CodeWatch*
Reference Manual

INTERACTIVE

product family

INTERACTIVE

A Kodak Company

CodeWatch*
Reference Manual

This manual provides specific information for using LPI's source-level debugger.

Contents

Chapter 2: Using CodeWatch (Cont.)

Chapter 3: CodeWatch Functionality

Chapter 3: CodeWatch Functionality (Cont.)

Chapter 4: CodeWatch Commands

Chapter 4: CodeWatch Commands (Cont.)

Chapter 5: Debugging LPI-BASIC Programs

Chapter 5: Debugging LPI-BASIC Programs (Cont.)

Chapter 6: Debugging LPI-C Programs

Chapter 7: Debugging LPI-COBOL Programs

Chapter 8: Debugging LPI-FORTRAN Programs

Chapter 9: Debugging LPI-PASCAL Programs

Chapter 10: Debugging LPI-PL/I Programs

Preface: Using This Manual

Product Information

CodeWatch provides an interactive source-level debugging capability for a number of high-level source languages including LPI-BASIC, LPI-C, LPI-COBOL, LPI-FORTRAN, LPI-PASCAL and LPI-PL/I. Language Processors, Inc., implements these languages on a variety of computers using different operating systems.

This manual provides general information describing the use of CodeWatch in the UNIX,* XENIX,* and MS-DOS* operating system environments. Within this manual, the name UNIX is used when referring to either UNIX or XENIX commands.

Related Documentation

LPI also provides user's guides describing source language information for specific implementations in order to use LPI languages on your particular system, as well as language reference manuals describing each LPI programming language.

Intended Audience

This manual is intended for experienced programmers and analysts. While the approach is not tutorial, the manual is organized so that a programmer who is familiar with one or more of the LPI languages should have no difficulty understanding and using all of the features of CodeWatch. Before you start, it may be helpful to read Chapters 1 through 4, and then go through the sample session (provided in each of the language-specific chapters) in the language with which you are most familar. These sessions present a step-by-step analysis of the debugging process using sample programs to illustrate the various commands and features of the debugger.

Organization of Information

Chapter 1 introduces the features and commands of CodeWatch. Chapter 2 presents the CodeWatch command-line format and the fundamentals of a debugging session. Chapter 3 explains CodeWatch functionality. Chapter 4 provides descriptions of each of the debugger commands. Chapters 5-10 provide language-specific information for debugging LPI-BASIC, LPI-C, LPI-COBOL, LPI-FORTRAN, LPI-PASCAL, and LPI-PL/I, respectively.

The glossary contains definitions of key CodeWatch concepts. The index helps you to locate information quickly on specific CodeWatch features.

Syntax Conventions

The following syntax conventions are used in this manual.

1. Debugger commands are printed in uppercase. You may enter debugger commands using either uppercase or lowercase characters.

2. Bold text indicates the abbreviations that you may use for CodeWatch keywords, for example,

   WHERE

   where **WH** is the abbreviation.

3. Text strings enclosed in angle brackets are optional items, for example,

   PRINT <n>

4. Variable information is printed in *italics*, for example,

   FIND *string*

5. Type names or expressions that are enclosed by braces "{ }" are required.

6. A choice between alternatives is indicated by a vertical bar, for example,

    DSTEP <IN | OVER>

Chapter 1: Overview

Chapter 1: Overview

Source-Level Debugging

Effective source-level debugging can help shorten the time spent on finding programming errors. A program may behave abnormally even after being successfully compiled. Although such a program's syntax is correct, the program still contains programming errors that cause unpredictable consequences such as unintended changes to the values of variables, infinite loops, incorrect output, or abnormal program termination.

Debugging is the process of discovering these bugs and eliminating them. Source-level debugging involves examining and modifying the source language elements of a program while it executes.

CodeWatch is a powerful software development tool that can help you, as a programmer or analyst, locate bugs in programs. When invoked, CodeWatch takes control of the execution of the program. Since the debugger is interactive, it enables you to issue CodeWatch commands from the keyboard. You can monitor what is happening, modify values, and evaluate results immediately.

CodeWatch keeps track of variables, subprograms, subroutines, and data types in terms of the symbols used in the source language. You can reference these items without having to consider the underlying machine language or architecture. You can use CodeWatch to access the source text of the program, to identify and reference program entities, and to detect errors in the program's algorithms and logic.

Summary of Features

The following is a list of CodeWatch features.

- Controlling Program Execution: CodeWatch has control over the execution of the program. Program execution can be started at a specified point, stopped and resumed at different points, transferred from one point to another, and terminated at any point.

  - Breakpointing: Breakpoints can be set to suspend execution at specified source program statements. The status of the program can then be evaluated or modified by entering other CodeWatch commands.
  - Catching: Process control can be returned to the debugger when a given signal is generated.
  - Stepping: Program execution can be stepped by one or more source program statements at a time. The debugger can also be stepped into, over, or out of a called procedure.
  - Tracing: Tracepoints can be set at statements or at procedure and block entries to track program execution.
  - Watchpointing: Watchpoints can be set to monitor given variables. When the contents of the watched variable change, program execution stops and control is returned to the debugger.

- Examining the Source Program: CodeWatch allows you to examine the source program by finding specified text strings, moving to specified line numbers, examining included source files, and displaying specified segments of the source program.
- Environment Control: CodeWatch allows you to set the environment it uses to reference a specific instance of a variable or statement.
- Symbolic Access: The value of a variable can be examined and/or altered as program execution continues. Expressions can be evaluated. Procedure entries and exits can be monitored at the debugger command level.
- Action Lists: One or more of the debugger commands can be combined as a unit and executed by the debugger.
- Macro Facility: Macros can be defined as a sequence of debugger commands that will be used repeatedly throughout a debugging session. The macros can then be used like any other debugger command.
- Command Files: Debugger command files can be created with a text editor or by logging commands to a specified file during a debugging session. These command files can then be used to execute the predefined sequence of debugging commands automatically during a debugging session.
- Issuing Error Messages: CodeWatch identifies error conditions and issues meaningful, complete sentence error messages.

CodeWatch Commands and Options

CodeWatch recognizes certain keywords as having special meaning or as specifying an action to be taken. These keywords are CodeWatch commands and command options.

A keyword can be abbreviated to one character or a minimum string of characters that makes it unique within the set of keywords. Note that partial spelling beyond the abbreviation is also acceptable. Table 1-1 lists the CodeWatch commands, options, and arguments.

TABLE 1-1 CodeWatch Commands, Options, and Arguments

COMMANDS	OPTIONS (If any)	ARGUMENTS
ARGUMENTS		*environment*
BREAKPOINT	/**IF**=*logical-expr* /**IGNORE** /**NIGNORE** /**SKIP**=*n*	*statement-id* + [*action list*]
CATCH	/**IGNORE** /**NIGNORE** /**DEFAULT**	< *signal* >
CONTINUE		
DSTEP	**IN** **OVER**	[*action list*]
ENVIRONMENT		*environment*
EVALUATE	/**ASCII** /**BIT** /**FLOAT** /**HEX** /**INTEGER** /**OCTAL**	*expression* +
FIND		*string* +

TABLE 1-1 CodeWatch Commands, Options, and Arguments (Cont.)

COMMANDS	OPTIONS (If any)	ARGUMENTS
GOTO		*statement-id* +
HELP		
LBREAKPOINT	/**ALL**	*statement-id* ‡
LCATCH	/**ALL**	< *signal* >
LENVIRONMENT	/**ALL**	
LET		*name* = *expression*
LMACRO	/**ALL**	*macro-name* ‡
LOG		*file-name*
LRETURN		
LSOURCE	/**ALL** /**FULL**	
LSTEP		
LWATCH	/**ALL**	< *variable* >
MACRO		*macro-name* = [*action list*]
NBREAKPOINT	/**ALL**	*statement-id* ‡
NCATCH	/**ALL**	< *signal* >
NLOG		
NMACRO	/**ALL**	*macro-name* ‡
NTRACE ENTRY		
NTRACE STATEMENT		
NWATCH	/**ALL**	< *variable* >
POINT	+ | -	*number*
PRINT		*number*
QUIT		
READ		*file-name*
RELOAD	/**ARGUMENTS**	*command-line-arguments*
RETURN		*expression*
SOURCE		*file-name*

TABLE 1-1 CodeWatch Commands, Options, and Arguments (Cont.)

COMMANDS	OPTIONS (If any)	ARGUMENTS
STACK	/**ALL**	*nframes*
	/**ARGUMENTS**	
	/**LOCALS**	
STEP	**IN**	[*action list*]
	OVER	*step-count*
	OUT	
TRACE ENTRY	/**IF**=*logical-expr*	[*action list*]
TRACE STATEMENT		
TYPE	/**FULL**	*expression*
WATCH	/**SILENT**	<*variable*>
	/**IGNORE**	
	/**NIGNORE**	
	/**SKIP**=*n*	
	/**ACTION**	[*action list*]
	/**IF**=*logical-expr*	
WHERE	/**ACTION**	*statement-id*
!		*command*

+ indicates required arguments.
‡ indicates either the argument or the /ALL option is required.

Notes
Command and option abbreviations are indicated by bold text.
Variable information that the user supplies is indicated by italics.

Chapter 2: Using CodeWatch

Chapter 2: Using CodeWatch

Installing CodeWatch

To install Codewatch follow the installation procedures in your Release Notes.

A copy of the release notes for this product can be found on your system in the LPI directory in the file dbgxxxxxx.info for UNIX systems and in the C:\LPI directory in the file dbxxxxxx.inf for MS-DOS systems, where *xxxxxx* is the version number of the release. For example, release notes for CodeWatch version 04.01.00 would be found in the file dbg040100.info on the release media for UNIX systems and dg040100.inf on MS-DOS systems.

Program Preparation

A program must be compiled using the -deb option before it can be run under the control of the debugger. This is because the compiler generates a data base file referred to as the STB file, which contains symbolic information that the debugger needs to reference and manipulate source program symbols and entities, set breakpoints and tracepoints, and control program execution. A program is referred to as being compiled in debug mode when one or more of the program modules have been compiled with the -deb option.

It is helpful to have a current source listing available before using CodeWatch. A listing is obtained by specifying the -l option at compile time. For example, an LPI-COBOL program contained in a source file named amort.cob is compiled in debug mode, with a listing generated, by entering the following command line.

```
lpicobol amort.cob -deb -l
```

The program is then linked as described in your LPI User's Guide.

Invoking CodeWatch

Once the program has been compiled in debug mode and linked, you can use CodeWatch to debug your program. To invoke CodeWatch, the following command format is used:

codewatch < -e >
< -srcpath < *source_directory* < :*source_directory* > . . . > >
< -stbpath < STB_*directory* < :STB_*directory* > . . . > >
< -path < *directory* < :*directory* > . . . > >
< *program-name* >
< *program-arguments* >

where:

-e (echo) specifies that every debugger command entered will be echoed back to the terminal.

-scrpath, -stbpath, and -path specify directory paths to be searched for source and STB files.

program-name is the name of the executable file to be debugged.

program-arguments are command line arguments to be passed to the program.

If the debugger is invoked without a *program-name* argument on a UNIX system, then the a.out file in the current directory will be used if it exists. Systems running under MS-DOS require *program-name* to be specified as there is no default program name.

A debugging session for the COBOL amort program compiled in debug mode is invoked by entering the following command.

codewatch amort

When the debugger is invoked and is in control of the program, it prompts for debugger commands with the following:

DEB>

The STB File

When a program source file is compiled in debug mode, a file called the STB file (symbol table file) is created which contains important information needed by the debugger during the debugging session. The name of this file is constructed by replacing the extension of the source file name (for example, .pl1 for a PL/I program), with a .stb extension. This STB file is placed by default in the directory in which the compilation took place.

In Search of Source and STB Files

After the program is linked and CodeWatch is invoked with the loaded executable file as an argument, the debugger requires access to both the program source file(s) and the STB file(s).

The source file(s) will, by default, be searched for first in the directory which was specified to the compiler relative to the current directory, and if not found, then in the current directory.

The STB file(s) will, by default, be searched for only in the current directory.

It is possible through the use of debugger command line arguments and/or environment variables to specify alternate directories in which to search for both source and STB files before the default places are searched.

The CodeWatch command line options -srcpath and/or -stbpath followed by a list of directory names separated by colons (:) for UNIX systems and semicolons (;) for MS-DOS systems, may be given to specify a list of directories to be searched before any other directories are searched. These directory lists will be searched (in order) for source files and/or STB files respectively. If a directory list starts with a colon or semicolon, then the current directory (.) is assumed to be first (a trailing colon or semicolon is ignored). The command line option -path followed by a directory list will be used in place of the -srcpath option and/or the -stbpath option if one or both is not given.

If the source and/or STB files are not yet found after searching any directories specified on the command line, then the debugger will search directories specified by environment variables in a similar manner. The environment variables CODEWATCH_SRCPATH and/or

CODEWATCH_STBPATH may be set to a list of directory names, separated by colons or semicolons. These directory lists will be searched (in order) for source files and/or STB files respectively. The environment variable CODEWATCH_PATH followed by a directory list will be used in place of the CODEWATCH_SRCPATH environment variables and/or the CODEWATCH_STBPATH environment variables if one or both is not given.

For example, using UNIX syntax, the command:

```
codewatch -srcpath :/pl1progs/src:/src/pl1progs
          -stbpath /pl1progs/stb:/pl1progs/obj  program-name
```

will cause the debugger to search for the source files in the following directories in order: (.) current directory, /pl1progs/src, /src/pl1progs, and the directory specified on compilation command line. The debugger will search for the STB files in the following directories in order: /pl1progs/stb, /pl1progs/obj, and (.) current directory.

Using MS-DOS syntax, the command:

```
codewatch -srcpath ;\pl1progs\src;\src\pl1progs
          -stbpath \pl1progs\stb;\pl1progs\obj  program-name
```

will cause the debugger to search for the source files in the following directories in order: (.) current directory, \pl1progs\src, \src\pl1progs, and the directory specified on compilation command line. The debugger will search for the STB files in the following directories in order: \pl1progs\stb, \pl1progs\obj, and (.) current directory.

