS WITH FORTRAN IV. (For more information on FORTRAN 77, see the FORTRAN 77 Reference Guide.)

One thing to remember when debugging FORTRAN 77 programs is that, instead of a "program execution complete" message, you receive a "program stop at (statement-id)" message if a STOP statement is executed, or a "program exit from (statement-id)" message if a program block calls EXIT.

In FORTRAN 77, if execution is suspended at an entry to a program block, you cannot evaluate adjustable character arguments or adjustable or assumed-size arrays. If the program is executed up to the first statement, these values can be evaluated.

SAMPLE SESSION

Suppose you have just written a FORTRAN 77 program that calculates an employee's yearly salary, monthly salary, percentage of a pay raise over last year's salary, and the percentage of the salary that goes toward deductions. The program contains one subroutine, RAISE, that performs these calculations. The program looks like this, with Debugger source line numbers included for convenience:

```
 1:          PROGRAM PAYRAISE
 2: C
 3: C     KEY
 4: C       A = NEW_GROSS_PAY
 5: C       B = NEW_YEARLY_PAY
 6: C       C = NEW_MONTHLY_GROSS
 7: C       D = OLD_GROSS_PAY
 8: C       E = OLD_NET
 9: C       F = NEW_NET_PAY
10: C       G = PERCENT_INCREASE
11: C       H = PERCENT_DEDUCTIONS
12: C
13:         PRINT*,'ENTER YOUR WEEKLY GROSS PAY:      '
14:         READ*, D
15:         PRINT*,'ENTER YOUR WEEKLY NET PAY:    '
16:         READ*, E
17:         PRINT*,'ENTER YOUR NEW WEEKLY GROSS PAY:     '
18:         READ*, A
19:         PRINT*,'ENTER YOUR NEW WEEKLY NET PAY:     '
20:         READ*, F
21:         CALL RAISE (A,B,C,D,E,F,G,H)
22:         PRINT 10,'YEARLY: $',B
23:         PRINT 10,'MONTHLY: $',C
24:         PRINT 20,'INCREASE: ',G,'
25:         PRINT 20,'DEDUCTIONS: ',H,'
26: 10      FORMAT (A,F8.2)
27: 20      FORMAT (A,I2, A)
28:         STOP
29:         END
30:
31:         SUBROUTINE RAISE (A,B,C,D,E,F,G,H)
32:         B = A*52
33:         C = B/12
34:         G = (A-D)/D*100
35:         H = (A-F)/F*10
36:         RETURN
37:         END
```

You compile and load the program successfully, but when it executes, you see that the percentage of deductions, represented by the variable H, is quite a bit lower than it should be:

```
OK, F77 PAYRAISE -DEBUG
[F77 Rev. 19.2]
0000 ERRORS [<PAYRAISE> F77-REV 19.2]
0000 ERRORS [<RAISE> F77-REV 19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD PAYRAISE
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
 ENTER YOUR WEEKLY GROSS PAY:
333.33
 ENTER YOUR WEEKLY NET PAY:
266.66
 ENTER YOUR NEW WEEKLY GROSS PAY:
399.99
 ENTER YOUR NEW WEEKLY NET PAY:
333.33
YEARLY: $20799.48
MONTHLY: $ 1733.29
INCREASE:  20%
DEDUCTIONS:  2%
**** STOP

OK,
```

Yes, something is definitely wrong with the deduction calculation because you know Uncle Sam takes more than 2 percent of your gross salary. So you enter the Debugger, place a breakpoint on source line 21, just before the call to subroutine RAISE, and restart program execution.

```
OK, DBG PAYRAISE

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT 21
> RESTART
 ENTER YOUR WEEKLY GROSS PAY:
333.33
 ENTER YOUR WEEKLY NET PAY:
266.66
 ENTER YOUR NEW WEEKLY GROSS PAY:
399.99
 ENTER YOUR NEW WEEKLY NET PAY:
333.33

**** breakpointed at PAYRAISE\21
>
```

You then check the values that were just assigned to variables D, E, A, and F. Values appear in scientific notation:

```
**** breakpointed at PAYRAISE\21
> : D
D = 3.333299E+02
> : E
E = 2.666599E+02
> : A
A = 3.999899E+02
> : F
F = 3.333299E+02
>
```

Seeing that the values were assigned correctly, you step into subroutine RAISE and check to see if the values were passed correctly:

```
> STEPIN

**** "in" completion at RAISE\32
> ARGUMENTS
H = -2.869004E+20
G = -1.784781E+24
F = 3.333299E+02
E = 2.666599E+02
D = 3.333299E+02
C = 9.158859E+14
B = 2.074890E+34
A = 3.999899E+02
>
```

You see that the values were passed to the subroutine correctly. (Variables H and G contain junk because their values have not been assigned yet.) Now you place a breakpoint on line 36, immediately after the value of H has been calculated, continue execution to line 36, then check the value of H:

```
> BREAKPOINT 36
> CONTINUE

**** breakpointed at RAISE\36
> : H
H = 1.999821E+00
>
```

Looking at the value of H, you discover that the problem is within the statement that calculates H because the exponent should be 01 instead of 00. Using the SOURCE command, you look at line 35, which is the statement that calculates H:

```
> SOURCE NEXT -1
  35:       H = (A-F)/F*10
>
```

You see that you accidentally multiplied F by 10 instead of 100. To be absolutely sure, you use the LET command to assign H the new formula (A-F)/F*100, then check the value of H again:

```
> LET H = (A-F)/F*100
> : H
H = 1.999820E+01
>
```

Yes, the exponent becomes 01. And now you correct the error in your source file, compile, load, and execute your program once again:

```
OK, F77 PAYRAISE -DEBUG
[F77 Rev. 19.2]
0000 ERRORS [<PAYRAISE> F77-REV 19.2]
0000 ERRORS [<RAISE> F77-REV 19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD PAYRAISE
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
 ENTER YOUR WEEKLY GROSS PAY:
333.33
 ENTER YOUR WEEKLY NET PAY:
266.66
 ENTER YOUR NEW WEEKLY GROSS PAY:
399.99
 ENTER YOUR NEW WEEKLY NET PAY:
333.33
YEARLY: $20799.48
MONTHLY: $ 1733.29
INCREASE: 20%
DEDUCTIONS: 20%
**** STOP

OK,
```

You now get correct results.

C

Sample Sessions with Pascal

This appendix offers a sample debugging session with Prime Pascal. Several other debugging sessions using Pascal programs are given in Chapter 3 and throughout the book. The Debugger supports Prime's extensions and restrictions to Pascal, including the new data type STRING, which was released at Rev. 19.2. (For more information on Prime Pascal, see the Pascal Reference Guide.)

Consider the following Pascal program named CALENDAR. Debugger source line numbers are added for convenience:

```
 1: PROGRAM CALENDAR;
 2: TYPE
 3:   MONTH = (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
 4:            JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER);
 5:   YEAR  = INTEGER;
 6:   DATE  = 28..31;
 7: FUNCTION DAYS_IN_MONTH (MONTH_IN : MONTH; YEAR_IN : YEAR) : DATE;
 8:   BEGIN {function DAYS_IN_MONTH}
 9:     CASE MONTH_IN OF
10:         JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER:
11:             DAYS_IN_MONTH := 31;
12:         APRIL, JUNE, SEPTEMBER, NOVEMBER:
13:             DAYS_IN_MONTH := 30;
14:         FEBRUARY:
15:             IF YEAR_IN DIV 4 = 0 THEN
16:                 DAYS_IN_MONTH := 29
17:             ELSE
18:                 DAYS_IN_MONTH := 28
```

```
19:     END;
20:   END; {function DAYS_IN_MONTH}
21: BEGIN {main program}
22:   WRITELN (DAYS_IN_MONTH (FEBRUARY, 1984), ' DAYS HATH FEBRUARY.')
23: END.
```

This program is supposed to output the correct number of days in any given month, in any given year. It has one function, DAYS_IN_MONTH, which, after being passed a month and year, calculates the number of days in the given month. You compile the program with the -DEBUG option and load it successfully:

```
OK, PASCAL CALENDAR -DEBUG
[PASCAL Rev. 19.2]
0000 ERRORS (PASCAL-REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD CALENDAR
$ LIBRARY PASLIB
$ LIBRARY
LOAD COMPLETE
$ QUIT
OK,
```

You execute the program, knowing that the supplied month and year, FEBRUARY 1984, has 29 days because 1984 is a leap year:

```
OK, SEG CALENDAR
        28 DAYS HATH FEBRUARY.
OK,
```

Looking at your output, you discover that it is wrong — 28 days instead of 29. So you place two breakpoints at potential problem areas, the entry to DAYS_IN_MONTH and the executable statement that handles FEBRUARY within the CASE structure:

```
OK, DBG CALENDAR

**Dbg**   revision 1.0 - 19.1 (16-June-1983)

> BREAKPOINT DAYS_IN_MONTH\\ENTRY
> BREAKPOINT 15
CALENDAR.DAYS_IN_MONTH\15 assumed.
>
```

Note

In the example above, the Debugger assumes the DAYS_IN_MONTH evaluation environment when you specify the second breakpoint. As of this software release, you do not need to specify the names of procedures or functions as long as they are declared within the current evaluation environment.

You activate execution using RESTART, and execution suspends at the entry to the DAYS_IN_MONTH function:

```
> RESTART

**** breakpointed at entry to CALENDAR.DAYS_IN_MONTH
>
```

At the entry to the function, you want to make sure the arguments MONTH_IN and YEAR_IN were passed correctly, so you use the ARGUMENTS command:

```
> ARGUMENTS
MONTH_IN = FEBRUARY
YEAR_IN = 1984
>
```

Seeing that the arguments were passed correctly, you continue execution until the breakpoint at the IF-THEN-ELSE statement, which calculates the number of days in FEBRUARY:

```
> CONTINUE

**** breakpointed at CALENDAR.DAYS_IN_MONTH\15
>
```

You use the evaluation command to check the result of the expression YEAR_IN DIV 4:

```
> : YEAR_IN DIV 4
496
>
```

To your surprise, you see that the value of YEAR_IN DIV 4 is nowhere near 0, and you realize you used the wrong arithmetic operator to calculate the leap year. What you really wanted was the MOD operator to yield a remainder of 0. So after changing DIV to MOD in the source program, you compile, load, and execute your program over again:

```
OK, PASCAL CALENDAR -DEBUG
[PASCAL Rev. 19.2]
0000 ERRORS (PASCAL-REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD CALENDAR
$ LIBRARY PASLIB
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
        29 DAYS HATH FEBRUARY.
OK,
```

The output is now correct, and FEBRUARY 1984 hath 29 days after all.