Start-up Files

Upon invocation of the debugger, a command file containing debugger commands is automatically executed, if present. The debugger searches for this file in the following locations in the following order:

(1) In the file specified by the environment variable CODEWATCH_INIT.
(2) In the file .codewatch in the user's current directory.
(3) In the file .codewatch in the user's home directory (not applicable for systems running under MS-DOS).

As soon as one of these files is found, the debugger commands contained within are executed, after which normal interactive debugging may continue. It is not required that a start-up file exist.

Ending a Debugging Session

To end a debugging session, enter the QUIT command.

```
DEB> QUIT
```

This causes normal termination of the debugger and the termination of your program. Once the debugging session has been terminated, the system prompt will be displayed.

Keyboard interrupts while the debugger is accepting commands are ignored, but a keyboard quit will terminate the debugging session.

Programs which terminate normally while running under the control of CodeWatch, will automatically be re-initialized with any explicitly set breakpoints (and any associated actions lists) preserved, so that execution may start from the beginning if desired. The exit status with which the program exited is given. Refer to your UNIX Programmer's Manual (exit (2)) or your MS-DOS Technical Reference, depending on your operating system, for more information.

Debugging a Program with Multiple Modules

Not all source modules need to be compiled in debug mode in order to use CodeWatch. You may compile in debug mode only those modules you want to debug.

If you debug a program with multiple modules and the main program is compiled in debug mode, then the initial environment will always be the main program.

If the main program is not compiled in debug mode, the initial environment will be the first debug module specified in the link step.

The COBOL program, amort.cob, used in Chapter 7, "Debugging LPI-COBOL Programs," is an example of a program with multiple modules. To debug just the subprogram mpcalc, only mpcalc needs to be compiled in debug mode. To do this, use the following command sequence.

```
$ lpicobol amort.cob
$ lpicobol mpcalc.cob -deb
$ lpild amort.o mpcalc.o -o amort
$ codewatch amort
```

The debugger is then invoked with mpcalc as the evaluation environment with the following message displayed.

```
CodeWatch setting up "amort". Wait ...

*********************************************************
* CodeWatch,  Revision 4.2.0                            *
* ------------------------                              *
*    Copyright(c) Language Processors, Inc. 1987        *
*********************************************************
Evaluation environment is MPCALC:(inactive)
DEB>
```

Object files generated on systems running under MS-DOS have an .OBJ suffix. For example, the link command line used in the previous example would appear as follows on a MS-DOS system.

```
$lpild amort.OBJ mpcalc.OBJ -o amort
```

Command Line Format

The debugger recognizes certain symbols as having special meaning within the context of a command line. Table 2-1 lists the symbols and their meanings.

TABLE 2-1 CodeWatch Special Symbols

SYMBOL	DEFINITION
+	Unary plus sign, and addition
-	Unary minus sign, and subtraction
;	Command separator
[ ]	Action list delimiters
' or "	String delimiters (' or " depending on source language)
/	Command option indicator
\	Statement reference qualifier
%	Command line continuator, statement label indicator, procedure entry point, and exit point indicator
:	Activation number indicator

A debugger command is made up of one or two keywords. The debugger command line contains a debugger command and optional or required arguments, depending on the command.

In the general format for the debugger command line, optional items are enclosed in angle brackets. Note that the bold square brackets in the action list argument are required delimiters.

The general command line format is:

command < */option* > < *argument* > < [*action list*] >

where:

command -- Names the specific debugger action to be performed

/option -- Modifies the action of the command

argument -- Supplies required or optional information to the command

[*action list*] -- Lists one or more debugger commands

Two examples of CodeWatch command lines, using special symbols follow.

Example 1

```
BREAKPOINT 90 [EVALUATE var1; E var2]
```

This example sets a breakpoint at line 90. The program variables "var1" and "var2" will be evaluated when this breakpoint is encountered.

Example 2

```
BREAKPOINT LABEL1 + 8; LBREAKPOINT /ALL
```

This example sets a breakpoint eight statements past the label LABEL1 and then lists all breakpoints.

Command Variations

Many of the debugger commands allow a number of prefixes which specify a variation of the command function. An explanation of these prefixes follows.

L*command*	• Lists information on *command.*
D*command*	• Sets the default values for *command.*
N*command*	• Removes instances of *command.*

For example,

LMACRO	• Lists information on the current macro definitions.
DSTEP	• Sets the default values for the STEP command.
NBREAKPOINT	• Removes breakpoints.

Entering CodeWatch Commands

Commands may be entered in either upper or lowercase. Case is unimportant to the debugger except in quoted literals. (However, for the purposes of the manual's syntax conventions, commands will be shown in uppercase.)

Enter debugger commands at the keyboard in response to the debugger prompt (DEB>). For example, the PRINT command displays the current line from the source program:

```
DEB>  PRINT
```

Several commands can be entered on a single line, separated by semicolons, as shown in the following example:

```
DEB> BREAKPOINT 25; EVALUATE DISTANCE
```

Using abbreviations, the previous command line can also be written as:

```
DEB> B 25; E DISTANCE
```

Commands are executed one at a time, from left to right. If any command causes an error message to be displayed, the rest of the command line may be discarded. If any command causes the program to resume execution, the rest of the command line will be executed when the debugger resumes control. For example, the following series of commands will be executed only until CONTINUE is reached.

```
DEB> B 93; FIND COUNT; CONTINUE; EVAL COUNT
```

If a command line is too long for a single line, it can be continued on the next line by entering a blank and a percent sign " %" as the last character on the line followed by a carriage return. The percent sign is not part of the command. It simply informs the debugger that you want to continue entering commands. The debugger prompts with "...>" indicating it is ready for the continuation of command input. For example:

```
DEB> BREAKPOINT 123 [ STEP IN; FIND EXPENSES; %
...> EVALUATE EXPENSES; CONTINUE ]
```

Using Action Lists

An action list is a set of one or more CodeWatch commands specified for execution at breakpoints, after steps, or at tracepoints. You specify an action list by enclosing one or more debugger commands within brackets ("[ ]"). The commands must be separated by semicolons. You may use spaces after semicolons to aid readability. The commands that you enter in an action list are not checked for correctness until they are executed. For example, the commands in the action list in the previous example will not be evaluated until program execution reaches the breakpoint set at line 123. The format for the action list is:

[*command1*; *command2*; ...; *commandn*]

A typical action list that could be set at a breakpoint is:

```
[ EVALUATE COUNTER; STEP; EVALUATE DISTANCE;
    CONTINUE ]
```

In more advanced debugging, commonly used sequences of commands can be defined as an action list with the MACRO command. The single macro name can then be used repeatedly throughout the debugging session to refer to a complex series of actions. (For an explanation of defining macros, see Chapter 3.)

Pointer Concepts

The debugger maintains two pointers, the execution pointer, and the source file pointer. This section explains these pointers and how they are used during a debugging session.

The Execution Pointer

The debugger maintains a pointer to the current execution point. This is the point at which execution resumes or begins following a STEP or CONTINUE command. The execution pointer is reset during execution of the program. The RELOAD command sets the execution pointer to the start of the program. The GOTO command and the RETURN command are the only commands that can directly reset the execution pointer. The WHERE command used without any arguments will display the current execution point.

The Source File Pointer

The debugger also maintains a pointer to the current line in the source file currently being displayed. The SOURCE command changes the current source file being displayed. The POINT and PRINT commands change and display the source file pointer respectively. The ENVIRONMENT command without any arguments resets the source file pointer to the current execution point.

Environment Concepts

The environment is a frame of reference for the debugger to identify program entities such as certain instances of statements and variables.

This section explains these entities and how to refer to them in the context of a debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. Program blocks are defined according to the source language. Refer to the appropriate language chapter for language specific definitions of program blocks.

Block Activation Numbers

Activation numbers only pertain to recursive procedures and to languages that allow recursive procedures.

A procedure is active when it has been called during program execution. Each new call to a procedure counts as a new activation. The activation number specifies a unique activation of a procedure when more than one activation exists. Multiple activations occur when a procedure is recursive (A calls A) or indirectly recursive (A calls B calls A). Activation numbers are either absolute or relative.

Absolute Activation Numbers

An absolute activation number is simply a positive integer. Number 1 denotes the first activation of a procedure, number 2 the second, and so forth. To refer to an activation of a procedure using absolute activation numbers, the procedure name is followed by a colon ":" and the positive integer that denotes the activation.

Relative Activation Numbers

A relative activation number is either zero or a negative integer. A relative activation number specifies the number of activations to count backwards from the most recent activation. The number 0 denotes the most recent activation. The activation directly preceding the most recent activation is denoted by "-1". If there have been five activations of the procedure FACTORIAL, then "FACTORIAL:0" refers to the fifth activation and "FACTORIAL:-2" refers to the third activation. This activation can also be referred to as "FACTORIAL:3" (that is, using the absolute activation number 3).

Using Activation Numbers

Each activation of a procedure has its own distinct values for automatic (or local) data, and each activation may have different actual parameters or arguments. An activation number can be used to specify the environment for evaluating or displaying variables and procedure arguments. An activation number can also be used when setting a breakpoint in a particular activation of a procedure.

If the activation number is omitted from the name, then the debugger uses a default of 0, the most recent activation. The only exception to this is the BREAKPOINT command, where omitting the activation number means that a breakpoint is to be taken on every activation of the procedure.

Statement Identifiers

Commands such as BREAKPOINT require identification of statements within the program being debugged. Statement identifiers are either source line numbers or statement labels.

Line Numbers

The simplest way to refer to a statement is by using its source line number, that is, the physical line number in the source file on which the statement starts.

If an unqualified source file line number is specified (no environment reference), the first executable statement on the specified line, if present, or the first executable line following that line is used by the debugger.

If a source file line number is qualified by an environment reference, for example, SIFT\18, the translation of the line number to an executable statement is executed in the same manner as described for unqualified line numbers, except the set of executable statements the line number can be translated to is restricted to those contained in the specified environment block.

Labels

The way to refer to a labeled statement is to use its statement label. For any language in which statement labels are numbers, such as Pascal or FORTRAN, a percent sign (%) must be used in front of the statement label. For example, in the following Pascal code:

```
325:
326:     9999: writeln('ERROR: INVALID DATA.');
327:     end. {Program Terminates}
```

the statement reference 326 and %9999 both refer to the writeln statement. In the following PL/I code:

```
 9:  DUMMY:
10:  %INCLUDE 'MYFILE.PL1'; /* PL/I sample */
11:  CHKPAGE:
12:  if I = 66 then
13:       do; put skip; I = 0; goto DONEIT;
14:       end;
15:  I = I + 1;  DONEIT: J = 0; K = 0;
16:  L = 0;
```

the statement reference 11 and CHKPAGE both refer to the same labeled line of code.

Line Number and Statement Offsets

A line number offset can be used to locate a statement when it is not labeled or its source line number is unknown. A line number offset is formed by placing a plus sign (+) followed by a positive integer after a source line number or a statement label. For example, line 15 in the PL/I code can be referred to by the program as "11 + 4" or as "CHKPAGE + 4". Of course, "15 + 0" refers to the same line.

A statement offset must be used to refer to an unlabeled statement that is not the first statement on a line. A statement offset is formed by placing a period followed by a positive integer (indicating the number of statements from the beginning of the line) after the line number offset. (If the line number offset equals zero, then the plus must be used.) For example, the K = 0 statement on line 15 of the PL/I code can be referred to by the debugger as "15 + 0.3" or CHKPAGE + 4.3".

Entry and Exit Points

Every procedure has two special statements, the entry point *to* and the exit point *from* the procedure. The entry point is the place during execution at which the procedure has been called, but before other prelude code has run. A procedure's entry point is referred to within CodeWatch by specifying the procedure name followed by a backslash and "%ENTRY" (or simply "%E"), for example, READ_INPUT\%ENTRY. The entry point to the current activation of a procedure may simply be referred to as "%ENTRY" (or "%E").

The exit point is the place after which the return value (if any) has been computed and the return statement has been executed. All local variables are still defined. An exit point is referred to by the procedure name followed by a backslash and "%EXIT" (or "%EX"), for example, READ_INPUT\%EXIT. The exit point to the current activation of a procedure may be referred to by "%EXIT" (or "%EX").

Entry and exit points may not be referred to in conjunction with statement offsets.

Refer to Chapter 7, "Debugging LPI-COBOL Programs," for information regarding the COBOL STOP RUN statement and exit points.

Statement Qualification by Environment

A statement reference may need to be qualified by a reference to the environment in which the statement occurs. This is done by putting the block name of the environment (with an optional activation number) and a backslash in front of the statement reference. Using just the backslash without the block name refers to the environment containing the current execution point. For example:

```
PRINT_OUT\10
```

Referencing Included or Copied Files

Many source programs have the contents of separate files inserted into the program at compile time. Programs written in BASIC, C, FORTRAN, Pascal, or PL/I use a INCLUDE statement for this purpose; COBOL programs use a COPY statement. The debugger handles all these statements in essentially the same way.

When the debugger encounters one of these statements in a program, you can display the contents of the included or copied file by entering the SOURCE command followed by the filename cited in the INCLUDE or COPY statement. The lines in the file will be numbered consecutively from the first line in the file. Executable code contained within included files cannot be debugged with CodeWatch.

To return to displaying lines from the source file containing the current environment, simply enter the SOURCE command with no arguments.

See also the discussion on "Examining the Source Program" in Chapter 3.

CodeWatch Error Messages

When CodeWatch encounters an error condition, a message is displayed at your terminal. The error message contains two lines of text: the first line explains the error; and the second line echoes the command line as it was entered, with a caret indicating the approximate location of the error. After displaying an error message, the debugger issues a prompt and waits for another command.

There are three general categories of error messages, as follows.

1. When the debugging session cannot begin, the debugger will display an error message at invocation. This indicates that a filename required by CodeWatch is in error because:

   — It does not exist or is not in the current directory.

   — The program file specified to be debugged is not an executable file.

   — None of the program modules in the program file was generated from a compilation using the -deb option.

   — The necessary debugger information files cannot be found in any of the specified or default directories. Refer to the section, "In Search of Source and STB Files," earlier in this chapter for further information.

2. When the debugging has started, the debugger issues a message when the command syntax is incorrect. Samples follow:

```
DEB> TYPEPRIMES
**Error** Invalid Command
TYPEPRIMES
^

DEB> TRACE EMTRY
**Error** Syntax error
TRACE EMTRY
      ^
```

3. The third kind of error occurs when the debugger cannot correctly interpret a command line because of an incorrect variable name, an invalid argument, or some similar problem. Two samples are shown here:

```
DEB> EVALUATE ths_prime
**Error** Undefined variable or built-in function
E ths_prime
  ^

DEB> BREAKPOINT 100000
**Error** Statement 100000 not found in procedure
```

Aborted Program Recovery

CodeWatch has the ability to recover from and to reasonably debug user programs which have stopped due to certain operating system errors. When this occurs, the following are indicated by the debugger:

- the error which caused the program to stop,
- the current user program counter (the machine address of the execution point)
- the name of the routine in which the error occurred.

If the routine in which the program stopped was compiled in debug mode, then the source file line number of the execution point at the time of the error will also be given. The current evaluation environment will automatically be set to this environment and the debugging session may continue.

If the routine in which the program stopped was NOT compiled in debug mode, then the environment of the most recently called routine which WAS compiled in debug mode, is indicated, and the current evaluation environment is set to that environment. In this case, most debugger actions are allowed, including continuing execution (using the CONTINUE command) from the point at which the program received the error. Note that it is not legal to STEP or GOTO from a non-debug routine.

Using the STACK command, a stack traceback may be used to indicate the sequence of subroutine calls up to the time of the error.

Chapter 3: CodeWatch Functionality

Chapter 3: CodeWatch Functionality

The first section of this chapter describes the commands that let you control the execution of your program. The second section discusses environment control issues. The third section explains how to examine, modify, and evaluate the contents of data items. The last section of this chapter describes other functionality, such as the macros facility, debugger command files, and online help.

Some of the important concepts that are referred to in this chapter include:

Execution pointer	• pointer to the current execution point.
Source file pointer	• pointer to the current line in the current source file.

Program Control

This section explains the CodeWatch commands that are used to control program execution. You may control the execution of your program in the following ways:

- stepping through one or more executable statements at a time
- setting breakpoints to suspend program execution at any executable statement
- catching signals to return process control to the debugger
- setting watchpoints to monitor changes in specified variables

Statement tracing and entry tracing give you the ability to monitor every executable statement and the entry and exit points of programs.

The Breakpoint Commands

Breakpointing is used to suspend program execution at specified locations so that you can interact with the program using debugger commands.

The BREAKPOINT command is used to set a breakpoint at a specified statement or to modify the characteristics of a breakpoint already set at that statement. Up to 64 breakpoints can be active at any one time.

A breakpoint counter is associated with each breakpoint. When the breakpoint is created, the counter is set to zero; each time the breakpoint is encountered, the counter is incremented by one. All breakpoint counters are set to zero when a program is reloaded.

The NBREAKPOINT command removes either a specified breakpoint or all breakpoints.

The LBREAKPOINT command lists information on a single breakpoint or on all breakpoints.

The Catching Commands

The CATCH command, with a signal name and no arguments specifies that process control will return to the debugger when the given signal is generated. The signal name can be either a signal mnemonic or the signal number corresponding to that mnemonic.

The NCATCH command stops process control from returning to the debugger when the specified signal is generated. The child (user) process continues without interruption, as if the given signal had been generated.

The LCATCH command with no arguments lists all the signals which are currently set to be caught. When used with the /ALL option, LCATCH lists two sets of signals, those which are currently set to be caught and those which are currently set to not be caught.

The Stepping Commands

The STEP command starts program execution at the current location of the execution pointer and stops execution after one or a specified number of statements.

The DSTEP command sets the default mode for stepping. With DSTEP, the default action of stepping over calls to routines can be changed to stepping into called routines. The DSTEP command is also used to specify the default action list for the STEP command.

The LSTEP command lists the current mode of stepping, that is, either IN or OVER called routines, and the default action list.

The Tracing Commands

Tracing enables information about all procedure and block entries and statements to be reported as the program executes.

The TRACE ENTRY command sets tracepoints that enable tracing of all procedure and block entries and exits by printing a message each time a procedure or block is entered or exited.

The TRACE STATEMENT command sets tracepoints that enable the tracing of every statement by printing a message identifying the statement.

At each statement, if there is no other action to be performed (that is, if there is no breakpoint, entry trace, single step action, and so on), the identifying message is printed and execution continues.

The NTRACE command disables entry or statement tracing.

The Watchpoint Commands

Watchpointing is used to monitor specified variables. The WATCH command designates the given variable to be watched. When the contents of the watched variable change, program execution stops and control is returned to the debugger.

A watchpoint counter is associated with each watchpoint. When the watchpoint is created, the counter is set to zero; each time the watchpoint is incurred, the counter is incremented by one. When a program is reloaded, all counters are set to zero.

The LWATCH command lists information on the given watchpoint(s). If the /ALL option is specified, information is listed on all of the current watchpoints.

The NWATCH command removes watchpoints for the given variable(s). If the /ALL option is specifed, all watchpoints currently set are removed.

Controlling Program Execution

During a debugging session, CodeWatch maintains a pointer that tracks the current point of program execution. When execution of the program has been halted by a debugger action, the pointer maintains the location as the current execution point. The current execution point can always be determined using the WHERE command. This allows you to control the execution of the program by using one of several simple debugger commands.

The CONTINUE command is used to begin program execution or to resume execution following a breakpoint or a step operation.

The GOTO command moves the execution pointer to a specified statement. Program execution resumes at this point when a CONTINUE or STEP command is issued.

The statement is specified by a line number, a simple statement label, or a statement label followed by a line number offset or statement offset. The named statement must exist in the current program block.

When the GOTO command is executed, the debugger displays the new execution point.

The RELOAD command reloads the user program, preserving any explicitly set breakpoints and any associated action lists. A subsequent CONTINUE or STEP will start program execution from the beginning of the program.

The RETURN command transfers the current execution point to the exit point of the current procedure. If the procedure returns a value, an expression must be given indicating the value to be returned. Refer to the section, "The RETURN Commands" later in this chapter for more information.

The QUIT command causes termination of the debugging session.

Environment Control

In many cases during a debugging session, it is necessary to refer to an environment to establish a frame of reference for identifying variables or statement identifiers. An environment is simply a program block which the program refers to by specifying the name of the block or any statement within the block. This allows the debugger to find a specific instance of a variable or statement. Usually, a simple block name is adequate to specify an environment. It refers to the most recent activation of that block. To establish some other activation of the block as an environment, use a block activation number.

An active environment exists on the program stack as a stack frame, containing automatic data for the environment. However, environments that do not exist on the program stack can also be referenced. Such environments are called inactive environments. For example, a procedure that has not been called, and thus has no stack frame, is inactive. Another example of an inactive environment is one that is referenced by a block name followed by an activation number that is higher than the current activation of the block. It is possible to examine static and external data in inactive environments and to set breakpoints in them. An active environment must be specified to examine automatic data or procedure arguments.

The ENVIRONMENT Commands

The ENVIRONMENT command sets the current evaluation environment to provide scope to the debugger for identifying variables and statements.

The environment specified in the command can be a simple block name or a block name followed by an activation number or a statement identifier. An activation number is used to establish, as an environment, an activation of a block other than the most recent one. A statement identifier is merely a convenient means of identifying the block which immediately contains the statement. If the environment argument is not specified, the evaluation environment is set to the environment containing the current execution point.

Whenever the debugger is reentered after program execution pauses for a breakpoint, or single step, and so on, the evaluation environment is reset to the environment containing the current execution point.

The LENVIRONMENT command lists the current evaluation environment or all the evaluation environments.

The STACK Command

The STACK command is used to print a traceback of a specified number of stack frames.

Symbolic Access

This section describes the various methods and commands to access, modify, and display symbolic information.

Variable Names

The rules for identifying variables in the debugger are almost the same as the rules defined for the evaluation language. The same naming conventions are used, and all type and scope rules apply. The one exception is that an environment name followed by a backslash may be used to reference a variable in another program block. The names of variables are case-sensitive or case-insensitive depending on the language of the module in which they are contained.

Referencing Elements of Arrays and Tables

Elements of arrays and tables are referenced using the constructs of the source language. For example, the tenth element of the array primes, used in the Pascal program in Chapter 9 would be evaluated with the following command:

```
DEB> EVALUATE primes(10)
```

CodeWatch allows you to access slices of arrays in programs written in LPI-COBOL, LPI-PASCAL, and LPI-PL/I by using the following syntax:

array-name [*m*:*n*]

or

array-name (*m*:*n*)

where *m* is the first array element and *n* is the last element to be referenced. An asterisk (*) used in place of the *m:n* syntax, specifies that the entire array slice be referenced.

The ARGUMENTS Command

The ARGUMENTS command prints the arguments to an environment. The environment specified in the command must be the name of an active procedure (see the section, "Environment Control," earlier in this chapter). If an environment argument is omitted, the debugger defaults to the current evaluation environment.

The EVALUATE Command

The EVALUATE command is used to evaluate and print the resultant value of expressions in the source language program. When the expression is evaluated, appropriate conversions are performed.

If the /display mode option is omitted, the debugger derives a default display mode from the resultant type of the expression. When the expression has been evaluated, the debugger prints the value in the specified display mode.

The LET Command

The LET command assigns a value of an expression to a name.

When the expression is evaluated, appropriate type conversions are performed according to the rules of the appropriate language. The resultant value is assigned to the named variable. If the type conversion is illegal then the debugger will issue an error message.

Note
If the current environment is within a module compiled by LPI-C, the LET command is equivalent to the EVALUATE command. (In C, assignments are merely expression operators with side effects.)

The RETURN Commands

The RETURN command allows you to set the return values of procedures. RETURN transfers the current execution pointer to the exit point of the current procedure and accepts expressions indicating the return value when the procedure returns a value.

The LRETURN command lists the return value of a procedure and is used when the current execution point is at the exit point of a procedure.

The TYPE Command

The TYPE command prints the resultant data type of an expression in terms of the data descriptions of the current source language.

Examining the Source Program

This section describes the CodeWatch commands for searching, printing, and displaying different locations in the source file(s).

The FIND Command

The FIND command locates a line in the source file containing a specified pattern of characters and reports that line.

The debugger finds the first occurrence of the character pattern after the current file position in the source listing file, then prints the line containing that occurrence of the pattern. The FIND command is case sensitive and matches character patterns exactly.

The WHERE Command

The WHERE command reports either a specified location or the current point of execution. The specified location may be the either a name of a routine, a line number, or a statement label.

The POINT Command

The POINT command locates a line number within the debug listing file and reports that line. If no line number is specified, the line corresponding to the current source file pointer is printed.

The PRINT Command

The PRINT command prints a specified number of source lines from the current debug listing file or prints the current source line.

The SOURCE Commands

The SOURCE command changes the name of the source file to be displayed.

The SOURCE command is needed when you want to look at the contents of COPY or INCLUDE files. Referencing these files is discussed in the earlier section "Referencing Included or Copied Files."

The LSOURCE command lists the name of the source file currently being displayed.

Other Functionality

This section describes additional features of CodeWatch.

MACROS Facility

The MACRO command is used to define a macro as shorthand for a series of debugger commands that might be reused during the debugging session. Sixteen macros may be defined at any one time.

Note
Do not choose a *macro-name* that is the same as a debugger command name because the *macro-name* will override the debugger command.

The series of debugger commands is specified as an action list (see Chapter 2, "Using Action Lists").

Once a macro has been defined, its name may be used just like any other debugger command name. The macro in the command line is replaced by the debugger commands specified in the action list. Those commands are then executed normally.

If the last (or only) command in the series of debugger commands does not end with a semicolon, any additional arguments which appear on the command line after the macro name, are supplied as arguments to the last command.

Listing Macros

The LMACRO command lists the definition of one or more specified macros or the definitions for all currently defined macros.

The debugger lists the series of debugger commands that was defined for the macro.

Removing Macros

The NMACRO command removes the definition of one or more specified macros or the definitions of all currently defined macros.

Debugger Command Files

The LOG command can be used to create debugger command files by causing all subsequent debugger commands to be logged to a specified file.

The NLOG command ends the logging of debugger commands to a file.

The READ command is used to execute debugger command files, displaying each command to the screen as the command is executed.

Online Help

The HELP command lists the available debugger commands.

Invoking the Command Interpreter

The ! command invokes the command interpreter to read the remainder of the line following the ! command.

This command is not available under the MS-DOS operating system.

Chapter 4: CodeWatch Commands

Chapter 4: CodeWatch Commands

Overview

This chapter lists and describes all the CodeWatch commands. The commands are presented in alphabetical order by their command name. Each command definition starts on a new page. The definitions consist of subsections that contain information on use, format, a brief description of the command, and an example.

Use

Prints the arguments of a specified procedure environment.

Format

ARGUMENTS < *environment* >

where:

environment is a name of an active procedure.

Description

The ARGUMENTS command prints the arguments to a named procedure environment. The procedure environment can be any program block. Refer to the section "Environment Control" in Chapter 2.

If *environment* is omitted, the debugger defaults to the current evaluation environment.

Example

In this example, the arguments to the sift procedure are displayed.

```
DEB> ARGUMENTS sift
N =    10 { INTEGER }
```

Use

Suspends program execution to allow for debugger actions.

Format

BREAKPOINT < *statement-id* > < [*action list*] >
< /**SKIP**=*n* > < /**IGNORE** > < /**NIGNORE** >
< /**IF**=*logical-expr* >

where:

statement-id identifies the statement at which the breakpoint will be set or modified.

Description

The BREAKPOINT command is used to set a breakpoint at a specified statement or modify the characteristics of a breakpoint already set at that statement. If *statement-id* is omitted, the breakpoint will be set at the current execution point, that is, the statement to be executed when program execution is resumed. The maximum number of breakpoints that can be active at one time is 64.