D

Sample Sessions with PL/I Subset G

This appendix offers a sample debugging session with a Prime PL/I Subset G program. Other debugging examples with PL/I-G are used throughout the book. (For more information about PL/I-G, see the PL/I Subset G Reference Guide.)

When you debug a PL/I-G program, it is recommended that you do not include the keywords OPTIONS (MAIN) to designate the main procedure. The term OPTIONS (MAIN), which is not necessary in Prime PL/I-G, may generate an error during your debugging session. It is better just to enter the comment /*options main*/ if you want to flag the main procedure.

SAMPLE SESSION

Consider the following PL/I Subset G program. Debugger source line numbers have been added for convenience.

```
1: CALC : PROCEDURE; /*options main*/
2:    DCL NUM FIXED BIN (15);
3:    PUT SKIP LIST ('Enter 0 to end program');
4:    PUT SKIP;
5:    PUT LIST('Enter a whole number: ');
6:    GET SKIP LIST (NUM);
7:    DO WHILE (NUM ^= 0);
8:      IF MOD(NUM, 2) ^= 0 THEN
9:        PUT SKIP LIST('This is an odd number');
```

```
      ELSE
        CALL CUBE_IT(NUM);
    END; /*do-while*/
    CUBE_IT : PROCEDURE(N);
      DCL (N, NSQR, NCUB) FIXED BIN (15);
      NSQR = N ** N;
      NCUB = N * NSQR;
      PUT SKIP LIST(N, NSQR, NCUB);
    END CUBE_IT;
  END CALC;
```

This program is supposed to read a whole number, which you enter at the terminal, and determine if the number is odd or even. If the number is odd, a message saying it is odd is displayed, and you are prompted for another number. If the number is even, a procedure CUBE_IT is called, and the value of the even number, its square, and its cube are displayed. You are then prompted for another number. The program is designed to terminate when you enter a 0.

Your program compiles and loads without error, but when you execute it, a couple of horrible things happen:

```
OK, PL1G CUBE -DEBUG
[PL1G Rev. 19.2]
0000 ERRORS (PL1G-REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD CUBE
$ LIBRARY PL1GLB
$ LIBRARY
LOAD COMPLETE
$ EXECUTE

Enter 0 to end program
Enter a whole number:  4

        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
        4           256          1024
            This goes on forever.
```

You had entered a 4, hoping to see its square, 16, and its cube, 64. Instead, you get back two very large numbers that come back at you over and over again in an infinite loop.

First, you would like to know why those large values were assigned to NSQR and NCUB. You enter the Debugger and place breakpoints at the entry to CUBE_IT and source line 16, where NSQR is calculated. Then you restart the program, suspend at the entry, and see if the value of N was passed to CUBE_IT correctly:

```
OK, DBG CUBE

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> BREAKPOINT CUBE_IT\\ENTRY
> BREAKPOINT 16
CALC.CUBE_IT\16 assumed.
> RESTART

Enter 0 to end program
Enter a whole number:  4

**** breakpointed at entry to CALC.CUBE_IT
> : NUM
NUM = 4
>
```

Note

In the example above, notice how the Debugger assumes the CUBE_IT evaluation environment when you specify the second breakpoint. As of this software release, you do not need to specify the name of a procedure as long as it is declared within the current evaluation environment.

Seeing that the value was passed correctly, you continue program execution to line 16 and check the values of N and NSQR:

```
> CONTINUE

**** breakpointed at CALC.CUBE_IT\16
> : 4; : NSQR
4
NSQR = 256
>
```

The value of N is still correct, but the value of NSQR is not. Something is wrong with the calculation of NSQR. You look at the source line on which the calculation is made and perform the calculation again, using the LET command:

```
> : N
N = 4
> LET N = N ** N
> : N
N = 256
>
```

Suddenly you realize that you used the wrong operator. You accidentally used an exponent of 4 instead of simply multiplying 4 times 4. To be sure, you evaluate the expression 4 * 4:

```
> : N * N
16
>
```

One problem has been solved. Now you want to find out why your program dives into an infinite loop. You expect the problem is within the DO-WHILE structure. You use the SOURCE command to examine the source code in the DO-WHILE. Then you break at entry of the loop and check the value of NUM:

```
> SOURCE POINT 6
   6:   GET SKIP LIST (NUM);
> SOURCE PRINT 7
   6:   GET SKIP LIST (NUM);
   7:   DO WHILE (NUM ^= 0);
   8:     IF MOD(NUM, 2) ^= 0 THEN
   9:       PUT SKIP LIST('This is an odd number');
  10:     ELSE
  11:       CALL CUBE_IT(NUM);
  12:   END; /*do-while*/
> BREAKPOINT 8
> RESTART

Enter 0 to end program
Enter a whole number:  4

**** breakpointed at CALC\8
> : NUM
NUM = 4
>
```

After seeing that the value of NUM has entered the loop correctly, you decide to single step through the loop and the first call to CUBE_IT, checking the value of NUM as you go. You create two macros that accomplish this, then single step into the procedure CUBE_IT:

```
> MACRO SP [STEP; : NUM; SOURCE PRINT]
> MACRO SIN [STEPIN; : NUM; SOURCE PRINT]
> SP

**** "step" completion at CALC\10
NUM = 4
   10:     ELSE
> SIN

**** "in" completion at CALC.CUBE_IT\15
NUM = 4
   15:     NSQR = N * N;
>
```

Everything is executing nicely so far. Using the OUT command, you leave the CUBE_IT procedure and go back to the DO-WHILE loop. You then resume single stepping to finish one full cycle of the DO-WHILE loop:

```
> OUT

        4           16          64
**** "out" completion at exit from CALC.CUBE_IT into CALC\12
> SP

**** "step" completion at CALC\12
NUM = 4
   12:   END; /*do-while*/
> SP

**** breakpointed at CALC\8
NUM = 4
    8:     IF MOD(NUM, 2) ^= 0 THEN
>
```

But of course. You see that you never allowed the program to get a new value for NUM inside the DO-WHILE loop. That is why the value of NUM remains 4 at the beginning of the second execution of the DO-WHILE. You add two lines of code at the bottom of the loop:

```
PUT SKIP LIST ('Enter a whole number: ');
GET SKIP LIST (NUM);
```

You recompile and reload the program successfully, then execute it once again:

```
OK, SEG CUBE

Enter 0 to end program
Enter a whole number:  4

        4           16           64
Enter a whole number:  5

This is an odd number
Enter a whole number:  0
OK,
```

This time the program executes properly, producing correct results.

E

Sample Sessions with COBOL 74

This appendix offers sample debugging sessions with Prime COBOL 74. The first section discusses some special considerations that you should know about when using the Debugger with COBOL 74 programs. (For more information about COBOL 74, see the COBOL 74 Reference Guide.)

SPECIAL CONSIDERATIONS

Data Types in COBOL 74

When using the Debugger with COBOL 74, you should understand the correspondence of data type names between the two. In other words, the name given to a COBOL 74 data type by the Debugger may differ from the official data type name defined by the language. The following table lists COBOL 74 data type names with their corresponding Debugger names.

COBOL 74	Debugger
ALPHANUMERIC DISPLAY (PIC X)	alphanumeric
NUMERIC DISPLAY (PIC 9)	trailing overpunch
COMPUTATIONAL	binary-1
COMPUTATIONAL-1 (real)	computational-1
COMPUTATIONAL-2 (double precision real)	computational-2
COMPUTATIONAL-3 (packed decimal)	computational-3

Some data types known to the Debugger do not exist in COBOL. Thus, some of the built-in functions listed in Chapter 6 cannot be used to evaluate expressions. These functions are built-in Pascal, PL/I-G, and FORTRAN functions, which are known to the Debugger. (Chapter 6 does not list the data type required by each function, so you will need the Pascal Reference Guide, the PL/I Subset G Reference Guide, the FORTRAN 77 Reference Guide and the FORTRAN Reference Guide to find out if each function can be used with COBOL 74.)

Breakpoints in COBOL 74

Breakpoints and tracepoints may be set on paragraph headings in COBOL, with the format BREAKPOINT CALLER\MAIN-PARAGRAPH. If a paragraph heading begins with a number, you must add a dollar sign ($) before the heading so that the Debugger does not mistake the paragraph heading for a line number. Thus, if a paragraph-name is 020-BEGIN, you would breakpoint it with the following:

```
> BREAKPOINT CALLER\$020-BEGIN.
```

A paragraph heading is also referred to as a "label" throughout this book. (See Chapter 4 for definitions of labels.)

Program Blocks

COBOL does not support procedures as they are known to Pascal and PL/I-G. However, any called program acts like a procedure. In this book, any program, procedure, function, subroutine, or other program unit is generically defined as a program block. (See Chapter 4 for a definition of "program block" in the context of COBOL 74.)

Data Initialization on Restarts

Data variables that are initialized in the WORKING-STORAGE section of a COBOL program are not reinitialized when the program is rerun with RESTART. Thus, to test whether a variable is being changed correctly, you may need to use the Debugger's LET command to reinitialize some data elements before restarting.

Record Element Names

Although the Debugger lists record elements in the form NAME1.NAME2.NAME3, if the language is defined as COBOL, you must nevertheless enter these elements in the COBOL format, as NAME1 OF NAME2 OF NAME3.