Breakpoints can be set at the entry point (%ENTRY) of external routines which have not been compiled in debug mode.

Breakpoints can be set only at lines containing executable statements within routines compiled in debug mode. For example, lines containing only comments or declarations, or only the keyword ELSE are not valid breakpoint locations. If you try to set a breakpoint at one of these locations, the debugger will set the breakpoint at the first executable statement after the one named in the command.

- If the *statement-id* is qualified by a block name with an explicit activation number, the breakpoint applies only to that activation. If a block activation number is not specified, the breakpoint will be taken on every activation of the specified statement. The numbered activation may be higher than the most recent activation of the block. Relative activation numbers are converted to absolute activation numbers when the BREAKPOINT command is entered. Refer to the section "Block Activation Numbers" in Chapter 2.
- If an *action list* is present, it is set as the action to be performed when the breakpoint is taken. To remove *action list*, simply specify a null action list, that is, BREAKPOINT [ ]. An *action list* of [CONTINUE] causes the breakpoint to act like a simple tracepoint; at each execution of the statement a message identifies the statement and execution continues.

A breakpoint counter is associated with each breakpoint. When the breakpoint is created, the counter is set to zero; each time the breakpoint is encountered the counter is incremented by one. All breakpoints are set to zero when a program is reloaded.

- If the /SKIP=*n* option is present, a breakpoint skip counter is set to the value specified by *n*, causing the breakpoint to be skipped *n* number of times. Once the breakpoint skip counter is set, it remains in effect until its value decreases to zero.

- If the /IGNORE option is present, the breakpoint is flagged to be ignored and is not executed when it is encountered. The /IGNORE option is useful to temporarily disable a breakpoint. The /NIGNORE option cancels the ignore flag so the breakpoint will be in effect.
- An IF condition may be used to qualify the breakpoint. If present, the IF condition must be the last element on the command line.

When a breakpoint with the /IF= option is set at a particular program statement, the executing program will be suspended and the debugger will be activated whenever that program statement is about to be executed. The following will then occur:

- The IF condition is evaluated; the evaluation always takes place in the environment and language of the program statement. If the IF condition is false, program execution continues. If the IF condition is true, the debugger performs the following actions:
  - The breakpoint counter is incremented.
  - If the /IGNORE flag is on for this breakpoint, execution continues.
  - If the skip count for this breakpoint is non-zero, the count is decremented and execution continues. However, the skip count for an IGNOREd breakpoint is not affected.
  - The debugger reports an announcement of the breakpoint.
  - If an action list is specified, it is executed. The action list may specify that program execution is to continue. If not,
  - The debugger prints its prompt and waits for commands.

Example

In the following example a breakpoint is set at the exit point of the MPCALC procedure.

```
DEB> BREAKPOINT MPCALC\%EXIT
```

In the following example the breakpoint is only executed if the value of MAXV is 5.

```
DEB> B READ_INPUT\18 /IF= MAXV = 5
```

Use

Returns process control to the debugger when the given signal is generated.

Format

CATCH [**/NIGNORE** | **/IGNORE**] < *signal* >

or

CATCH /DEFAULT

where:

signal can be either a signal name mnemonic (such as SIGALRM, SIGINT, and so on) or the signal number corresponding to that mnemonic. Signal names may be specified in either uppercase or lowercase.

Description

The CATCH command, with a signal name and no arguments, specifies that process control will return to the debugger when the given signal is incurred. When the child (user) process is subsequently resumed (for example, by using the CONTINUE or STEP commands), it will continue as if it had incurred the given signal.

The /IGNORE option allows the debugger to catch the signal (that is, stop the child process) but disregard the signal. In other words, when the child process is resumed, it will continue as if it had never incurred the given signal. The /NIGNORE option allows the debugger to

recognize the signal so that when the child process is resumed, it will continue as if it had incurred the given signal. The /DEFAULT option sets all CATCH settings back to the debugger's default settings. Refer to the NCATCH and LCATCH commands for more information.

By default, certain signals are caught by the debugger; for a listing of these signals type LCATCH /ALL.

Note
The signals SIGILL and SIGTRAP are actually set to be ignored by the debugger. These are special signals because they are used by the debugger for breakpointing. The settings for these two signals are not modifiable by the user.

Examples

In the following example, process control will return from the child process to the debugger when SIGALRM is generated.

```
DEB> CATCH /IGNORE SIGALRM
Changing SIGALRM to be caught and ignored.
```

When the child process resumes, it will ignore the SIGALRM and continue as if it had never generated a SIGALRM.

In this example, the number 28 represents the signal SIGWINCH. Process control will return from the child process to the debugger when a SIGWINCH is generated.

```
DEB> CATCH /NIGNORE 28
Changing SIGWINCH to be caught and not ignored.
```

When the child process resumes, it will recognize the signal SIGWINCH and continue as if it had incurred a SIGWINCH.

Use

Begins or continues program execution.

Format

CONTINUE

Description

The CONTINUE command is used to begin program execution initially or following a reload operation, or to resume program execution following a breakpoint or a step operation.

The CONTINUE command is often the last command in an action list.

Example

In this example, program execution is continued until the breakpoint at line 112 is encountered.

```
DEB> CONTINUE
Break at AMORT\112
```

In this example, CONTINUE is the last command executed in conjunction with the STEP command. Refer also to the sections describing the STEP and PRINT commands later in this chapter.

```
DEB> STEP [PRINT; CONTINUE]
Step at MPCALC\23
  23:           S1 SECTION.
Break at MPCALC\%EXIT
```

Use

Sets the default mode and/or action list for the STEP command.

Format

DSTEP < IN | OVER > < [*action list*] >

Description

The DSTEP command sets the default stepping mode to either IN or OVER called routines; and/or the default action list for the STEP command. If DSTEP is not used, OVER is the default stepping mode.

Example

In this example, the default stepping mode is set to IN and the PRINT, TRACE ENTRY, and CONTINUE commands are set as the default actions to be executed at each step. When the subsequent STEP is executed the current execution point is displayed, program execution continues, and tracing information is displayed. Refer to the sections on PRINT, TRACE ENTRY, and CONTINUE in this chapter.

continued

```
DEB> DSTEP IN [PRINT; TRACE ENTRY; CONTINUE]
DEB> STEP
Step at PRINT_OUT\51
  51:          J=1
**** PRINT_OUT\%EXIT
**** SIFT\%EXIT
     .
     .
     .
```

Use

Sets the current evaluation environment.

Format

ENVIRONMENT < *environment* >

where:

environment can be a simple block name or a block name followed by an activation number or a statement identifier.

Description

The ENVIRONMENT command sets the current evaluation environment to provide scope to the debugger for identifying variables and statements.

An activation number is used to establish, as an environment, an activation of a block other than the most recent one. If the environment argument is not specified, the evaluation environment is set to the environment containing the current execution point and the current source file pointer is set to the current execution point.

Whenever the environment changes during program execution, the evaluation environment is reset to the environment containing the current execution point.

Example

In the following example, the environment is set to MAIN and then reset to the environment containing the current execution point, which in this example is the read_input routine.

```
DEB> ENVIRONMENT MAIN
DEB> ENV
Environment reset to read_input
```

Use

Evaluates and prints the resultant value of expressions.

Format

EVALUATE < */display mode* > < *expression* >

where:

expression can be any expression that may occur in the source language program including references to simple as well as aggregate (such as array, record, and structure elements) type variables.

Description

The EVALUATE command is used to evaluate and print the resultant value of expressions in the source language program. If *expression* is omitted, then EVALUATE uses the previous expression that was evaluated. If no previous expression was evaluated, then an error message is generated.

The display mode is the mode in which the value of the expression is to be printed. The valid display modes are ASCII, BIT, FLOAT, HEX, INTEGER, and OCTAL. Table 4-1 defines these display modes.

TABLE 4-1 Display Modes

Display Mode	Definition
ASCII	Displays the value as a series of ASCII characters.
BIT	Displays each bit in the value as a 1 or a zero.
FLOAT	Displays the value as a single-precision floating-point number.
HEX	Displays the value as a series of hexadecimal numbers.
INTEGER	Displays the value as a series of signed decimal numbers.
OCTAL	Displays the value as a series of unsigned octal numbers.

A specified range of an array can be evaluated using the following syntax when debugging LPI-COBOL, LPI-PASCAL, and LPI-PL/I programs.

EVALUATE *array-name* [*m*:*n*]

or

EVALUATE *array-name* (*m*:*n*)

where *m* is the starting point of the array range and *n* is the ending point of the array range.

An asterisk (*) specifies that all elements of a dimension are to be evaluated. For example,

EVALUATE *array-name* [*,3:5]

or

EVALUATE *array-name* (*,3:5)

displays the values of each element of the first dimension and the third through fifth elements of the second dimension of a two dimensional array.

Specifying only the array name for these languages will cause the evaluation of every element of every dimension of the array. Similarly, specifying a structure, record or group name for these languages will cause the evaluation of every member of the group.

Examples

In this example, the value of the variable count is evaluated.

```
DEB> EVALUATE count
COUNT=     5 INTEGER
```

In this example, the third through fifth elements of the array, primes, are evaluated.

```
DEB> EVALUATE primes[3:5]
PRIMES(3)=     3 INTEGER
PRIMES(4)=     5 INTEGER
PRIMES(5)=     7 INTEGER
```

FIND F

Use

Locates and prints a specified string.

Format

FIND < *string* >

Description

The FIND command locates a line in the source file containing a specified string and reports that line.

Quotation marks may be used to delimit *string* and are required for strings that contain spaces. If *string* is not specified, the default value will be the string specified in the previous FIND command. If there is no *string* specified in the first instance of the FIND command an error message will be generated.

The debugger finds the first occurrence of *string* after the current file position in the source file, then prints the line containing that occurrence of the string. The FIND command is case sensitive, so that string ABC differs from abc and Abc.

If *string* is not found the current source file pointer is positioned at the last line of the user program.

Example

In this example, the string "Main" is located.

```
DEB> FIND Main
6: *      Main Program
```

If *string* is not identical in terms of case to the string in the source file it will not be found. For example,

```
DEB> FIND main
94: BOTTOM
```

Use

Moves the execution pointer to a specified statement.

Format

GOTO *statement-id*

where:

statement-id can be a line number, a simple statement label, or a statement label followed by a line number offset or statement offset.

Description

The GOTO command moves the execution pointer to a specified statement. Program execution resumes at this point when a CONTINUE or STEP command is issued.

The named statement must exist in the current program block.

Notes
It is not valid to put an activation number on the statement-id. If the activation number of an existing previous block invocation is put on the statement-id, it is ignored; otherwise, the error message "specified activation does not exist" is displayed.

It is not possible to use the GOTO command to go to the %EXIT point of a block. The RETURN command can be used to go to the %EXIT point, however.

When the GOTO command is executed, the debugger displays the current execution point.

Example

In this example, the execution point is changed to line 19, from the previous execution point as displayed by the WHERE command. Refer to the description of the WHERE command later in this chapter.

```
DEB> WHERE
Current execution point is READ_INPUT\%ENTRY
DEB> GOTO 19
Execution point is now READ_INPUT\19
```

HELP H

Use

Lists all of the debugger commands and options.

Format

HELP

Description

The HELP command lists debugger commands and options information on the screen.

Example

The following example is a partial list of the help command display.

```
DEB> HELP
CodeWatch Commands ---

ARGuments                          Let
  <environment>                    LCATch /All | <signal>
Breakpoint                         LMAcro /All | name
  <statement_id>                   LOg file_name
  [action_list]                    LRETurn
  /IF=logical_expr                 LSOurce /All | /Full
  /SKIP=n                          LStep
  /NIgnore | /Ignore               LWAtch /All | <variable>
CATch <signal>                     MAcro
  /Ignore | /NIgnore                 name = [action_list]
  /DEFault                         NBreakpoint
Continue                             statement_id | /All
DStep In | Over                    NCATch <signal>
  [action_list]                    NLOg
ENVironment <env_name>             NMAcro /All | name
Evaluate /display_mode             NTRace Entry
  <expression>                     NTRace Statement
Find <string>                      NWAtch variable | /All
Goto statement_id                  POint < +n | -n | n >
Help                               Print <n_lines>
LBreakpoint                        Quit
  statement_id | /All              REAd command_file
LENVironment /All                  Reload /ARGuments
```

LBREAKPOINT LB

Use

Lists information on breakpoints.

Format

LBREAKPOINT < *statement-id* | /ALL >

where:

statement-id names the statement at which the breakpoint is to be listed.

Description

The LBREAKPOINT command lists information on a single breakpoint or on all breakpoints.

If *statement-id* is omitted, information is listed on the breakpoint at the current execution point. If /ALL is specified, information is listed on all breakpoints. The information listed includes the skip count, execution count (the number of times the breakpoint has been encountered) and any specified action list.

Example

In this example, all breakpoints are listed.

```
DEB> LBREAKPOINT /ALL
Break set at MAIN\95 (count = 2)
Break set at READ_INPUT\%ENTRY (count = 1) [E maxv]
```

In this example, only the breakpoint at line 95 is listed.

```
DEB> LBREAKPOINT 95
Break set at MAIN\95 (count = 2)
```

Use

Lists CATCH settings for a given signal.

Format

LCATCH [/**ALL** | < *signal* >]

where:

signal can be either a signal name mnemonic (such as SIGALRM, SIGINT, and so on) or the signal number corresponding to that signal. Signal names may be specified in either uppercase or lowercase.

Description

The LCATCH command with no arguments lists all the signals which are currently set to be caught. When used with the /ALL option, LCATCH lists two sets of signals, those which are currently set to be caught and those which are currently set to not be caught. The LCATCH command, with a signal name specified, lists CATCH settings for that given signal.

Refer to the CATCH and NCATCH commands for more information.

Examples

In the following example, the current CATCH settings for the signal SIGIO are listed using the /LCATCH command.

```
DEB> LCATCH SIGIO
SIGIO being caught and not ignored.
```

In this example, the signals currently set to be caught and those that are set to not be caught are listed.

```
DEB> LCATCH /ALL
CATCHing signals:
SIGINT SIGQUIT SIGILL SIGTRAP SIGIOT SIGBUS
SIGTTOU SIGSYS SIGPIPE SIGTERM SIGURG SIGSTOP
SIGTTIN SIGIO SIGXCPU SIGXFSZ SIGVTALRM SIGPROF
SIGSEGV

Not CATCHing signals:
SIGHUP SIGEMT SIGFPE SIGKILL SIGALRM SITSTP
SIGCONT SIGCHLD SIGWINCH SIGLOST SIGUSR1
SIGUSR2
```

LENVIRONMENT LENV

Use

Lists the current evaluation environment.

Format

LENVIRONMENT /ALL

Description

The LENVIRONMENT command lists the current evaluation environment.

The /ALL option lists all environments within the source file.

Refer to the earlier section describing the ENVIRONMENT command.

Example

In this example, the current evaluation environment displayed is MAIN.

```
DEB> LENVIRONMENT
Evaluation environment is MAIN:(inactive)
```

Use

Assigns a value of an expression to a name.

Format

LET *name* = *expression*

where:

name is the name of any variable or destination construct (such as a substring or based reference in PL/I) that occurs in the source language program. *expression* can be any expression allowed by the source language whose value can be converted to the data type of the named variable.

Description

The LET command assigns a value of an expression to a name.

When the expression is evaluated, appropriate type conversions are performed. The resultant value is assigned to the named variable. If the type conversion is illegal then the debugger will issue an error message.

The value of a character string constant can be changed by enclosing the expression within single or double quotes depending on the source language.

Refer to Chapter 6, "Debugging LPI-C Programs," for specific information on assigning values to variables in C.

Example

In this example, the value of maxv is assigned a new value (maxv/2).

```
DEB> LET maxv = maxv/2
```

Use

Lists the macro definitions.

Format

LMACRO { *macro1, macro2, ..., macron* | **/ALL** }

Description

The LMACRO command lists the definition of one or more specified macros or the definitions for all currently defined macros.

The debugger lists the action list that was defined for the macro.

Either the indicated macro-names or /ALL must be specified.

Example

In this example, the definition of the macro named top is listed.

```
DEB> LMACRO  top
  top = [POINT 1; PRINT 5]
```

Use

Logs debugger commands to a specified file.

Format

LOG *file-name*

where:

file-name is the pathname of any file allowing write permission to the user.

Description

The LOG command causes all subsequent debugger commands to be appended to a specified file. See also the descriptions of the NLOG and READ commands.

To log commands to a new file after initiallly logging commands to *file-name*, use either NLOG followed by LOG with a new file-name or simply use LOG with a new file-name.

Example

```
DEB> LOG debug.history
Logging debugger commands to "debug.history"
DEB> FIND do while
99:    do while (n > 1)
DEB> PRINT 4
 99:    do while (n > 1)
100:              call sift(n);
101:              call read_input(n);
102:    end;
DEB> LBREAKPOINT /ALL
Break set at PRIMES\95 (count = 1)
Break set at PRIMES.SIFT\74 (count = 14)
DEB> CONTINUE
Break set at PRIMES.SIFT\74
```

LRETURN LRET

Use

Lists the return value of a function.

Format

LRETURN

Description

When the current execution point is at the exit point of a function, the LRETURN command prints the value to be returned by the function.

Example

In this example, the return value for the isprime function is listed.

```
DEB> LRETURN
Return value for PRIMES.ISPRIME is    15 {INTEGER*4}
```

Use

Prints the name of the current source file being displayed.

Format

LSOURCE < /**ALL** > < /**FULL** >

Description

The LSOURCE command prints the name of the source file currently being displayed. If the source file being displayed does not contain the current evaluation environment, LSOURCE will also display the source file which does contains the current evaluation environment.

If the /FULL option is specified, the name of the symbol table file associated with the source file containing the current evaluation environment will also be displayed.

If the /ALL option is specified, all the source files in the program being debugged, which were compiled in debug mode, will be displayed.

Example

In this example, the name of the symbol table file associated with the source file containing the current evaluation environment, the source file, and the source file currently being displayed are listed.

```
DEB> LSOURCE /FULL
Current display source file is "mpcalc.cob"  (COBOL)
Symbol table file is "mpcalc.stb"
```

Use

Lists the current stepping mode and the default action list.

Format

LSTEP

Description

The LSTEP command lists the current mode of stepping, that is, either IN or OVER, and the default action list.

Example

In this example, the default stepping mode is IN and the action list contains the PRINT, TRACE ENTRY, and CONTINUE commands.

```
DEB> LSTEP
Step IN; Action = [PRINT; TRACE ENTRY; CONTINUE]
```

Use

Lists information on watchpoints.

Format

LWATCH [< *variable* > | /ALL]

where:

variable specifies the variable to be watched.

Description

The LWATCH command lists information on the given watchpoint. If the /ALL option is specified, information is listed on all watchpoints. The information listed includes the skip count, execution count (the number of times the watchpoint has been incurred), specified switch settings, and any specified action list. Refer to the WATCH and NWATCH commands for more information.

Examples

In the following example, all of the existing watchpoints are listed.

```
DEB> LWATCH /ALL
Watchpoint set for MP /NSILENT /NIGNORE
Watchpoint set for LOAN /SILENT /NIGNORE
Watchpoint set for MONTHLY-RATE /NSILENT /IGNORE
Watchpoint set for CURRENT-BALANCE /SILENT /IGNORE
```

In this example, only the information concerning the LOAN watchpoint is listed.

```
DEB> LWATCH LOAN
Watchpoint set for LOAN /SILENT /NIGNORE
```

Use

Defines a macro.

Format

MACRO *macro-name* = [*action list*]

Description

The MACRO command is used to define a macro as shorthand for an action list. Sixteen macros can be defined at any one time.

Note
Do not choose a *macro-name* which is the same as a debugger command name because the *macro-name* will override the debugger command.

The *action list* is specified in the usual way (see Chapter 2, "Using Action Lists").

Once a macro has been defined, its name may be used just like any other debugger command name. The macro in the command line is replaced by the debugger commands specified in the action list. Those commands are then executed normally.

If the last (or only) command in the action list does not end with a semicolon, any additional arguments to that command can appear on a command line after the macro name. For example, suppose a macro

FOO = [LET A = A +]

has been defined, the command 'FOO 3' can then be entered to add 3 to A.

Example

In this example, the macro top is defined to point to the first line of the program and print 5 lines. The macro is then used.

```
DEB> MACRO top = [POINT 1; PRINT 5]
DEB> top
  1:
  1:
  2: PROGRAM main;
  3:
  4:   CONST
  5:     max_value = 1000;
```

NBREAKPOINT NB

Use

Removes breakpoints.

Format

NBREAKPOINT < *statement-id* | /ALL >

where:

statement-id is used to name the statement at which the breakpoint is to be removed.

Description

The NBREAKPOINT command removes the breakpoint at the named statement.

If *statement-id* is omitted, the breakpoint at the current execution point is removed. If /ALL is specified, all breakpoints currently set are removed.

Example

In this example, the breakpoint at line 74 is removed.

```
DEB> NBREAKPOINT 74
```

In this example, all breakpoints are removed.

```
DEB> NBREAKPOINT /ALL
```

Use

Prevents process control from returning to the debugger when the given signal is generated.

Format

NCATCH [/ALL | < *signal* >]

where:

signal can be either a signal name mnemonic (such as SIGALRM, SIGINT, and so on) or the signal number corresponding to that signal. Signal names may be specified in either uppercase or lowercase.

Description

The NCATCH command prevents process control from returning to the debugger when the given signal is generated. The child (user) process continues without interruption, as if the given signal had been generated. This implies that user-defined handlers will be executed. Refer to the CATCH and LCATCH commands for more information.

Example

In the following example, the process control will not return from the child process to the debugger when a SIGSEGV is generated.

```
DEB> NCATCH SIGSEGV
Changing SIGSEGV to now not be caught.
```

As a result, NCATCH allows the child process to continue uninterrupted when a SIGSEGV is generated. If the user process has its own signal handler for a SIGSEGV, then the signal handler will be executed.

Use

Stops the logging of debugger commands to a file.

Format

NLOG

Description

The LOG command ceases the logging of debugger commands to a file specified in the previous LOG command.

See also the description of the LOG command.

Example

```
DEB> LOG debug.history
Logging debugger commands to "debug.history"
DEB> FIND do while
99:    do while (n > 1)
DEB> PRINT 4
 99:    do while (n > 1)
100:              call sift(n);
101:              call read_input(n);
102:    end;
DEB> LBREAKPOINT /ALL
Break set at PRIMES\95 (count = 1)
Break set at PRIMES.SIFT\74 (count = 14)
DEB> CONTINUE
Break set at PRIMES.SIFT\74
DEB> NLOG
Ending commands logging to "debug.history"
```

Use

Removes macro definitions.

Format

NMACRO { *macro1, macro2, ..., macron* | /ALL }

Description

The NMACRO command removes the definition of one or more specified macros or the definitions of all currently defined macros.

Either the specific macro-name(s) or /ALL must be specified.

Example

In this example, the macro named top is removed.

```
DEB> NMACRO top
```

In this example, all macro definitions are removed.

```
DEB> NMACRO /ALL
```

Use

Disables entry or statement tracing.

Format

NTRACE { ENTRY | STATEMENT }

Description

The NTRACE command disables entry or statement tracing.

If ENTRY is specified, any action list and IF condition are removed as well.

Example

In this example, TRACE ENTRY and TRACE STATEMENT are disabled.

```
DEB> NTRACE ENTRY
DEB> NTRACE STATEMENT
```

Use

Removes watchpoints.

Format

NWATCH [< *variable* > | /**ALL**]

where:

variable specifies the variable being watched.

Description

The NWATCH command removes the watchpoint for the given variable. If the /ALL option is specified, all watchpoints currently set are removed.

Refer to the WATCH and LWATCH commands for more information.

Examples

In the following example, the watchpoint for the variable LOAN is removed from the current listing.

```
DEB> LWATCH /ALL
Watchpoint set for LOAN /SILENT /NIGNORE
Watchpoint set for MONTHLY-RATE /NSILENT /IGNORE
DEB> NWATCH LOAN
DEB> LWATCH /ALL
Watchpoint set for MONTHLY-RATE /NSILENT /IGNORE
```

In this example, all current watchpoints are removed.

```
DEB> LWATCH /ALL
Watchpoint set for CURRENT-BALANCE
Watchpoint set for MP /NSILENT /NIGNORE
Watchpoint set for MONTHLY-RATE /NSILENT /IGNORE
DEB> NWATCH /ALL
DEB> LWATCH /ALL
No watchpoints are currently set.
```

POINT PO

Use

Resets and displays the source file pointer within the current source file.

Format

POINT < +*n* | -*n* | *n* >

Description

The POINT command relocates the source file pointer within the current source file and prints that line.

If a sign is specified, the debugger relocates the source file pointer to the line which is *n* lines before (-) or after (+) the current source file pointer. Otherwise, the debugger relocates the source file pointer to the line specified by *n* and prints that line. If *n* is omitted, the source file pointer is reset to the current execution point.

Example

In this example, the source file pointer is reset to line 25 and displayed.

```
DEB> POINT 25
  25:          05 FILLER PIC X(8) VALUE SPACES.
```

In this example, the source file line three lines before the current source file pointer is displayed.

```
DEB> PO -3
  22:          05 H2 PIC X(8) VALUE "INTEREST".
```

PRINT P

Use

Prints a specified number of lines from the current source file.

Format

PRINT < *n* >

where:

n indicates the number of source lines to be printed.

Description

The PRINT command prints a specified number of source lines from the current debug listing file.

Printing starts at the current line. The current source line is reset to the last line printed. If argument *n* is omitted, only the current source line is printed.

Example

In this example, 10 lines of the Primes program are printed.

```
DEB> PRINT 10
  1:  REM  ** Primes **
  2:
  3:   DEF read_input
  4:   INTEGER read_input, maxv
  5:  10       INPUT " Input maximum prime boundary "; maxv
  6:           IF maxv <= max_value THEN GOTO 12
  7:   PRINT   toobig$ : GOTO 10
  8:  12       read_input = maxv
  9:   FEND
 10:
```

QUIT Q

Use

Terminates a debugging session.

Format

QUIT

Description

The QUIT command causes termination of the debugging session.

Example

This example shows the message the debugger displays as it exits.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Use

Reads and executes debugger commands from a specified file.

Format

READ *file-name*

where:

file-name is any file containing debugger commands. If the file is not in the current working directory, its pathname must be specified.

Description

The READ command is used to execute debugger commands which are contained in *file-name*. Each command is printed to the terminal as it is executed.

Seven levels of READ commands may be nested in debugger command files.

See also the descriptions of the LOG and NLOG commands.

Example

```
DEB> READ debugger.history
Reading debugger commands from "debug.history" ...
DEB> FIND do while
99:    do while (n > 1)
DEB> PRINT 4
 99:    do while (n > 1)
100:              call sift(n);
101:              call read_input(n);
102:    end;
DEB> LBREAKPOINT /ALL
Break set at PRIMES\95 (count = 1)
Break set at PRIMES.SIFT\74 (count = 14)
DEB> CONTINUE
Break set at PRIMES.SIFT\74
DEB> NLOG
Done reading commands from "debug.history"
```

Use

Reloads the user program.

Format

RELOAD < /ARGUMENTS *command-line-arguments* >

Description

The RELOAD command will reinitialize the user program while preserving any explicitly set breakpoints and any associated action lists. All breakpoint counters will be reset to zero.

A subsequent CONTINUE or STEP command will execute the user program from the beginning.

If the /ARGUMENTS option is not given then the program will be initialized with the command line arguments specified when CodeWatch was invoked. If /ARGUMENTS is specified, the arguments which follow the option will be used as command line arguments to the reinitialized program. If more than one argument is specified, they must be separated by a space.

Example

In this example, the debugging session is restarted in the original evaluation environment, with all breakpoints preserved, and without returning to the system level.

```
      .
      .
      .

**** sift\%ENTRY
Break at sift\%57
DEB> RELOAD
Reloading...
Evaluation environment is primes:(inactive)
DEB>
```

Use

Transfers the execution pointer to the exit point of the current procedure and allows a return value to be set.