SESSION ONE -- EXPLORING WITH THE DEBUGGER

In the first example, there are no apparent bugs. The program fills a record (REC) with data, moves it to an output buffer (DISP-REC), and displays it on the terminal. Here is the COBOL 74 program.

```
OK, SLIST EX1.CBL
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ADDRESS.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. PRIME.
       OBJECT-COMPUTER. PRIME.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 REC.
          05 NAME.
             10 LAST-NAME PIC X(20).
             10 FIRST-NAME PIC X(10).
             10 MIDDLE-INIT PIC X.
          05 ADDR.
             10 STREET PIC X(25).
             10 CITY PIC X(20).
             10 STATE PIC X(2).
             10 ZIP PIC X(5).
       01 DISP-REC.
          05 LINE1.
             10 FIRST-NAME PIC X(10).
             10 FILLER PIC X VALUE ' '.
             10 MIDDLE-INIT PIC X.
             10 FILLER PIC XX VALUE '. '.
             10 LAST-NAME PIC X(20).
          05 LINE2.
             10 STREET PIC X(25).
          05 LINE3.
             10 CITY PIC X(20).
             10 FILLER PIC XX VALUE ', '.
             10 STATE PIC X(2).
             10 FILLER PIC XX VALUE '  '.
             10 ZIP PIC X(5).
       PROCEDURE DIVISION.
       CREATE-TEST-RECORD.
           MOVE 'HARPER' TO LAST-NAME OF REC.
           MOVE 'JAMES' TO FIRST-NAME OF REC.
           MOVE 'R' TO MIDDLE-INIT OF REC.
           MOVE '25 MAIN ST.' TO STREET OF REC.
           MOVE 'ANYTOWN' TO CITY OF REC.
           MOVE 'MA' TO STATE OF REC.
           MOVE '12345' TO ZIP OF REC.
       SET-UP-OUTPUT.
           MOVE CORRESPONDING NAME TO LINE1.
           MOVE STREET OF ADDR TO STREET OF LINE2.
           MOVE CITY OF ADDR TO CITY OF LINE3.
           MOVE STATE OF ADDR TO STATE OF LINE3.
           MOVE ZIP OF ADDR TO ZIP OF LINE3.
       DISPLAY-OUTPUT.
           DISPLAY LINE1.
           DISPLAY LINE2.
           DISPLAY LINE3.
```

Now you compile and load the program with the -DEBUG option:

```
OK, CBL EX1 -DEBUG

[CBL rev 19.2]
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD EX1
$ LIBRARY CBLLIB
$ LIBRARY
LOAD COMPLETE
$ QUIT
```

Then you enter the Debugger and use the SOURCE command to examine the code and decide where to put breakpoints so that data can be examined before and after the MOVE statements. You then use RESTART to run the program:

```
OK, DBG EX1

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> SOURCE LOCATE PROCEDURE
   34:          PROCEDURE DIVISION.
> SOURCE PRINT 23
   34:          PROCEDURE DIVISION.
   35:          CREATE-TEST-RECORD.
   36:              MOVE 'HARPER' TO LAST-NAME OF REC.
   37:              MOVE 'JAMES' TO FIRST-NAME OF REC.
   38:              MOVE 'R' TO MIDDLE-INIT OF REC.
   39:              MOVE '25 MAIN ST.' TO STREET OF REC.
   40:              MOVE 'ANYTOWN' TO CITY OF REC.
   41:              MOVE 'MA' TO STATE OF REC.
   42:              MOVE '12345' TO ZIP OF REC.
   43:          SET-UP-OUTPUT.
   44:              MOVE CORRESPONDING NAME TO LINE1.
   45:              MOVE STREET OF ADDR TO STREET OF LINE2.
   46:              MOVE CITY OF ADDR TO CITY OF LINE3.
   47:              MOVE STATE OF ADDR TO STATE OF LINE3.
   48:              MOVE ZIP OF ADDR TO ZIP OF LINE3.
   49:          DISPLAY-OUTPUT.
   50:              DISPLAY LINE1.
   51:              DISPLAY LINE2.
   52:              DISPLAY LINE3.
BOTTOM
> BREAKPOINT 36
> BREAKPOINT DISPLAY-OUTPUT
> RESTART

**** breakpointed at ADDRESS\36 (CREATE-TEST-RECORD+1)
```

Now you use the evaluation command to examine all of the REC record structure:

```
> : REC
REC.NAME.LAST-NAME = ''
REC.NAME.FIRST-NAME = ''
REC.NAME.MIDDLE-INIT = ''
REC.ADDR.STREET = ''
REC.ADDR.CITY = ''
REC.ADDR.STATE = ''
REC.ADDR.ZIP = ''
> CONTINUE

**** breakpointed at ADDRESS\50 (DISPLAY-OUTPUT+1)
> : REC
REC.NAME.LAST-NAME = 'HARPER                '
REC.NAME.FIRST-NAME = 'JAMES      '
REC.NAME.MIDDLE-INIT = 'R'
REC.ADDR.STREET = '25 MAIN ST.                '
REC.ADDR.CITY = 'ANYTOWN              '
REC.ADDR.STATE = 'MA'
REC.ADDR.ZIP = '12345'
> CONTINUE
```

Finally the program displays the information on the screen, as expected, and terminates normally:

```
JAMES      R. HARPER
25 MAIN ST.
ANYTOWN              , MA  12345

**** Program execution complete.
```

But now you rerun the program, test the data name REC before information is moved there, and find that it has not been reinitialized to blanks:

```
> RESTART

**** breakpointed at ADDRESS\36 (CREATE-TEST-RECORD+1)
> : REC
REC.NAME.LAST-NAME = 'HARPER                '
REC.NAME.FIRST-NAME = 'JAMES      '
REC.NAME.MIDDLE-INIT = 'R'
REC.ADDR.STREET = '25 MAIN ST.                '
REC.ADDR.CITY = 'ANYTOWN             '
REC.ADDR.STATE = 'MA'
REC.ADDR.ZIP = '12345'
```

So you reinitialize some of the data:

```
> LET LAST-NAME OF NAME OF REC = ''
> LET ZIP OF ADDR OF REC = '00000'
> CONTINUE
```

Now the two data items in question have been reinitialized correctly:

```
**** breakpointed at ADDRESS\36 (CREATE-TEST-RECORD+1)
> : REC
REC.NAME.LAST-NAME = '                      '
REC.NAME.FIRST-NAME = 'JAMES      '
REC.NAME.MIDDLE-INIT = 'R'
REC.ADDR.STREET = '25 MAIN ST.                '
REC.ADDR.CITY = 'ANYTOWN             '
REC.ADDR.STATE = 'MA'
REC.ADDR.ZIP = '00000'
> QUIT
```

SESSION TWO -- THINGS GET WORSE

The next session handles two programs, one of them calling the other. CALLER passes a structure (A1) to CALLED. CALLED, to which the structure is known as ARG1, puts values in it and passes control back to CALLER, which displays the values in the structure. The combination has a couple of bugs:

```
OK, SLIST CALLER.CBL
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CALLER.
      *
       DATA DIVISION.
      *
       WORKING-STORAGE SECTION.
       01 A1.
           02 A2                       COMP PIC S999 VALUE 0.
           02 A3                       PIC XX VALUE 'A3'.
           02 A4.
              03 A5                    PIC XX VALUE 'A5'.
              03 A6                    PIC S9(4) VALUE 0.
           02 A7                       PIC XX VALUE 'A7'.
       77 ACCEPTED-CALLS               PIC X(5).
       77 WORK-CALLS                   PIC X(5) JUSTIFIED RIGHT.
       77 NUM-OF-CALLS                 PIC S9(5).
      *
       PROCEDURE DIVISION.
       001-BEGIN.
           EXHIBIT A3.
           DISPLAY 'ENTER THE NUMBER OF CALLS DESIRED: '
           ACCEPT ACCEPTED-CALLS.
           UNSTRING ACCEPTED-CALLS DELIMITED BY SPACE INTO WORK-CALLS.
           MOVE WORK-CALLS TO NUM-OF-CALLS.
           PERFORM 002-CALLED NUM-OF-CALLS TIMES.
           EXHIBIT A2.
           EXHIBIT A3.
           EXHIBIT A6.
           DISPLAY 'END OF RUN'.
           STOP RUN.
       002-CALLED.
           CALL 'CALLED' USING A1.
```

```
OK, SLIST CALLED.CBL
     IDENTIFICATION DIVISION.
     PROGRAM-ID. CALLED.
    *
     DATA DIVISION.
    *
     LINKAGE SECTION.
     01 ARG1.
         02 B1                    COMP PIC S999.
         02 B2                    PIC XX.
         02 B3.
            03 B4                 PIC XX.
            03 B5                 PIC S9(4).
         02 B6                    PIC XX.
    *
     PROCEDURE DIVISION USING ARG1.
         ADD 1 TO B1.
         MOVE 'B2' TO B2.
         MOVE 'B4' TO B4.
         ADD 99999 TO B5.
         MOVE 'B6' TO B6.
         GOBACK.
```

You invoke the Debugger and run the program with RESTART:

```
OK, DBG CALLER

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> RESTART
A3 = A3
ENTER THE NUMBER OF CALLS DESIRED:
3
A2 =        3
A3 = B2
A6 =    9997
END OF RUN

EXIT.  Program exit from CALLER\30.
```

As soon as the program displays the final values, you realize that A3, instead of displaying the value B3, displayed the value B2, and A6, which should have a value that is a multiple of 99999, displays another value. So, you set all values in the structure back to the original WORKING-STORAGE values in CALLER and set three breakpoints: one in CALLER before control passes to CALLED, one in CALLER after control returns, and one in CALLED after CALLED sets values:

```
> LET A2 = 0
> LET A3 = 'A3'
> LET A6 = 0
> BREAKPOINT 25
> BREAKPOINT 29
> BREAKPOINT CALLED\20
> RESTART
A3 = A3
ENTER THE NUMBER OF CALLS DESIRED:
3

**** breakpointed at CALLER\25
```

You evaluate the structure and see that the reinitialized values (A2, A3, A6) are as expected before the call:

```
> : A1
A1.A2 = 0
A1.A3 = 'A3'
A1.A4.A5 = 'B4'
A1.A4.A6 = 0
A1.A7 = 'B6'
> CONTINUE
```

Then you let three calls be made to CALLED:

```
**** breakpointed at CALLED\20
> CONTINUE

**** breakpointed at CALLED\20
> CONTINUE

**** breakpointed at CALLED\20
```

After the third call, you evaluate the data while control is still with the called program:

```
> : ARG1
ARG1.B1 = 3
ARG1.B2 = '__'
ARG1.B3.B4 = ''
ARG1.B3.B5 = 0
ARG1.B6 = ''
```

The evaluation command shows that the elements of ARG1 are not parallel to the elements of A1 in the calling program. You intended a correspondence between A2 and B2.