Format

RETURN < *expression* >

Description

The RETURN command transfers the current execution pointer to the exit point (%EXIT) of the current procedure. If the procedure returns a value than an expression indicating the return value must be given. An expression is not given if the procedure does not return a value.

If the current execution point is already at the exit point of the current procedure, no action is taken if the procedure does not return a value, or the return value is set to *expression* if the procedure does return a value.

See also the description of the LRETURN command.

Example

In this example, the return value of the isprime routine is set to 15.

```
      .
      .
      .
Break at PRIMES.ISPRIME\%ENTRY
DEB> RETURN 15
```

Use

Changes the current source file to be displayed.

Format

SOURCE < *file-name* >

where:

file-name is the name of the source file to be displayed.

Description

The SOURCE command changes the source file to be displayed. The SOURCE command is used, for instance, when you want to look at the contents of "COPY" or "INCLUDE" files. Referencing these files is discussed in Chapter 3.

If *file-name* is omitted, the current source file is set to the source file associated with the current execution point. The current source file pointer is not explicitly set. To reset the current source file pointer to the curent execution pointer use the ENVIRONMENT command without any arguments.

The SOURCE command applies to the current debugger action only. Its effect is lost when the program resumes execution, that is, the current source file is set to the source file associated with the current execution point.

Example

In this example, the execution point is printed, which is line 2 of the main module of the amort program. The source file to be displayed is changed to mpcalc.cob and two lines are printed. The debugger returns to the location of the execution pointer when the SOURCE command is used without an argument. The execution point, which remains line 2 of amort, is printed.

```
DEB> PRINT
  2:      PROGRAM-ID. AMORT
DEB> SOURCE mpcalc.cob
DEB> P 2
  1:      IDENTIFICATION DIVISION
  2:      PROGRAM-ID. MPCALC.
DEB> SO
DEB> P
  2:      PROGRAM-ID. AMORT
```

Use

Displays information on a specified number of stack frames.

Format

STACK < *nframes* | /**ALL** > < /**ARGUMENTS** >
< /**LOCALS** >

where:

nframes is an integer specifying the count of the most recent stack frames to be displayed.

Description

The STACK command is used to display information on a specified number of stack frames.

If the /ALL option is specified, all stack frames back to the outermost procedure are displayed. There may be a number of "invisible" stack frames below the user's main program which are of no consequence. If neither *nframes* nor /ALL is specified, then only the current stack frame is displayed.

If the /ARGUMENTS option is specified, the arguments to each procedure are displayed. If /LOCALS is specified, all local variables for each stack frame are displayed.

Example

In this example, information is displayed on the current stack frame and the arguments to the procedure.

```
DEB> STACK /ARG
Stack contains 4 frames.
Current execution point is MPCALC\%EXIT

  4: Owner is "MPCALC"
    Arguments:
      LOAN =        6000.00 {right overpunch (10)}
      TERM =         4 {right overpunch (4)}
      RATE =       0.120000 {right overpunch (10)}
      MP =       158.00 {right overpunch (8)}
    Called from AMORT\78
```

Use

Executes a specified number of statements.

Format

STEP < OUT > < IN | OVER > < *count* > < [*action list*] >

where:

count is an integer specifying the number of statements to be executed.

Description

The STEP command starts program execution at the current location of the execution pointer and stops execution after one or a specified number of statements.

- To execute a single statement, use the STEP command with no options specified. To execute a number of statements, use the STEP command with *count* number of statements specified.

- If OUT is specified, the debugger steps to the exit point of the current routine. If OUT is specified with a *count*, the first step is the exit point of the current routine and each step thereafter is to a succeeding statement in the calling routine. Thus, if the execution pointer is in a called routine and you want to step to the next executable statement in the calling routine, issue STEP OUT 2. The first step is to the exit point of the current routine and the second step returns and steps forward in the calling routine. If the execution pointer is already at the exit point of the called routine, issuing STEP OUT causes the execution pointer to step to the exit point of the calling routine.
- If IN is specified and the current execution point is at a subroutine call, the debugger steps to the entry point of the called routine. If IN with a *count* is specified, the entry point is the first step, the first executable statement is another, and so on. If OVER is specified, the debugger steps over calls to routines. OVER is the default mode for the STEP command.
- If an *action list* is present, it is performed at the last step and is set as the action to be taken for any other STEP command. To remove an action list at STEP, supply a null action list as an argument to the STEP command. Use the DSTEP command to do this without stepping.
- At the completion of a step operation, the debugger issues a message indicating the current execution point in the following format:

  Step at *x*

  where *x* is the location of the current execution point, that is, the statement that is about to be executed.

Note
When using CodeWatch the STEP command cannot be invoked when the current execution point of the user program is at a non-local goto.

Example

In this example, the default stepping mode is listed as IN with an action list containing the PRINT command. Two step commands are executed (the second command with a count of 2), followed by a STEP OUT command.

```
DEB> LSTEP
Step IN; Action = [PRINT;]
DEB> STEP
Step at print_out\40
  40:    printf (" Number of primes found was %d\n\n",total);
DEB>  STEP 2
Number of primes found was 5

Step at print_out\42
  42:       printf ("%7d",values[i]);
DEB> STEP OUT
     1     2     3     5     7

Step at print_out\%EXIT
  47: }
```

TRACE ENTRY TR E

Use

Displays all procedure and block entries information during program execution.

Format

TRACE ENTRY < [*action list*] > < /IF=*logical-expr* >

Description

Tracing enables information about all procedure and block entries to be reported as the program executes.

The TRACE ENTRY command sets tracepoints which enable tracing of all procedure and block entries by printing a message each time a procedure or block is entered.

When entry tracing is enabled, the debugger will be activated as each procedure or block is entered, and activated again as each procedure or block is exited. The specified action list applies only to the entry tracepoint; exit tracepoints always print a message and continue.

If *action list* is specified, the debugger will process the commands and print the results. If *action list* is omitted, a default action list of [CONTINUE] is supplied. A common *action list* is [ARGUMENTS; CONTINUE]. An action list may not refer to statements or variables in a way that is dependent on scope. To take a breakpoint and enter interactive debugging at each entry, supply an explicit null action list, that is, TRACE ENTRY [].

Example

In this example, entry tracing is turned on and program execution continues. The read_input procedure is entered and the program is waiting for input from the user.

```
DEB> TRACE ENTRY
DEB> CONTINUE
****read_input\%ENTRY
Input maximum prime boundary:
```

TRACE STATEMENT TR S

Use

Sets tracepoints.

Format

TRACE STATEMENT

Description

The TRACE STATEMENT command sets tracepoints which enable the tracing of every statement by printing a message identifying the statement.

At each statement, if there is no other action to be performed (if there is no breakpoint, entry trace, single step action, and so on), the identifying message is printed and execution continues.

Example

In this example, statement tracing is turned on and program execution is continued. Statements are traced until the breakpoint at line 28 is encountered.

```
DEB> TRACE STATEMENT
DEB> CONTINUE
****MPCALC\24
****MPCALC\26
****MPCALC\27
Break at MPCALC\28
```

Use

Displays the resultant type of an expression.

Format

TYPE <**/FULL**> *expression*

where:

expression can be any expression that occurs in the source language program.

Description

The TYPE command prints the resultant type of an expression.

If *expression* refers to a structure or record, then the /FULL option can be used to display the entire structure or record, when debugging an LPI-COBOL, LPI-PASCAL, or LPI-PL/I program.

Example

In this example, type information is displayed on the array primes.

```
DEB> TYPE primes
   auto  int primes[500]
```

In this example, type information is displayed on the elements of the array primes.

```
DEB> TYPE primes[count]
     int
```

Use

Sets watchpoints on specified variables.

Format

WATCH < *variable* > [**/SILENT** | **/NSILENT**]
[**/IGNORE** | **NIGNORE**] [**/SKIP**=*n*]
[*action list* | **/ACTION** [*action list*]] [**/IF**=*logical-expr*]

where:

variable specifies the variable to be watched.

Description

Watchpoints are used to monitor any changes to data. The WATCH command designates the given variable to be watched. When the contents of the watched variable change, program execution stops and control is returned to the debugger. The maximum number of watchpoints that can be active at one time is 64.

A watchpoint counter is associated with each watchpoint. When the watchpoint is created, the counter is set to zero; each time the watchpoint is incurred, the counter is incremented by one. When a program is reloaded, all counters are set to zero.

The /SILENT option will not output modified variables when a watchpoint is encountered; the output simply reports that the program has stopped due to a watchpoint. The /NSILENT option allows the debugger to output the modified variable, the location of the change, and the old and new values of the watched variable.

The /IGNORE option allows the debugger to disregard a watched variable. In other words, when the given watchpoint is encountered, it will not be reported and the program will continue without interruption. The /NIGNORE option allows the debugger to recognize and report any encounter with the given watchpoint.

The /SKIP=*n* option assigns a watchpoint skip counter to the given variable, which is set to the value specified by *n*. As a result, the watchpoint will be skipped the specified (*n*) number of times. Once the watchpoint skip counter is set, it remains in effect until the value decreases (resets) to zero.

An *action list* is one or more of the debugger commands separated by a semicolon, which may specify that program execution is to continue. Any specified action list is executed when a watchpoint is incurred.

Note
The /ACTION [*action list*] syntax is necessary to avoid ambiguity with array subscripts in C and Pascal, but may also be used for other languages.

An IF condition may be used to qualify the watchpoint; it must be the last option on the command line. When a watchpoint with the IF option is set, the executing program will be suspended and the debugger will be activated whenever the watched variable is changed. The IF condition is then evaluated. The evaluation always takes place in the environment and language of the program statement. If the IF condition is false,

program execution continues and the watchpoint is never reported. If the IF condition is true, the debugger incurs a watchpoint as usual.

A watchpoint is incurred when a watched variable has changed, the skip counter is either zero or not specified, and the /IGNORE option is not specified. When a watchpoint is encountered, the following occurs:

- If an IF condition is specified, it is evaluated.
- The watchpoint counter is incremented.
- If the /IGNORE option is set for the given watchpoint, execution continues.
- If the skip count for this watchpoint is non-zero, the count is decremented and execution is continued; the skip count for an ignored watchpoint is not affected.
- The debugger announces the occurrence of the watchpoint. Output from the WATCH command includes the block name, the activation number, the line number where the change occurred, and the new and old hexadecimal values of the variable.
- If an action list is specified, it is executed.
- The debugger prompt appears on the screen.

Refer to the LWATCH and NWATCH commands for more information.

Example

In the following example, LWATCH lists information concerning the given watchpoint. The /NSILENT and /NIGNORE options allow the debugger to output the occurrence of a watchpoint and any modifications to that watchpoint.

```
DEB> WATCH MONTHLY-RATE
DEB> LWATCH MONTHLY-RATE
Watchpoint set for MONTHLY-RATE /NSILENT /NIGNORE
DEB> CONTINUE
Program stopped due to a watchpoint.
Watched variable MONTHLY-RATE modified about line
  AMORT\84

Old Value:  00 00 00 00 00 00
New Value:  00 00 00 00 15 0C
```

Use

Displays either the current execution point or the location of a specified statement.

Format

WHERE < *statement-id* >

where:

statement-id is a statement label or a source line number

Description

The WHERE command prints a location as the name of a routine followed by a statement label.

If *statement-id* is a statement label, then the name of the routine containing the label and the source line number of the label are printed. If the *statement-id* is a source line number, then the name of the routine is printed. If the *statement-id* argument is omitted, then the name of the routine and the line number of the current execution point are printed.

Example

In this example, the location of the statement label, PAGEHEADER, is displayed.

```
DEB> WHERE PAGEHEADER
AMORT\108
```

In this example, the current execution point is displayed.

```
DEB> WHERE
Current execution point is AMORT\100
```

In this example, the location of the ISPRIME routine entry point is displayed.

```
DEB> WHERE ISPRIME\%ENTRY
ISPRIME\%ENTRY (line 36)
```

Use

Invokes the host operating system command interpreter to perform a specified command.

Format

! *command*

Description

The ! command invokes the command interpreter to read the remainder of the line following the ! command. Refer to the *System V Programmer's Manual* for information on UNIX commands.

The ! command followed by the UNIX sh command pushes to a new command interpreter. To return to the debugger press <ctrl> <d>.

Note
This command is only available on machines using the UNIX or XENIX operating system.

Example

In this example, the UNIX pwd command is invoked, which lists the current working directory.

```
DEB> !pwd
/usr/mary/pl1progs
```

Chapter 5: Debugging LPI-BASIC Programs

Chapter 5: Debugging LPI-BASIC Programs

Specific Ways to Use CodeWatch Features

This section describes CodeWatch features that relate to debugging LPI-BASIC programs. The second section contains a sample debugging session of an LPI-BASIC program. The third section contains the listing of the program used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. A LPI-BASIC program block is a main program or subroutine or function subroutine. The main program or subroutine is referred to by the name of that program or subroutine.

Built-in Function Support

The following BASIC built-in functions are supported by CodeWatch:

ABS	CHR$	INT	LOG	SGN	STR$	VAL
ASC	COS	LEFT$	MID$	SIN	STRING$	
ATN	EXP	LEN	RIGHT$	SQR	TAN	

The following CBASIC specific built-in functions are supported by CodeWatch:

FLOAT	MATCH	SADD	UCASE$
INT%	MOD	SHIFT	

The following MBASIC specific built-in functions are supported by CodeWatch:

CINT	CVI	HEX$	MKS$
CSNG	CVS	INSTR	OCT$
CSRLIN	DATE$	MKD$	SPACE$
CVD	FIX	MKI$	TIME$

Referencing Arrays and Aggregate Structures

References to both arrays and array elements are allowed in the TYPE command only. For example,

TYPE A(10)	• refers to element 1 of array A
TYPE A()	• refers to array A

Slices of arrays cannot be referenced.

Sample CodeWatch Session Using CodeWatch

This debugging session illustrates how to use the commands and features of CodeWatch when debugging LPI-BASIC programs. Following the session is the source listing of the sample CBASIC program.

The sample program, primes.cbas, calculates the number of prime numbers within a given range. The program has been compiled using the -deb option to produce the necessary information for the debugger and the -l option to produce a listing file, the -cbext option to enable the CBASIC extensions, and has been linked. For example,

```
lpibasic primes.cbas -deb -l -cbext
lpild primes.o -o primes
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild primes.obj -o primes
```

In this sample session, comments (which are not part of the session) are the bulleted items. The system prompt is $. For clarity, the abbreviated form of the commands is used only after the command has been previously spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
$ codewatch primes
CodeWatch setting up "primes". Wait ...

************************************************************
* CodeWatch,  Revision 4.2.0                               *
* -------------------------                                *
*    Copyright(c) Language Processors, Inc. 1987           *
************************************************************
Evaluation environment is primes:(inactive)
```

- Find the main program.

```
DEB> FIND Main
  49  REM      ** Main Program **
```

- Print 20 lines.

```
DEB> PRINT 20
  76  REM      ** Main Program **
  77   INTEGER max_value, max_primes, n
  78  REM      REAL a
  79
  80   max_value   = 1000
  81   max_primes  = 500
  82   toobig$     = " Value too big.  Try again."
  83
  84   PRINT
  85   PRINT "*** Sieve of Eratosthenes ***"
  86   PRINT
  87
  88   n = read_input
  89
  90     40   IF n <= 1 THEN GOTO 50
  91   CALL sift(n)
  92   n = read_input
  93   GOTO 40
  94
  95     50   END
  96
```

- Set a breakpoint at line 91 of the program.

```
DEB> BREAKPOINT 91
```

- Enable entry tracing and begin program execution.

```
DEB> TRACE ENTRY
DEB> CONTINUE
**** primes\%ENTRY

 *** Sieve of Erathosthenes ***

**** primes.READ_INPUT\%ENTRY
 Input maximum prime boundary: 10
**** primes.READ_INPUT\%EXIT
Break at primes\91
```

- Set a breakpoint at line 71 of the SIFT routine and continue program execution.

```
DEB> B 71
DEB> C
**** primes.SIFT\%ENTRY
Break at primes.SIFT\71
```

- Evaluate data items in the SIFT routine.

```
DEB> EVALUATE i
      11  { integer }
DEB> E this_prime
      10  { integer }
```

- Evaluate the mod of this_prime and i.

```
DEB> E MOD(this_prime, i)
      10  {integer}
```

- Evaluate the floating point division of i by 3.

```
DEB> E i/3
      3.6666666666666E+000  {floating point decimal}
```

- Evaluate the integer division of i by 3.

```
DEB> E i\3
        3  {integer}
```

- Display type information about various data items.

```
DEB> TYPE count
{ <static> integer }
DEB> TY primes
{ <static array of> integer }
```

- Display all traceback information, including the arguments, to SIFT.

```
DEB> STACK /ARGUMENTS /ALL

Stack contains 4 frames.
Current execution point is primes.SIFT\71

   4: Owner is "primes.SIFT"
      Arguments:
         N =       10  { integer }
      Called from primes\in 91

   3: Owner is "primes"
      Arguments: None
      Main program
```

- Set the default stepping mode to step in and the stepping action list to print the current source line.

```
DEB> DSTEP IN [P]
```

- Step one statement.

```
DEB> STEP
Step at primes.PRINT_OUT\%ENTRY
32:         DEF print_out(total)
```

- Step out of the print_out routine.

```
DEB> STEP OUT
Number of primes found was  (prime)  5
     1     2     3     5     7
Step at primes.PRINT_OUT\%EXIT
  42:     FEND
```

- Set a breakpoint at the entry point of the read_input procedure with an action list evaluating the value of maxv and stepping one statement. Continue program execution.

```
DEB> B read_input\%ENTRY [E maxv, S]
DEB> C
**** primes.SIFT\%EXIT
Break at primes.read_input\%ENTRY
          10   { integer }
Step at primes.READ_INPUT\11
    11:
```

- Modify the value of maxv.

```
DEB> LET maxv = maxv\2
```

- Evaluate the character string, toobig$.

```
DEB> E toobig$
' Value too big.  Try again. '   {character string (27)}
```

- Modify the character string to read "large" instead of "big."

```
DEB> LET toobig$ = MID$(toobig$,1,11) + 'large' +
       MID$(toobig$,15,13)
```

- Evaluate the new value of toobig$.

```
DEB> E toobig$
' Value too large. Try again. '   {character string (29)}
```

- Go to line 13 and continue program execution using the new value of maxv and toobig$.

```
DEB> GOTO 13
Execution point is now primes.READ_INPUT\13
DEB>  C
**** primes.READ_INPUT\%EXIT
Break at primes\91
```

- Remove all previously set breakpoints, set a breakpoint at the entry point of the isprime procedure, and continue execution.

```
DEB> B isprime%\%entry
DEB> C
**** primes.SIFT\%ENTRY
**** primes.PRINT_OUT\%ENTRY

 Number of primes found was
Break at primes.ISPRIME\%ENTRY
```

- Set the return value of isprime to 15.

```
DEB> RETURN 15
```

- List the return value of isprime.

```
DEB> LRETURN
Return value for primes.ISPRIME is   15 {integer}
```

- Define a macro that removes all breakpoints, removes the default stepping action list, removes all macros, disables tracing, moves the current source file pointer to line 1 of the source program, and prints 5 lines.

```
DEB> MACRO refresh = [NB /ALL; DS []; NMA /A; NTR E;
POINT 1; P 5]
```

- Use the macro.

```
DEB> refresh

   1 REM      ** Sieve of Eratosthenes **
   1 REM      ** Sieve of Eratosthenes **
   2
   3  INTEGER primes(1), flags(1)
   4  DIM flags(100), primes(100)
   5
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: primes.cbas
Compiled: 01-Jan-85 12:00:01 by LPI-BASIC, Rev 1.00.00
Options: deb opt 2 l cbext


```
 1 REM      ** Sieve of Eratosthenes **
 2
 3  INTEGER primes(1), flags(1)
 4  DIM flags(100), primes(100)
 5
 6 REM      ** FUNCTION read_input **
 7
 8  DEF     read_input
 9
10  INTEGER read_input, maxv
11
12     10   INPUT " Input maximum prime boundary: "; maxv
13  IF maxv <= max_value THEN GOTO 12
14  PRINT toobig$ : GOTO 10
15     12   read_input = maxv
16  FEND
17
18 REM      ** Is Prime **
19
20  DEF isprime%(number)
21         INTEGER number, n
22               for n=1 to number
23                     if number = primes(n) then isprime = number:
24                     RETURN
25               NEXT n
26               isprime = -1
27               RETURN
28  FEND
29
30
31 REM      ** Print Out **
32
33  DEF print_out(total)
34         INTEGER total, i
35         PRINT "Number of primes found was ";
```

```
36          if isprime%(total) >= 0 then PRINT " (prime) ";
37          PRINT total:PRINT:PRINT
38          for i=1 to total
39                  PRINT primes(i);
40                  if MOD(i,10)=0 then PRINT
41          NEXT i
42          PRINT
43  FEND
44
45  REM       ** FUNCTION sift **
46
47  DEF       sift(n)
48
49  INTEGER n, i, k, count, flags(1), primes(1), this_prime
50
51  FOR i = 1 TO n
52          flags(i) = 0
53  NEXT i
54
55  count = 1
56  primes(1) = 1
57
58  FOR i = 2 TO n
59          IF flags(i) = 1 THEN goto 20
60
61          this_prime = i
62          count = count + 1
63          primes(count) = this_prime
64          k = i + this_prime
65
66  15              WHILE (k < n)
67          flags(k) = 1
68          k = k + this_prime
69          WEND
70      20   NEXT i
71
72   CALL print_out(count-1)
73   FEND
74
75
76
77  REM       ** Main Program **
78   INTEGER max_value, max_primes, n
79  REM       REAL a
```

```
80
81  max_value   = 1000
82  max_primes  = 500
83  toobig$     = " Value too big.  Try again."
84
85  PRINT
86  PRINT "*** Sieve of Eratosthenes ***"
87  PRINT
88
89  n = read_input
90
91    40  IF n <= 1 THEN GOTO 50
92  CALL sift(n)
93  n = read_input
94  GOTO 40
95
96    50  END
97
```

Chapter 6: Debugging LPI-C Programs

Chapter 6: Debugging LPI-C Programs

Specific Ways to Use CodeWatch Features

This section describes CodeWatch features that relate to debugging C programs. The second section contains a sample debugging session of an LPI-C program. The third section contains the listing of the program used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. A C program block is a function delimited by matching left and right braces, { and }. Function blocks are identified by the entry name. Functions in C cannot be nested.

Built-in Function Support

CodeWatch does not support any LPI-C built-in functions.

Referencing Arrays and Aggregate Structures

Arrays and aggregate structures cannot be referenced in their entirety, but only by fully qualified references to a structure member or array element. Slices of arrays cannot be referenced.

Modifying Variables

In C, assignments are merely expression operators with side effects. Therefore, the EVALUATE command provides the same functionality as the LET command in other languages. For example,

```
DEB> EVALUATE  total          • displays the value of total
       10  { int  }
```

```
DEB> E total = 5
       5   { int  }
```

- modifies and displays the value of total

```
DEB> E ++total
       6   { int  }
```

- increments and displays the value of total (prefix)

```
DEB> E total++
       6   { int  }
```

- displays and increments the value of total (postfix)

```
DEB> E total
       7   { int  }
```

Sample CodeWatch Session Using LPI-C

This debugging session illustrates how to use the commands and features of CodeWatch when debugging C programs. Following the session is the source listing of the sample program.

The sample program, primes.c, calculates the number of prime numbers within a given range. The program has been compiled using the -deb option to produce the necessary information for the debugger, and the -l option to produce a listing file. The program has also been linked. For example,

```
lpic primes.c -deb -l
lpild primes.o -o primes
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild primes.obj -o primes
```

In this sample session, explanatory comments (which are not part of the session) are the bulleted items. The system prompt is $. For clarity, the abbreviated form of the commands is used only after the command has been previously spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
$ codewatch primes
CodeWatch setting up "primes". Wait ...

************************************************************
* CodeWatch,  Revision 4.2.0                               *
* -------------------------                                *
*    Copyright(c) Language Processors, Inc. 1987           *
************************************************************
Evaluation environment is main:(inactive)
```

- Find the call to the read_input routine.

```
DEB> FIND read_input
  106:      read_input(&n);
```

- Print 6 lines.

```
DEB> PRINT 6
  106:   read_input (&n);
  107:
  108:   while (n > 1) {
  109:      sift (n);
  110:      read_input (&n);
  111:   }
```

- Set a breakpoint at line 108 of the main program.

```
DEB> BREAKPOINT 108
```

- Enable entry tracing and begin program execution.

```
DEB> TRACE ENTRY
DEB> CONTINUE
**** main_input\%ENTRY

 *** Sieve of Erathosthenes ***

**** read_input\%ENTRY
 Input maximum prime boundary: 10
**** read_input\%EXIT
Break at main\108
```

- Set a breakpoint at line 95 of the sift routine if count is greater than 0 and continue program execution.

```
DEB> B sift\95 /IF= count > 0
DEB> C
**** sift\%ENTRY
Break at sift\95
```

- Evaluate the value of count in the sift routine.

```
DEB> EVALUATE count
      5   { int  }
```

- Modify the value of count by an increment of 1.

```
DEB> E ++count
      6   { int  }
```

- Print the integer value of flag[0].

```
DEB> E (int)(*flags)
      -1   { int  }
```

- Evaluate a logical expression.

```
DEB> E flags[5] == *((flags) + (5))
 TRUE   { <logical>   }
```

- Use the logical AND, the NOT operator, and the MOD operator.

```
DEB> E 1 && 2
TRUE { <logical>  }
DEB> E !1
FALSE  { <logical>  }
DEB> E 7%3
         1  { int  }
```

- Use the bitwise AND, OR, NOT, and EXCLUSIVE OR operators.

```
DEB> E  0xAA &  (0xA0|0x9) ^ 0x7
         175  {int}
```

- Display declaration information about various data items.

```
DEB> TYPE primes
   auto  int primes[500]
DEB> TY flags
   auto  char flags[1000]
```

- Display stack frame information and the arguments to the sift procedure.

```
DEB> STACK /ARGUMENTS /ALL

Stack contains 4 frames.
Current execution point is sift\95

   4: Owner is "sift"
     Arguments:
                  10  { int  }
     Called from main\109

   3: Owner is "main"
     Arguments: None
     Main program
```

- Set the default stepping mode to step in and the stepping action list to print the current source line.

```
DEB> DSTEP IN [P]
```

- Step one statement.

```
DEB> STEP
Step at print_out\%ENTRY
  46: print_out(values,total)
```

- Step to next executable statement.

```
DEB> S
Step at print_out\52
  52:    printf (" Number of primes found was ");
```

- Step out of the print_out routine.

```
DEB> STEP OUT
Number of primes found was 5
     1     2     3     5     7

Step at print_out\%EXIT
```

- Set a breakpoint at the entry point of the read_input procedure with an action list evaluating the value of maxv. Continue program execution.

```
DEB> B read_input\%ENTRY [E *maxv]
DEB> C
**** sift\%EXIT
Break at read_input\%ENTRY
          10   { int   }
```

- Modify the value of maxv.

```
DEB> E *maxv = *maxv/2
           5   { int   }
```

- Change the current evaluation environment to primes and evaluate the new value of N. Continue program execution.

```
DEB> ENVIRONMENT main
DEB> E n
           5   { int   }
```

- Reset the current evaluation environment to the previous evaluation environment by using the ENVIRONMENT command without an argument.

```
DEB> ENV
Environment reset to read_input
```

- Go to line 21 and continue program execution.

```
DEB> GOTO 21
Execution point is now read_input\21
DEB>  C
 Input upper boundary: 10
**** read_input\%EXIT
Break at main\108
```

- Remove the breakpoint at line 108, set a new breakpoint at the entry point of the isprime procedure, and continue execution.