It becomes evident also that argument B5 never changes with ADDs. You check whether the declaration of B5 allows successive additions of so large a number:

```
> TYPE A6
trailing overpunch (4)
```

The item A6 only allows four digits or a maximum value of 9999. Thus, even one addition of 99999 in CALLED causes the value to overflow. This confirms that A6, which corresponds to B5, cannot be used for repeated additions of the number. These items should be defined as much larger than PIC 9(4).

F

Sample Sessions with RPG II

This appendix offers a sample debugging session with Prime RPG II and special considerations to know about when using the Debugger with RPG. (For more information on RPG II, see the RPG II V-mode Compiler Reference Guide.)

SPECIAL CONSIDERATIONS

Using Breakpoints with RPG

In RPG programs, you may set breakpoints only on calculation statements. Breakpoints cannot be set on any other type of statements because, unlike calculation statements, other statements are not executable.

Using SOURCE with RPG

When you use the SOURCE command to examine the contents of your source RPG program or output file, be aware that source line numbers are added to the beginning of each line. If your source program or output file is set up for 80 columns, some lines could wrap around to the next line when they are displayed at your terminal.

Evaluating Variables in RPG

When evaluating variables in RPG, you should understand that the names given to RPG variable types by the Debugger differ from the official type names defined by the language. The following table lists RPG variable type names with their corresponding Debugger data type names:

RPG Variable Type	Debugger Data Type
Field	alphanumeric or trailing overpunch
Data Structure	alphanumeric
Array	alphanumeric or trailing overpunch
Table	alphanumeric or trailing overpunch
Table Index	binary-1 (15)
Indicator	binary-1 (15) external

Arrays and tables are one dimensional arrays and are referenced as described in Chapter 6. Each table within RPG has an internal index that references the currently selected element of the table. The internal index is referenced by the name IX$yyy, where yyy are the last characters of the table TAByyy. For example, IX$ABC contains the current index of table TABABC. You may change this internal index with the LET command. Note that the internal index should only contain integers within the index range of the table.

RPG indicators can be referenced by the name IND$xx, where xx is any legal RPG indicator. For example, IND$L3 is a reference for the L3 indicator. The value for an indicator is always 0 or 1.

Using RESTART with RPG

If you have suspended execution in the calculation part of your program, and you have specified CONSOLE as the input device, then you may use RESTART to rerun execution -- you supply the input as your program executes. However, if you specify DISK as the input device, you cannot use RESTART to rerun the program unless you close the disk file from which your program is reading data. To accomplish this, enter the Debugger's ! command followed by the PRIMOS command CLOSE with the name of the file. Then you can enter RESTART. For example:

```
> ! CLOSE INPUT
> RESTART
```

Examining the Input or Output File

If you have specified DISK as the input device and DISK or PRINTER as the output device, and want to examine either file while program execution is suspended, enter the Debugger's ! command followed by the PRIMOS command CLOSE with the name of the file. Then use the SOURCE NAME command to view the file:

```
> ! CLOSE filename
> SOURCE NAME filename
```

The filename is the name of the input or output file you want to examine. Be aware that if you examine your input or output file in this manner, your file will be closed, and you cannot continue program execution without a RESTART, which will open the file once again.

SAMPLE SESSION WITH RPG

Consider the following RPG program. Debugger source line numbers have been added for convenience:

```
 1:      F*
 2:      FINFO    IP  F      80            DISK
 3:      FREPORT  O   F      80            PRINTER
 4:      IINFO    AA  01
 5:      I                                      1   50NUMBER
 6:      I                                      6   22 NAME
 7:      I                                     23   252RATE
 8:      I                                     26   280HOURS
 9:      C   01       HOURS     COMP 40                    102020
10:      C   20       RATE      MULT HOURS     PAY     62
11:      C   10                 EXSR SUBRU
12:      C   20       PAY       ADD  0         TOTPAY  62
13:      C   10       PAY       ADD  BONUS     TOTPAY  62
14:      CSR          SUBRU     BEGSR
15:      CSR          RATE      MULT 40        PAY     62
16:      CSR          RATE      MULT 5         OTRATE  42
17:      CSR          OTRATE    ADD  RATE      OTWAGE  52
18:      CSR          HOURS     SUB  40        OTHRS   20
19:      CSR          OTWAGE    MULT OTHRS     BONUS   52
20:      CSR                    ENDSR
21:      OREPORT  H   201   1P
22:      O                          UDATE Y    9
23:      O                                    38 'WEEKLY'
24:      O                                    47 'EARNINGS'
25:      O        H   2     1P
26:      O                                     8 'NUMBER'
27:      O                                    18 'NAME'
28:      O                                    34 'RATE'
29:      O                                    41 'HOURS'
30:      O                                    52 'OT PAY'
31:      O                                    65 'TOTAL PAY'
32:      O        D   1     01
33:      O                          NUMBER     8
34:      O                          NAME      27
35:      O                          RATE  1   34
36:      O                          HOURS 1   40
37:      O                          BONUS 1B  52 '$'
38:      O                          TOTPAY1B  64 '$'
```

This program calculates an employee's weekly pay and uses a subroutine to calculate overtime earnings. This program compiles and loads correctly:

```
OK, VRPG OVERTIME -DEBUG
F
I
C
O
0000 ERRORS (VRPG - REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD OVERTIME
$ LIBRARY VRPGLB
$ LIBRARY
LOAD COMPLETE
$ QUIT
OK,
```

This program reads data from a disk file named INFO and sends output to a disk file named REPORT. You execute the program then look at the contents of the output contained in the file REPORT:

```
OK, SEG OVERTIME
OK, SLIST REPORT

11/05/83                        WEEKLY EARNINGS

NUMBER      NAME              RATE  HOURS     OT PAY     TOTAL PAY

 49800  Mixon, Richard        4.50     43     $81.00      $261.00
 59080  Young, Lance          5.00     45    $150.00      $350.00
 59990  Kramer, Dave          3.67     44     $88.08      $234.88
 67090  Singer, Alfie         2.99     42     $35.88      $155.48
 77770  Baggins, Moe          6.75     45    $202.50      $472.50
OK,
```

After seeing the output you know that something is wrong. If someone works more than 40 hours, he or she is paid time and a half for those extra hours. Somehow the program is not calculating the overtime pay correctly because it is much more than you would expect it to be. You enter the Debugger and decide to place a breakpoint on the first calculation statement, line 9, then examine the variables RATE and HOURS to see if they are being read properly during the input cycle.

```
OK, DBG OVERTIME

**Dbg**   revision 1.0 - 19.1  (30-November-1983)
> BREAKPOINT 9
> RESTART

**** breakpointed at RPG$MAIN\9
> : HOURS
HOURS = 43
> : RATE
RATE = 4.50
```

Now that you know the variables are being read in properly, you suspect the problem is located within the subroutine. You want to find out why the first employee, Richard Mixon, was given a whopping $81 in overtime pay. You place a breakpoint on line 19 to check the value of OTWAGE, which is the employee's overtime wage at time and a half:

```
> BREAKPOINT 19

**** breakpointed at RPG$MAIN.SUBRU\19
> : OTWAGE
OTWAGE = 27.00
```

To your amazement, you find out Mr. Mixon is making $27 an hour on time and a half, but he makes only $4.50 per hour in regular wages. You know that something is wrong with the formula that calculated OTWAGE. Using the SOURCE command, you start examining the subroutine's calculation statements, one at a time:

```
> SOURCE POINT 15
   15:      CSR         RATE      MULT 40        PAY      62
```

You see there is nothing wrong with line 15, so you examine line 16:

```
> SOURCE NEXT
   16:      CSR         RATE      MULT 5         OTRATE   42
```

There's the problem. The regular hourly rate is being multiplied by 5 and should be multiplied by 0.5. You correct the source file then compile, load, and execute the program again:

```
OK, VRPG OVERTIME -DEBUG
F
I
C
O
0000 ERRORS (VRPG - REV 19.2)
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD OVERTIME
$ LIBRARY VRPGLB
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
OK, SLIST REPORT

11/05/83                        WEEKLY EARNINGS

NUMBER      NAME              RATE  HOURS     OT PAY     TOTAL PAY

 49800  Mixon, Richard        4.50     43     $20.25      $200.25
 59080  Young, Lance          5.00     45     $37.50      $237.50
 59990  Kramer, Dave          3.67     44     $22.00      $168.80
 67090  Singer, Alfie         2.99     42      $8.96      $128.56
 77770  Baggins, Moe          6.75     45     $50.60      $320.60
OK,
```

G

Sample Sessions with C

This appendix offers special considerations specific to Prime C and a sample debugging session with Prime C. (For more information on Prime C, see the C User's Guide.)

SPECIAL CONSIDERATIONS

Assignment

When the Debugger's evaluation command (:) is used with the special C assignment operators (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=), assignment is implicitly performed. This means that expressions are evaluated in the Debugger in exactly the same way they are evaluated in C programs. Ordinarily, the Debugger's LET command must be used to assign values. For example, if X equals 1 the evaluation command can be used with the += operator this way:

```
> : x += 2
X = 3
```

In the example above, the value of 3 is assigned to X.

Attempts to assign a value to an rvalue (for example, an expression enclosed within parentheses) will not cause an error and will appear to be successful. However, this is an illegal operation. The Debugger does not report the error.

C Operators

The only C operator not supported by the Debugger is the CAST operator. All other Prime C operators are supported. Furthermore, these supported operators used to evaluate expressions in the Debugger are functionally identical to the corresponding operators in the Prime C compiler. All expected side effects that occur in C programs also occur when the operators are used from the Debugger. For example, an increment operator (++) that precedes an integer variable will return the value of the variable plus 1 and have the side effect of incrementing that variable by 1. For example:

```
> : I = 1
I = 1
> : ++I
2
> : I
I = 2
```

Special Characters

The C escape character (\) is not supported by the Debugger. Instead, you should use the Debugger's escape character (^). A null character (\0) can be generated by evaluating a null string (""). See the section in Chapter 4 on the escape character for representing a literal newline (\n) and single quote (\').

Defaults for Constants

The default for a floating-point constant is DOUBLE. The default for an integer constant is LONG.

The ?: Construct

The ?: construct is not supported by the Debugger. You should use the IF, ELSE construct instead.

SAMPLE SESSION

Suppose you have written a C program that takes a positive number and returns a power of 2 greater than this number. The program compiles and loads successfully, but watch what happens when you execute it:

```
OK, CC TEST -DEBUG
[CC revision 1.0 - 19.1]
0 Error(s) and 0 Warning(s) detected in 99 source lines.
OK, SEG -LOAD
[SEG rev 19.2]
$ LOAD TEST
$ LIBRARY CCLIB
$ LIBRARY
LOAD COMPLETE
$ EXECUTE
GET POWER OF 2 GREATER THAN ANY POSITIVE NUMBER
TO EXIT THE PROGRAM ENTER 0
ENTER A POSITIVE NUMBER
13
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
16
ENTER A POSITIVE NUMBER
4
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
 4
ENTER A POSITIVE NUMBER
0
OK,
```

Notice that the program correctly gives the power of 2 greater than 13, which is 16. But it also returns, incorrectly, 4 as the power of 2 greater than 4. You enter the Debugger and look at the source program:

```
OK, DBG TEST

**Dbg**   revision 1.0 - 19.1  (30-November-1983)

> SOURCE PRINT 23
    1: #include <stdio.h>
    2: main () {
    3:   int x;
    4:   printf("GET POWER OF 2 GREATER THAN ANY POSITIVE NUMBER\n");
    5:   printf("TO EXIT THE PROGRAM ENTER 0\n");
    6:   printf("ENTER A POSITIVE NUMBER\n");
    7:   scanf("%d",&x);
    8:   while (x != 0) {
    9:     fun(x);
   10:     printf("ENTER A POSITIVE NUMBER\n");
   11:     scanf("%d",&x);
   12:   }
   13: }
   14:   fun(x)
   15:   int x; {
   16:   int i;
   17:     for (i=1; i<x; i<<=1);
   18:   printf("THE POWER OF 2 GREATER THAN THIS NUMBER IS...\n");
   19:   printf("%2d\n", i);
   20: }
BOTTOM
>
```

You make sure your input is read correctly, so you place a breakpoint on source line 8, restart the program, and check the value of variable X:

```
> BREAKPOINT 8
> RESTART
GET POWER OF 2 GREATER THAN ANY POSITIVE NUMBER
TO EXIT THE PROGRAM ENTER 0
ENTER A POSITIVE NUMBER
4

**** breakpointed at MAIN\8
> : X
X = 4
>
```

Now that you know the value of X has been assigned correctly, you suspect the problem lies in the function FUN. You step into this function at source line 9, then make sure the argument for the function has been successfully passed:

```
> BREAKPOINT 9
> CONTINUE

**** breakpointed at MAIN\9
> STEPIN

**** "in" completion at FUN\17
> ARGUMENTS
X = 4
>
```

The argument X for function FUN has been passed correctly. Therefore, you decide to trace the value of I, because it is the only variable whose value changes in this function:

```
> WATCH I
> CONTINUE
The value of FUN\I has been changed at FUN\17
  from 1
  to   2
The value of FUN\I has been changed at FUN\17
  from 2
  to   4
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
 4
ENTER A POSITIVE NUMBER

**** breakpointed at MAIN\8
> QUIT
OK,
```

You now notice that variable I fails to loop the appropriate number of times. Your loop should stop as soon as I's value is greater than or equal to X's value; and, in this case, I's value is equal to X's value. Instead of:

```
I < X
```

your final loop value for I should read:

```
I <= X
```

After you correct the mistake, you run the program again, and it executes successfully:

```
OK, SEG TEST
GET POWER OF 2 GREATER THAN ANY POSITIVE NUMBER
TO EXIT THE PROGRAM ENTER 0
ENTER A POSITIVE NUMBER
33
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
64
ENTER A POSITIVE NUMBER
12
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
16
ENTER A POSITIVE NUMBER
4
THE POWER OF 2 GREATER THAN THIS NUMBER IS...
 8
ENTER A POSITIVE NUMBER
0
OK,
```

H
Special Considerations

This appendix lists special considerations and restrictions to Debugger usage. These apply to all languages. For information on language-specific considerations and restrictions, see the appendix on sample debugging sessions with your particular language.

Special considerations and restrictions follow:

- If your program is inputting data from a PRIMOS data file and you have suspended execution, you cannot use RESTART to rerun the program unless you close the input file by entering the Debugger's ! command followed by the PRIMOS CLOSE command and the name of the input file you want to close. This applies to output files as well.

- When debugging a program that closes file units indiscriminately (that is, with CLOSE ALL), the -FULL_INIT option should be specified on the DBG command line. If quick initialization is used, the CLOSE ALL command should not be given from the Debugger.

- If the program being debugged creates an on-unit for the system condition ILLEGAL_INST$ or ANY$, this on-unit will be invoked whenever any breakpoint is encountered. Hence, if such on-units are used, the following Debugger commands should not be used: BREAKPOINT, TRACEPOINT, STEP, STEPIN, STRACE, and VTRACE.

- If the program being debugged uses specific segments in the range 4001 through 4037 for some purpose (for example, for temporary storage) without allocating those segments in SEG, the Debugger may overwrite them for its own storage. To avoid this problem, you may tell the Debugger which segments you are using with the A/SYMBOL command in SEG. For example, the SEG command A/SYMBOL TEMP1 177777 tells the Debugger that your program is using segment 4027.

- If execution is suspended at an entry to a program block, you should not enter a Debugger GOTO command. It may skip over code between the block entry and the first executable statement, which sets up information used later in the block.

I

Strategies in Debugging

STRATEGIES IN DEBUGGING

This appendix suggests some debugging strategies. These strategies are just guidelines for debugging. They describe the symptoms of some of the more common programming errors and suggest ways of zeroing in on the causes. Debugging strategies are for the most part personal skills acquired only through experience and a certain amount of frustration.

It is assumed that you have a good idea of what your program is supposed to do and the algorithms and data structures that your program uses.

It is assumed that your program has been compiled and loaded with no errors. You should not attempt to execute a program which has not been compiled or loaded successfully. When you execute your program, it will fit into one of the following four Cases:

1. It will run until completion and produce correct results.

2. It will run until completion and produce incorrect results or no results at all.

3. It will terminate abnormally, printing an error message and returning to PRIMOS command level.

4. It will not terminate.

Program Completes Successfully

If Case 1 applies, it is assumed you wish to use the Debugger to look at the final state of the static variables declared within the program. First execute the program using the Debugger. When a message appears indicating that program execution has completed, the user may inspect program-defined data using the : (evaluation) command. This message will be "program execution complete" if the main program returns, "program exit from (statement-id)" if a program block calls EXIT, or "program stop at (statement-id)" if a STOP statement is executed.

Program Completes with Incorrect Results

If Case 2 applies, you might attempt to locate the problem by tracing program execution and data computation.

If it is obvious that the error lies within one or a few statements, setting breakpoints and monitoring the values of variables will probably identify the source of the problem quite readily.

If, on the other hand, you don't know where to begin looking for the problem, set breakpoints and tracepoints sparsely throughout your program in an attempt to find the first indication that things are awry. When this occurs, the problem area is more densely populated with breakpoints until you finally zero in on the problem. Specifically, the method is:

1. Determine by deduction the section of the program in which the problem exists. If you are unable to localize the problem, the entire program is considered suspect.

2. Choose no more than eight key locations within the suspected code and place a breakpoint or tracepoint at each. Key locations are statements, entries, and exits at which you may detect correct or incorrect program operation by:

    - Inspecting the values of variables
    - Simply knowing that program control has arrived there

3. When the program encounters or fails to encounter a key location and you detect something amiss, the size and location of the suspected code can be modified appropriately. Steps 2 and 3 may be repeated until the suspected code consists of only a few statements, at which time you may use the simpler method described above.

Program Terminates Abnormally

If the program terminates abnormally, you may reexecute the program using the Debugger. When the error condition that caused termination is raised, the Debugger regains control and returns to command level. Through the use of the TRACEBACK command, you may determine the program location at which the error occurred.

If you are unable to determine the cause of the error by looking at the source code for the statement, the condition name may provide some additional information.

If the error is the result of an access violation (condition ACCESS_VIOLATION$), illegal segment reference (ILLEGAL_SEGNO$), pointer fault (POINTER_FAULT$), or illegal page reference (OUT_OF_BOUNDS$), the program was probably attempting to:

- Reference through a pointer, label, or entry variable that contains garbage (has not been initialized).
- Reference a subroutine argument that has not been supplied by the caller.
- Call a procedure that has not been loaded.
- Reference an external symbol that has not been defined.

An error of the type described above may also be the indirect result of references to out-of-bounds array elements. (If this is suspect and the compiler has the capability to generate code to check for out-of-bounds array references, you should use it.)

If the error resulted from an illegal or restricted instruction (conditions ILLEGAL_INST$ and RESTRICTED_INST$), your program either intentionally or inadvertently executed an instruction which was undefined (illegal) or unable to be executed in rings 1 or 3 (restricted). Most likely, you have overwritten a location in your program via an out-of-bounds array reference or garbaged pointer, or have caused control to be transferred to an incorrect location. The latter may be done in a variety of ways. One way is "going to" an uninitialized label variable in PL/I-G or supplying an illegal alternate return in a FORTRAN subroutine call. If you are confronted with this error, a procedure call/return stack trace using the TRACEBACK command is usually a good place to start.

If the error occurs as the result of a null pointer (condition NULL_POINTER$), the explicit or implied pointer reference in the faulting statement was attempted through a pointer whose value was, in the PL/I-G sense, null. (The segment number was '7777.)

If the error was the result of a linkage fault (condition LINKAGE_FAULT$), your program attempted to execute a procedure call to a routine that was not defined in the operating system or in a shared library. Check with your System Administrator as to the whereabouts of the missing routine.

An arithmetic exception error (condition ARITH$) will occur when the CPU detects an illegal arithmetic operation, such as an exponent overflow or a divide-by-zero. An explanatory message is displayed by the Debugger, indicating the cause of the condition and the action taken by the Debugger (if any) to allow continuation of execution. Find the source statement and inspect the values of the variables, which comprise the expression (if necessary) to determine the cause.

A bad nonlocal GOTO error (condition BAD_NONLOCAL_GOTO$) indicates that you attempted to go to a label variable which contained an illegal display (stackframe) pointer. Find the source statement and, using the Debugger evaluation command, examine the label variable. You might choose to use the value tracing to identify the locations within the program that the variable is modified.

Program Fails to Terminate

Case 4 applies if, after a reasonable period of time, the program fails to terminate. This case can be called the "infinite loop."

As is done with the other cases described above, you should reexecute the program under control of the Debugger. After allowing it to execute a sufficient amount of time (presumably, to permit it to enter the infinite loop), you may type the QUIT, BREAK, or CONTROL-P key to cause control to return to the Debugger.

By noting the value of the execution environment pointer, optionally performing a traceback, continuing execution, and quitting again, you should begin to get some idea of the area in which the loop exists. This last step should be repeated as many times as necessary.

The area of the loop may be further refined by setting tracepoints and breakpoints at key locations within the suspected code and continuing execution.

J

Summary of Debugger Commands

This appendix summarizes all 53 Debugger commands in alphabetical order. Each command line is followed by a brief explanation of the command's function, the chapters in which it is discussed, and brief explanations of command line syntax elements, such as arguments and options. Command abbreviations are underlined.

▶ :[language-name[,print-mode] / print-mode] expression

The : (evaluation) command evaluates a variable or an expression (Chapters 3 and 6).

The expression is the variable or expression that you wish to evaluate. The optional language-name is the language of evaluation in which you want your expression evaluated. The print-mode is the print mode in which you want your expression evaluated. The print mode can be ASCII, BIT, DECIMAL, FLOAT, HEX, or OCTAL.

▶ ! primos-command-line

The ! command executes certain PRIMOS commands from Debugger command level (Chapter 11).

The primos-command-line is one or more PRIMOS commands that you want to execute from the Debugger. You must use internal PRIMOS commands, not external commands.

▶ * [value]

The * command executes the current command line a specific number of times or forever (Chapter 11).

The value is the optional number of times you want the command line to be repeated.

▶ ACTIONLIST { SUPPRESS | PRINT }

ACTIONLIST displays the commands in a breakpoint action list or in a macro command list (Chapters 5 and 9).

The PRINT option specifies all lists to be displayed. The SUPPRESS option deactivates the PRINT option, causing no lists to be displayed.

▶ AGAIN

AGAIN repeats the Debugger command line that has just been executed (Chapter 11).

▶ ARGUMENTS program-block-name [\activation-number]

ARGUMENTS displays the values of the arguments passed to the program block defined by the evaluation environment pointer (Chapter 6).

The program-block-name is the name of the program block whose arguments you want to display. The activation-number is a particular activation of a specified program block.

▶ BREAKPOINT [breakpoint-identifier] [action-list]
[-AFTER value] [-BEFORE value]
[-EVERY value] [-COUNT value] [-EDIT]

[-IGNORE | -NIGNORE]

BREAKPOINT suspends the execution of your program (Chapters 3 and 5).

The breakpoint-identifier identifies the place where you want to suspend execution, which can be an executable statement, statement label, or entry to or exit from a program block. The action-list is an optional list of Debugger commands that execute whenever a breakpoint trap occurs.

The -AFTER option causes the breakpoint trap to occur only when the value of the breakpoint counter exceeds the value of the specified value following -AFTER. The -BEFORE option causes the breakpoint trap to occur only when the value of the breakpoint counter is less than the value following -BEFORE. The -EVERY option causes the breakpoint trap to occur every n iterations through the breakpoint location, where n is the value following -EVERY. The -COUNT option can be used to set the breakpoint counter.

The -IGNORE option sets the ignore flag, causing the breakpoint trap to be suppressed and never taken. The -NIGNORE option deactivates the ignore flag so that the breakpoint trap is taken again. The -EDIT option invokes the Debugger's command line editor so that you may modify a breakpoint command line.

Entry/exit breakpoints are identified by one of the following three formats:

- program-block-name\\breakpoint-type
- \breakpoint-type
- program-block-name\

The breakpoint-type can be either ENTRY or EXIT. The program-block-name is the name of the program block at which you want to break.

▶ CALL variable [(argument-list)]

CALL allows you to call a program block from Debugger command level (Chapter 7).

The variable is the name of the program block you want to call. The argument-list is a list of expressions or "parameters" that are supplied or "passed" to the program block according to the rules of the host language.

▶ CLEAR [breakpoint-identifier]

CLEAR deletes a breakpoint or a tracepoint (Chapters 5 and 8).

The breakpoint-identifier must be any valid breakpoint or tracepoint identifier, such as a source line number or statement label. Used by itself, with no breakpoint identifier, CLEAR deletes the breakpoint or tracepoint specified by the execution environment pointer.

▶ CLEARALL [program-block-name [-DESCEND]] [-BREAKPOINTS | -TRACEPOINTS]

CLEARALL deletes either all breakpoints and tracepoints in the debugging environment or all breakpoints and tracepoints in a specific program block (Chapters 5 and 8).

The program-block-name is the name of the program block containing the breakpoints and/or tracepoints that you want to delete. The -BREAKPOINTS option causes only breakpoints to be deleted. The -TRACEPOINTS option causes only tracepoints to be deleted. The -DESCEND option deletes all breakpoints and tracepoints in a specified program block and in all the nested program blocks or "descendants" contained in the specified block. Used without any arguments, CLEARALL deletes all breakpoints and tracepoints in the debugging environment.

▶ CMDLINE

CMDLINE allows you to enter your program's command line arguments (Chapters 3 and 13).

▶ CONTINUE

CONTINUE resumes program execution following a breakpoint, a single-step operation, or an error condition. Program execution resumes at the location specified by the execution environment pointer (Chapters 3 and 5).

▶ ENVIRONMENT [program-block-name [\activation-number] | -POP]

ENVIRONMENT changes the evaluation environment, which is the program block that the Debugger considers current (Chapter 6).

The program-block-name is the name of the program block that you want as the new evaluation environment. The activation-number specifies a particular activation of program-block-name. The -POP option removes

or "pops" an environment from the evaluation environment stack. Used by itself, with no argument, ENVIRONMENT displays the name of the current evaluation environment.

▶ ENVLIST

ENVLIST displays the current evaluation environment and the contents of the evaluation environment stack (Chapter 6).

▶ ETRACE { ON / ARGS / OFF }

ETRACE displays a trace message each time a program block is called or returns. This is known as entry tracing (Chapter 8).

The ON option displays a trace message when each program block is called and returned. The ARGS option displays trace messages at the entry and exits to called program blocks and displays the values of arguments passed to each called block at each entry (but not each exit). The OFF option turns off entry tracing.

▶ GOTO [program-block-name\[activation-number\]]statement-identifier

GOTO moves the location of the execution environment pointer to another statement in your program (Chapter 5).

The program-block-name is the name of the active program block that contains the statement to which you want to transfer control. The statement-identifier is the statement to which you want to transfer control. It can be a source line number, statement label, or any other valid identifier as defined in Chapter 4. The activation-number specifies that control is transferred to a statement in a particular activation of a program block.

▶ HELP [-LIST / -SYM_LIST / command-name / syntax-symbol]

HELP can help you remember and understand Debugger commands and features by displaying information about those commands and features (Chapter 3).

The command-name is the name of any Debugger command about which you want command line syntax information. The syntax-symbol is any symbol that is used in command syntax descriptions. The -LIST option lists all Debugger commands in alphabetical order. The -SYM_LIST option lists all Debugger syntax symbols used in Debugger command line syntax.

▶ IF expression action-list [ELSE action-list]

IF executes a breakpoint action list, or any Debugger command conditionally, contingent upon the result of an expression (Chapter 5).

The expression is any valid expression in the host language. The expression can be either true or false. If the expression is true, the first action-list immediately following the expression is executed, and the ELSE clause, if present, is ignored. If the expression is false, the first action list is ignored, but the ELSE action list, if present, is executed.

▶ IN

IN continues program execution until the next program block is called and suspends execution inside that block immediately before the first executable statement (Chapter 7).

▶ INFO { program-block-name\ / statement-identifier }

INFO displays information about a program block or statement (Chapter 13).

The program-block-name is the name of the program block about which you want information. The statement-identifier is the executable statement about which you want information.

▶ LANGUAGE [FORTRAN / F77 / PL1G / PASCAL / COBOL / RPG / C]

LANGUAGE changes the language of evaluation to evaluate expressions (Chapter 6).

Used without an argument, LANGUAGE displays the name of the current host language. If you want to change the current language to another language, enter one of the language names shown in the command format.

▶ LET variable = expression

LET allows you to assign a new value to any variable defined by the program (Chapters 3 and 6).

The variable is a user variable name, as defined in Chapter 4. The expression is any expression permitted by the host language whose resultant value is convertible to the data type of the variable.

▶ LIST [breakpoint-identifier]

LIST displays the attributes of one breakpoint or one tracepoint (Chapters 5 and 8).

The breakpoint-identifier is the breakpoint or tracepoint that you want to display. Used without the breakpoint identifier, LIST displays the attributes for the breakpoint or tracepoint defined by the execution environment pointer.

▶ LISTALL [program-block-name [-DESCEND]] [-BREAKPOINTS | -TRACEPOINTS]

LISTALL lists the attributes of all breakpoints and tracepoints (Chapters 5 and 8).

The program-block-name is the name of the program block that contains the breakpoints and tracepoints you want to display. The -BREAKPOINTS option displays only breakpoints. The -TRACEPOINTS option displays only tracepoints. The -DESCEND option displays all breakpoints and tracepoints for a specified block and for all nested program blocks or "descendants" contained in the specified block. If LISTALL is used without arguments, a list of all breakpoint and tracepoint attributes is displayed.

▶ LOADSTATE filename

LOADSTATE restores to your debugging session all the breakpoints, tracepoints, and macros you have saved with SAVESTATE (Chapter 10).

The filename is the name of the PRIMOS file that contains the breakpoints, tracepoints, and macros you want to use. (These files are created with the Debugger's SAVESTATE command.)

```
                    ┌ macro-name ┌ command-list ┐                    ┐
                    │            │ -DELETE      │                    │
                    │            └ -EDIT        ┘                    │
▶ MACRO             │                                                │
                    │ -CHANGE_NAME old-macro-name new-macro-name     │
                    │ -ON                                            │
                    └ -OFF                                           ┘
```

MACRO allows you to create new commands, known as macros, that can be used in place of one or more Debugger commands (Chapters 9 and 12).

The macro-name is the name of the macro that you want to create. The command-list is the list of one or more Debugger commands that you want your macro name to stand for.

The -DELETE option deletes a specified macro. The -EDIT option invokes the Debugger command line editor so that you can modify the macro specified by macro-name. The -CHANGE_NAME option changes the name of a macro from old-macro-name to new-macro-name. The -OFF option turns off the use of macros without destroying your current macros. The -ON option enables the use of macros once again.

▶ MACROLIST [macro-name]

MACROLIST displays one or all of your currently defined macros and their command lists (Chapters 9 and 12).

The macro-name is the name of a specific macro that you want to display. Used by itself, with no macro name, all macros in the macro list and their corresponding command lists are displayed.

▶ MAIN [program-block-name]

MAIN tells the Debugger what the main program block should be. The main program is the program block that the Debugger calls when a RESTART command is entered (Chapter 5).

The program-block-name is the name of the program block that you want the Debugger to call when a RESTART command is entered. Used by itself, with no program block name, MAIN displays the name of the main program that the Debugger currently recognizes.

▶ OUT

OUT continues program execution until the current block, defined by the execution environment pointer, returns and execution suspends at the exit of that block (Chapter 7).

▶ PAUSE

PAUSE temporarily suspends your Debugging session and returns you to PRIMOS command level. You must enter only internal PRIMOS commands with PAUSE, not external commands (Chapter 11).

▶ PMODE print-mode variable-1 [,variable-2 ...]

PMODE sets the print mode explicitly so that the specified print mode will be used in your debugging session whenever a variable is evaluated (Chapter 13).

The print-mode is the print mode you want to specify. It can be ASCII, BIT, DECIMAL, FLOAT, HEX, OCTAL, or DEFAULT. variable-1, variable-2, etc., are the variables you want evaluated with print-mode.

▶ PSYMBOL

PSYMBOL displays a list containing the names and current character values of special symbols the Debugger recognizes (Chapter 4).

▶ QUIT

QUIT causes the debugging session to end and returns you to PRIMOS command level (Chapter 3).

▶ RESTART [step-command]

RESTART starts and restarts program execution from within the Debugger (Chapters 3 and 5).

The step-command is an optional Debugger single-stepping command.

▶ RESUBMIT

RESUBMIT invokes the Debugger command line editor so that you can modify the most recent command line entered (Chapter 10).

▶ SAVESTATE filename [-MACROS] [-BREAKPOINTS] [-TRACEPOINTS]

SAVESTATE saves your breakpoints, tracepoints, and macros and places them into a PRIMOS file in your directory for future use (Chapter 10).

The filename is the pathname of the PRIMOS file where you want your breakpoints, tracepoints, and macros placed. If you do not specify a pathname, the file will be placed in the directory to which you are attached.

The -MACROS option causes only your macros to be placed into the file specified by filename. The -BREAKPOINTS option causes only your breakpoints and their action lists to be placed into the file. The -TRACEPOINTS option causes only your tracepoints to be placed into the file.

If you specify only a filename without an option, then all of your breakpoints, tracepoints, and macros are placed into the file specified by filename.

▶ SEGMENTS

SEGMENTS displays a list of segments in memory that are currently in use (Chapter 13).

▶ SOURCE source-command [argument]

SOURCE allows you to examine your source file while debugging (Chapters 3 and 11).

The source-command is any EDITOR subcommand that can be used with SOURCE. The argument is an EDITOR source subcommand object such as a line number or text string.

▶ STATUS

STATUS displays various useful information about the state of your debugging environment (Chapter 13).

▶ STEP [value]

STEP executes one or more statements at a time and steps across calls to program blocks (Chapter 7).

The value is the number of statements you want to execute before suspending execution. If no value is specified, one statement is executed by default.

▶ STEPIN [value]

STEPIN executes one or more statements at a time and steps into program blocks that are called (Chapter 7).

The value is the number of statements you want to execute before suspending execution. If no value is specified, one statement is executed by default.

▶ STRACE { FULL / QUIET / OFF }

STRACE allows you to display a trace message prior to the execution of every statement in your program. This feature is known as statement tracing (Chapter 8).

The FULL option displays a trace message prior to the execution of every statement in your program. The QUIET option displays a trace message only prior to the execution of each labelled statement. The OFF option turns off statement tracing.

▶ SYMBOL symbol-name character-value

SYMBOL changes the value of a special symbol that is recognized by the Debugger (Chapter 4).

The symbol-name is the name of the character symbol you want to change. The character-value is the new character value of the symbol.

▶ TRACEBACK [-FRAMES value [-LEAST_RECENT]] [-FROM value] [-TO value]
[-REVERSE] [-DBG] [-ONUnits] [-ADDRESSES]

TRACEBACK allows you to look at the contents of the call/return stack, which is a list of currently active program blocks in your program execution (Chapter 8).

The value is a positive non-zero integer.

Used by itself, with no option, TRACEBACK displays the contents of the call/return stack from the most recent frame to the least recent frame.

The -FRAMES option displays only the number of frames that are limited to the specified value. The -FROM option starts displaying frames beginning with the number represented by value. The -TO option specifies that the last frame displayed is the frame represented by value. The -REVERSE option displays the frames in reverse order. The -DBG option displays all of the Debugger-owned frames. The -ONUNITS option displays names of all on-units and their corresponding program blocks. The -ADDRESSES option displays internal address information.

▶ TRACEPOINT [breakpoint-identifier] [-AFTER value]
[-BEFORE value] [-EVERY value] [-COUNT value]

[-IGNORE
-NIGNORE]

TRACEPOINT allows you to display a trace message each time a statement, label, or entry/exit to a program block is encountered (Chapter 8).

The breakpoint-identifier is the statement, label, or entry/exit where you want to display a trace message.

The -AFTER, -BEFORE, -EVERY, -COUNT, -IGNORE, and -NIGNORE options work the way they do for breakpoints. (For an explanation of these options, see the discussion under the BREAKPOINT command in this appendix, and see Chapter 5.)

▶ TYPE expression

TYPE displays the data type and other attributes of a variable or expression (Chapters 3 and 6).

The expression is any expression permitted by the host language.

▶ UNWATCH { variable-1 [, variable-2 ...]
-ALL }

UNWATCH removes one or more variables from the watch list, which was created during value tracing with the WATCH command (Chapter 8).

variable-1, variable-2, etc., are the variables you want to remove from the watch list. The -ALL option removes all variables from the watch list.

▶ UNWIND

UNWIND erases the call/return stack and causes the execution environment pointer to become undefined (Chapter 8).

▶ VPSD

VPSD invokes the 64V mode Prime Symbolic Debugger (VPSD), which is one of Prime's machine-level debuggers (Chapter 13).

▶ VTRACE { FULL / ENTRY_EXIT / OFF }

VTRACE can trace values at the entry or exit of a program block and turn off value tracing (Chapter 8).

The ENTRY_EXIT option enables value tracing on only the entries to and exits from program blocks. The OFF option causes value tracing to not occur, although the contents of the watch list are undisturbed. The FULL option enables value tracing at every statement once again.

▶ WATCH variable-1 [,variable-2 ...]

WATCH displays a message whenever the value of one or more variables changes during program execution. This feature is known as value tracing (Chapters 3 and 8).

variable-1, variable-2, etc., are the variables whose values you want to trace. The variables that you trace are placed onto an internal Debugger table known as the watch list.

▶ WATCHLIST

WATCHLIST displays the names of variables currently in the watch list (Chapter 8).

▶ WHERE [segment-number/address]

WHERE displays the location of the execution environment pointer (Chapter 5).

You can find the program location that corresponds to a given segment and halfword memory address by specifying the segment-number, which is a segment number represented in octal, and the address, which is a halfword address represented in octal.

Used by itself, with no argument, WHERE displays the current location of the execution environment pointer.

K

Glossary of Terms

This appendix is a glossary of terms that are related to the Debugger and its operations. The terms appear in alphabetical order with accompanying definitions.

This glossary is not intended to provide descriptions of syntax symbols that appear on Debugger command lines. Rather, it provides brief definitions of Debugger-related terms that are used freely throughout the book. For descriptions of Debugger command syntax symbols, see Appendix J, the HELP command on-line descriptions, or the descriptions given in the in-depth discussions throughout the book.

- absolute activation number

An unsigned integer constant specifying the actual number of an activation — second, third, fourth, etc.

- action list

A list of Debugger commands, attached to a breakpoint, that execute when the breakpoint trap occurs.

- action list depth counter

A number enclosed in angle brackets that appears at the left margin, immediately preceding an action list, when an action list is displayed with the ACTIONLIST command. This counter specifies the nesting level of the action list.

- activation

A particular execution of a program block.

- activation number

An integer designating a particular activation of a program block.

- active

A program block that has been called but has not yet returned.

- address

A 16-bit (halfword) location in memory. Prime memory is addressable in 16-bit offsets.

- binary file

Same as object file (See object file.)

- block

Same as program block. (See program block.)

- breaking

The process of executing a breakpoint.

- breakpoint

A suspension of program execution specified by the BREAKPOINT command.

- breakpoint counter

A counter that keeps track of the number of times a breakpoint has been encountered during program execution.

- breakpoint ignore flag

A flag, set by the BREAKPOINT command's -IGNORE option, that suppresses a breakpoint so that the breakpoint is never taken.

- bug

A problem that prevents the successful execution of a program.

- built-in function

A language-defined function that can be used to evaluate expressions. The Debugger supplies standard Pascal, built-in PL/I-G and C, and intrinsic FORTRAN functions for the user.

- CALL frame

A frame on the call/return stack that is created by a CALL command invocation of the Debugger.

- call/return stack

An internal stack that contains a list of active program blocks during program execution.

- command line editor

The Debugger's line editor that edits the most recent command line entered or any breakpoint action list or macro command list.

- command list

A list of Debugger commands that usually refers to a macro command list, which appears inside square brackets following a macro name when a macro is created. (This is synonymous with action list.)

- compile time

The time or moment at which a program is compiling.

- conditional action list

An action list that executes contingent upon the result of an expression designated by the IF command.

- conditional breakpoint

A breakpoint that is taken or not taken, depending on the conditions that are specified by the BREAKPOINT command's -AFTER, -BEFORE, and -EVERY options.

- condition frame

A frame on the call/return stack that represents a a call to an error condition.

- current evaluation environment

The environment, or block, defined by the evaluation environment pointer. Synonymous with current program block.

- current program block

The block, or environment, defined by the evaluation environment pointer. Synonymous with current evaluation environment.

- data manipulation

Debugger features that allow the user to examine, evaluate, and modify variables and expressions.

- DBG

The PRIMOS command that invokes the Debugger.

- debug mode

The mode that a compiled module is in when the -DEBUG option has been activated. All Debugger features are available in debug mode.

- Debugger control

Debugger features that allow the user to make the Debugger interpret or treat information in a particular way.

- Debugger-defined variables

Three variables ($MR, $COUNT, and $COUNTERS) that are created by and always known to the Debugger. These variables may be referenced by the user.

- Debugger-owned frame

A frame on the call/return stack that represents a call by the Debugger that returns to the Debugger.

- default

A state of being, action, or lack of action that takes place automatically when no alternative is specified.

- ECB

The entry control block for an entry to an object routine.

- entry

A position in program execution at the entry to a program block — immediately after the call to the block, prior to the execution of the first executable statement.

- entry breakpoint

A breakpoint that occurs at the entry of a program block.

- entry tracing

The process of displaying a trace message each time a program block is called or returned.

- environment

A location that the Debugger recognizes as the current area in which an expression is to be evaluated (evaluation environment) or execution is to resume (execution environment).

- erase character

A special character that erases the previous character typed. The system default is the double quote (").

- escape character

A circumflex or up-arrow character (^) that affects the meaning of the character or characters that immediately follow it.

- escape sequence

A sequence of characters beginning with the escape character and ending with the last character to be affected by the escape character.

- evaluation environment

A program block that the Debugger considers current and uses to identify statements and evaluate variables and expressions. The program block in which the user is debugging is the default.

- evaluation environment pointer

A pointer that gives the default (current) evaluation environment (program block) to be used for finding variables and statements and for examining source files.

- evaluation environment stack

An internal stack that contains evaluation environments recently selected using the ENVIRONMENT command.

- executable file

Same as runfile. (See runfile.)

- executable statement

A statement in a source program that performs some action. The Debugger can suspend execution only at executable statements.

- execution environment

A location that the Debugger recognizes as the current area in which execution is to resume.

- execution environment pointer

A pointer that gives the location at which execution resumes when a CONTINUE or single-step command is given.

- exit

A position in program execution at the exit from a program block — outside the block, immediately after the block has returned.

- exit breakpoint

A breakpoint that occurs at the exit from a called program block.

- fault frame

A frame on the call/return stack that represents a hardware fault.

- frame

A representation of a program block call that is placed on the call/return stack.

- host language

The language of the program block defined by the current evaluation environment.

- information request

Debugger features that allow the user to request special information.

- insert line

A physical line number in a $INSERT or %INCLUDE file.

- interactive

Dialog between the Debugger and the user that takes place at the source code level.

- kill character

A character that causes a line typed thus far to be ignored. The system default is the question mark (?).

- label

Same as statement label. (See statement label.)

- language of evaluation

The high-level language whose syntax rules the Debugger uses at any given time to evaluate expressions.

- line offset

The number of physical source lines following the line containing a statement label.

- macro list

An internal Debugger table that contains macros created with the MACRO command.

- main program

The program block that is loaded first during the loading process and the program block that the Debugger calls when a RESTART command is entered.

- multilingual

A Debugger capability that describes the way the Debugger understands source code syntax rules in all supported Prime high-level languages.

- nested action list

An action list contained within another action list.

- nodebug mode

The mode that a compiled module is in when the -NODEBUG option has been activated or when no debugging options have been specified. (No Debugger features are available in nodebug mode.)

- object (binary) file

A file containing one or more binary modules created by compilation.

- option

A part of a Debugger or PRIMOS command line that can be specified to perform or not perform a particular function that is related to the command's function.

- owner block

The program block that is represented by a particular frame on the call/return stack.

- pathname

The complete name of a PRIMOS file, which may include the name of the Master File Directory (MFD), the User File Directory (UFD), and one or more sub-UFD names, as well as the filename.

- pop

A removal of an evaluation environment off the top of the evaluation environment stack using the ENVIRONMENT command's -POP option.

- print mode

A mode that specifies the format in which the result of a variable or expression evaluation is printed. Print modes include ASCII, BIT, DECIMAL, FLOAT, HEX, and OCTAL.

- production mode

The mode that a compiled module is in when the -PRODUCTION option has been activated. (Debugger features that do not involve statements are available in production mode.)

- program block

A universal language-independent definition of a main program, procedure, function, subroutine, BEGIN block, or any other program unit in any of Prime's languages. (Chapter 4 defines program blocks for each language.)

- program control

A type of Debugger feature that allows the user to manipulate the execution of a program.

- prompt

A symbol that appears on the terminal and waits for command input from the user.

- recursion

The process by which a program block calls itself or causes itself to be called.

- relative activation number

A number that specifies the number of activations to count backwards from, beginning at the most recent activation of the specified program block. This number is specified by a minus sign (-) immediately followed by an integer constant.

- runfile

An executable version of a program — a SEG file containing a program that is ready to be executed.

- runtime

The time or moment at which a program is executing.

- SAVESTATE file

A PRIMOS file, created by the SAVESTATE command, that contains saved breakpoints, tracepoints, and macros.

- SEG

Prime's segmented loading utility for V-mode and I-mode files.

- segment

A block of address space consisting of 131,072 bytes.

- separator character

A character that separates one command line from another. The default is a semicolon.

- source file

A file containing programming (compilable) source code in the format defined by any of Prime's high-level languages.

- source line number

The physical line number in the source file.

- special character

A certain character, which the user can enter at the terminal, that has a special meaning to the Debugger. These characters either cause special actions or are interpreted as part of special command syntax.

- special symbol

A special character that can be examined and changed with the PSYMBOL and SYMBOL commands.

- stack

A dynamic work area for storing addresses and other data values. It is called dynamic because it is assigned when a program block is called and released upon return from the block.

- statement label

A statement label number or label constant, RPG tag, or COBOL paragraph name or section name.

- statement offset

The number of statements to count from the first statement on a multistatement line. The first statement on a line has a statement offset of 0, the second has an offset of 1, and so on.

- statement tracing

The process of displaying a trace message prior to the execution of every statement in a program.

- step counter

An invisible counter that contains the number of statements left to be executed before execution is suspended and control returns to Debugger command level.

- subcommand

Any non-Debugger or non-PRIMOS command that is used with or after a Debugger command, such as a SOURCE subcommand or command line EDITOR subcommand.

- suffix conventions

A method for naming source files, object files, and runfiles, in which particular suffixes are attached to these files during the process of compiling, loading, and executing.

- symbol table

An internal table of data objects, which includes the name, location, and attribute of each source program variable, and a statement map that contains the location of compiled code corresponding to each source language statement. The Debugger uses this table to relate the object (binary) code to the high-level language source code.

- tracepoint

A message that is displayed on the terminal, via the TRACEPOINT command, each time a selected statement,entry, or exit is encountered.

- tracing

Debugger features that allow the user to trace the progress of program execution from beginning to end.

- trap

Same as a breakpoint. (See breakpoint.)

- user-owned frame

A frame on the call return stack that represents a user program block call.

- V-mode

The addressing mode normally generated by Prime's compilers. Programs in V-mode can take full advantage of virtual address space and of the V-mode instruction set.

- value tracing

The process of keeping track of or "watching" the values of variables change through the execution of a program.

L

Commands Listed by Chapter

On the following page, this appendix provides a quick reference of commands, listed chapter by chapter. For convenience, you may want to remove the page and tape it to your office wall, or to the front or back cover of this book.

List of Commands by Chapter

Chapter 3

DBG
RESTART
CMDLINE
SOURCE
BREAKPOINT
CONTINUE
:
TYPE
LET
WATCH
HELP
QUIT

Chapter 4

PSYMBOL
SYMBOL

Chapter 5

RESTART
CONTINUE
BREAKPOINT
IF
ACTIONLIST
LIST
LISTALL
CLEAR
CLEARALL
WHERE
GOTO
MAIN

Chapter 6

:
TYPE
LET
ARGUMENTS
ENVIRONMENT
ENVLIST
LANGUAGE

Chapter 7

STEP
STEPIN
IN
OUT
CALL

Chapter 8

TRACEPOINT
LIST
LISTALL
CLEAR
CLEARALL
WATCH
WATCHLIST
UNWATCH
VTRACE
ETRACE
STRACE
TRACEBACK
UNWIND

Chapter 9

MACRO
MACROLIST
ACTIONLIST

Chapter 10

RESUBMIT
BREAKPOINT -EDIT
MACRO -EDIT
SAVESTATE
LOADSTATE

Chapter 11

SOURCE EX
SOURCE NAME
SOURCE RENAME
!
PAUSE
*
AGAIN

Chapter 12

MACRO
MACROLIST

Chapter 13

DBG
CMDLINE
INFO
SEGMENTS
STATUS
PMODE
VPSD

INDEX

Index

F

S

W

SURVEY

READER RESPONSE FORM

DOC4033-193 Source Level Debugger User's Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760