```
DEB> NB 108
DEB> B isprime\%entry
DEB> C
**** sift\%ENTRY
**** print_out\%ENTRY
 Number of primes found was Break at isprime\%ENTRY
```

- Set the return value of isprime to 15.

```
DEB> RETURN 15
```

- List the return value of isprime.

```
DEB> LRETURN
Return value for isprime is        15 { int }
```

- Define a macro that lists all breakpoints, the current evaluation environment, all macros, and the current execution point.

```
DEB> MACRO info = [LB /A; LENV; LMA /A; WH]
```

- Use the macro.

```
DEB> info
Break set at sift\95 /IF= count > 0 (count = 1)
Break set at isprime\%ENTRY (count = 1)
Break set at read_input\%ENTRY (count = 1) [E *maxv;]
Evaluation environment is isprime
  info = [LB /ALL; LENV; LMA /ALL; WHERE]
Current execution point is isprime\%EXIT
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: primes.c
Compiled: 1-Jan-85 12:00:01 by LPI-C, Rev 1.00.00
Options: deb opt 2 l


```
 1
 2  #include <stdio.h>
 3
 4  #define FALSE          (0)
 5  #define TRUE           (-1)
 6  #define MAX_VALUE      1000
 7  #define MAX_PRIMES     500
 8
 9  typedef int     bool;
10  typedef char    sieve[MAX_VALUE];
11  typedef int     results[MAX_PRIMES];
12
13  void
14  read_input(maxv)
15  int *maxv;
16  {
17          bool ok;
18
19          ok = FALSE;
20
21          do {
22                  printf (" Input maximum prime boundary: ");
23                  scanf ("%d",maxv);
24                  if (*maxv > MAX_VALUE)
25                          printf(" Value too big.  Try again.0);
26                  else
27                          ok = TRUE;
28          } while (!ok);
29  }
30
31  int
32  isprime (number,values,total)
33  int *values;
34  int total;
35  {
```

```
36          int i;
37
38          for (i = 0 ; i < total ; i++) {
39                  if (number == values[i])
40                          return (number);
41          }
42          return (-1);
43  }
44
45  void
46  print_out(values,total)
47  int *values;
48  int total;
49  {
50          int i;
51
52          printf (" Number of primes found was");
53          if (isprime(total,values,total) >= 0)
54                  printf (" (prime)");
55          printf (" %d",total);
56
57          printf ("0);
58
59          for (i = 0 ; i < total ; i++) {
60                  printf ("%7d",values[i]);
61                  if (((i + 1) % 10) == 0)
62                          printf("0);
63          }
64          printf ("0);
65  }
66
67  void
68  sift(n)
69  int n;
70  {
71          int i, k, count;
72          sieve flags;
73          results primes;
74          int this_prime;
75
76          for (i = 0 ; i < n ; i++)
77                  flags[i] = TRUE;
78
79          count = 0;
```

```
80          primes[0] = 1;
81
82          for (i = 0 ; i < n ; i++) {
83                  if (flags[i]) {
84                          this_prime = i + 2;
85                          count++;
86                          primes[count] = this_prime;
87                          k = i + this_prime;
88                          while (k < n) {
89                                  /* cancel all multiples */
90                                  flags[k] = FALSE;
91                                  k += this_prime;
92                          }
93                  }
94          }
95          print_out (primes,count);
96  }
97
98  main()
99  {
100         int n;
101
102         setbuf(stdout,NULL);
103
104         printf(" *** Sieve of Eratosthenes ***0);
105
106         read_input (&n);
107
108         while (n > 1) {
109                 sift (n);
110                 read_input(&n);
111         }
112 }
```

Chapter 7: Debugging LPI-COBOL Programs

Chapter 7: Debugging LPI-COBOL Programs

Specific Ways to Use CodeWatch Features

The first two sections of this chapter describe CodeWatch features that relate to debugging LPI-COBOL programs. The third section contains a sample debugging session of an LPI-COBOL program. The fourth section contains the listings of the programs used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. An LPI-COBOL program block is a complete paragraph whose name is the name given in the PROGRAM-ID PARAGRAPH.

Referencing Arrays and Aggregate Structures

Arrays or lists and aggregates or groups can be referenced in their entirety by referring to them by their list or group name, or individual subfields or members can be referenced using conventional COBOL syntax as in "P-NO IN PAGE-LINE". A subfield can be referenced directly as long as its name is unique. Slices of arrays can be referenced as described under the "EVALUATE" section in Chapter 4.

Procedure Division Paragraph-Names

PROCEDURE DIVISION paragraph-names can be referenced and are equivalent to statement labels in other languages; thus, paragraph-names serve debugging purposes in ways analogous to statement labels in other languages. For example,

```
DEB> GOTO PAGE-HEADER
```

Group Item Assignments

The LET command cannot be used to assign a string to a group item.

Representation of LPI-COBOL Data Types in CodeWatch

Table 7-1 lists the LPI-COBOL data types and the way that they are represented by CodeWatch. The debugger representation also contains a numerical value in parentheses that stands for the precision of the data item based on its data type. The LPI-COBOL data types are explained in the USAGE clause section in the *LPI-COBOL Language Reference Manual.*

TABLE 7-1 LPI-COBOL Data Types in CodeWatch

LPI-COBOL DATA TYPE	CODEWATCH REPRESENTATION
COMPUTATIONAL PIC S9(n)	computational
COMPUTATIONAL PIC 9(n)	computational unsigned
COMPUTATIONAL-1	float binary
COMPUTATIONAL-2	float binary
COMPUTATIONAL-3 PIC S9(n)	fixed decimal
COMPUTATIONAL-3 PIC 9(n)	fixed decimal unsigned
COMPUTATIONAL-6(n)	fixed decimal unsigned
DISPLAY PIC 9(n)	decimal unsigned
DISPLAY PIC S9(N) SIGN TRAILING	right overpunch
DISPLAY PIC S9(n) SIGN TRAILING SEPARATE	right separate

LPI-COBOL DATA TYPE	CODEWATCH REPRESENTATION
DISPLAY PIC S9(n) SIGN LEADING	left overpunch
DISPLAY PIC S9(n) SIGN LEADING SEPARATE	left separate
PIC S9(n)	right overpunch
PIC S9(n)V9(n)	right overpunch
PIC X(n)	character
PIC X(n) JUSTIFIED	character justified
PIC ZZZ99	picture
PIC XX/XX/XX	character pictured
INDEX	fixed binary

Sample CodeWatch Session Using LPI-COBOL

This debugging session illustrates how to use the commands and features of CodeWatch when debugging COBOL programs. Following the session are the source listings of the two programs used.

The main program AMORT calculates a loan amortization schedule and writes the results to the file OUTALL. AMORT also calls the subprogram MPCALC which computes the monthly payment given the amount of the loan, the interest rate, and the term of the loan (in years).

The programs have been separately compiled using the -deb option to produce the necessary information for the debugger, the -l option to produce a listing file, and have been linked into a single run unit called amort. For example,

```
lpicobol amort.cob -deb -l
lpicobol mpcalc.cob -deb -l
lpild amort.o mpcalc.o -o amort
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild amort.o mpcalc.o -o amort
```

In this sample session, comments (which are not part of the session) are the bulleted items. The system prompt is $. For clarity, the abbreviated form of the commands is used only after the command has been previously spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
$ codewatch amort

************************************************************
* CodeWatch,  Revision 4.2.0                               *
* -------------------------                                *
*    Copyright(c) Language Processors, Inc. 1987           *
************************************************************
Evaluation environment is AMORT:(inactive)
```

- Find where MPCALC is called.

```
DEB> FIND MPCALC
  78:     CALL "MPCALC" USING LOAN, TERM, RATE, MP.
```

- Set a breakpoint at the CALL to MPCALC.

```
DEB> BREAKPOINT 78
```

- Set a breakpoint at the exit point of MPCALC.

```
DEB> B MPCALC\%EXIT
```

- Enable entry tracing and display arguments, if any, and begin program execution.

```
DEB> TRACE ENTRY [ ARGUMENTS ]
DEB> CONTINUE
****AMORT\%ENTRY
No arguments
```

- Continue program execution.

```
DEB> C
ENTER LOAN AMOUNT: 6000.00
ENTER TERM IN YEARS: 4
ENTER INTEREST RATE: .12
Break at AMORT\78
```

- Print the present source line.

```
DEB> PRINT
    78:    CALL "MPCALC" USING LOAN, TERM, RATE, MP.
```

- Evaluate a data item before entering the called program.

```
DEB> EVALUATE LOAN
LOAN =          6000.00  {right overpunch (10)}
```

- Step into the called program.

```
DEB> STEP IN
Step at MPCALC\%ENTRY
```

- Step, print one line, and continue.

```
DEB> S; P; C
Step at MPCALC\23
    23:            S1 SECTION.
Break at MPCALC\%EXIT
```

- Display the data items specified in the USING phrase.

```
DEB> ARG MPCALC
LOAN =      6000.00  {right overpunch (10)}
TERM =         4     {right overpunch (4)}
RATE =     0.120000  {right overpunch (10)}
MP =        158.00   {right overpunch (8)}
```

- List the current evaluation environment.

```
DEB> LENVIRONMENT
Evaluation environment is MPCALC
```

- List the name of the current source file using the /FULL option to display the name of the symbol table file.

```
DEB> LSOURCE /FULL
Current display source file is "mpcalc.cob"  (COBOL)
Symbol table file is "mpcalc.stb"
```

- Set a breakpoint, enable statement tracing, and continue.

```
DEB> B 28
DEB> TRACE STATEMENT
DEB> C
****MPCALC\24
****MPCALC\26
****MPCALC\27
Break at MPCALC\28
```

- Display stack frame information on the called program.

```
DEB> STACK
Current execution point is MPCALC\28
   Stack contains 4 frames
   4: Owner is "MPCALC"
   Called from AMORT\78
```

- Set a breakpoint in the calling program and step out of the called program.

```
DEB> B AMORT\80
DEB> S OUT
Step at MPCALC\%EXIT
```

- Evaluate a data item that was passed to the calling program.

```
DEB> E MP
MP =       158.00  {right overpunch (8)}
```

- Disable statement tracing, and step and print one source line.

```
DEB> NTR S; S; P
Step at AMORT\79
   79:                OPEN OUTPUT OUT-FILE.
```

- Define and use a MACRO.

```
DEB> MACRO sp = [ s; p ]
DEB> sp
Step (and break) at AMORT\80
   80:         PERFORM PAGE-HEADER.
```

- Where is the paragraph-name PAGE-HEADER?

```
DEB> WHERE PAGE-HEADER
AMORT\108
```

- Point to the line containing PAGE-HEADER and print 5 source lines.

```
DEB> POINT 108
   108:      PAGE-HEADER.
DEB> P 5
   108:      PAGE-HEADER.
   109:       ADD 1 TO PAGE-COUNT.
   110:       MOVE PAGE-COUNT TO P-NO IN PAGE-LINE.
   111:       WRITE OUT-REC FROM PAGE-LINE AFTER
                   ADVANCING PAGE.
   112:       MOVE 0 TO LINE-COUNT.
```

- Where is the current execution point?

```
DEB> WH
Current execution point is AMORT\80
```

- Use the macro, set a breakpoint, and continue.

```
DEB> sp
Step at AMORT\108
108:            PAGE-HEADER.
DEB> B 112
DEB> C
Break at AMORT\112
```

- Evaluate 2 data items.

```
DEB> E PAGE-COUNT
PAGE-COUNT =       1        {computational (4)}
DEB> E OUT-REC
OUT-REC IN  IN OUT-FILE "                          PAGE 0001 "
      {character (70)}
```

- Set a breakpoint at source line 94 to evaluate CURRENT-BALANCE each time through the PERFORM loop and continue program execution.

```
DEB> B 94 [ E CURRENT-BALANCE ]
DEB> C
Break at AMORT\94
CURRENT-BALANCE =        6000.00  {fixed decimal (10,2)}
DEB> C
Break at AMORT\94
CURRENT-BALANCE =        5902.00  {fixed decimal (10,2)}
```

- Set another breakpoint with an action list.

```
DEB> B 106 [ E OUT-REC ]
DEB> C
Break at AMORT\106
OUT-REC IN IN OUT-FILE " 3  59.02  98.98  5803.02  119.02 "
         {character  (70)}
```

- Evaluate and modify the value of the data item MONTHLY-RATE, and continue.

```
DEB> E MONTHLY-RATE
MONTHLY-RATE =          0.010000  {fixed decimal} (10,6)}
DEB> LET MONTHLY-RATE = .015
DEB> C
Break at AMORT\94
CURRENT-BALANCE =          5803.02  {fixed decimal (10,2)}
DEB> C
Break at AMORT\106
OUT-REC IN IN OUT-FILE  " 2    87.05    70.95    5732.07    206.07 "
        {character  (70)}
```

- List all breakpoints.

```
DEB> LB /ALL
Break set at AMORT\79  (count = 1)
Break set at MPCALC\28  (count = 1)
Break set at AMORT\80  (count = 1)
Break set at AMORT\112  (count = 1)
Break set at AMORT\94  (count = 3)  [ E CURRENT-BALANCE; ]
Break set at AMORT\106  (count = 2)  [ E OUT-REC; ]
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: amort.cob
Compiled: 01-Jan-85 12:00:01 by LPI-COBOL, Rev 1.00.00
Options: deb opt 2 l

Compiler Runtime Library Products,
Copyright(c) Language Processors,Inc. 1985.

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. AMORT.
3   ENVIRONMENT DIVISION.
4   CONFIGURATION SECTION.
5    SOURCE-COMPUTER. XXX.
6    OBJECT-COMPUTER. XXX.
7   INPUT-OUTPUT SECTION.
8   FILE-CONTROL.
9    SELECT OUT-FILE
10     ASSIGN TO PRINTER
11     ORGANIZATION SEQUENTIAL.
12   DATA DIVISION.
13   FILE SECTION.
14   FD OUT-FILE
15    LABEL RECORDS STANDARD
16    VALUE OF FILE-ID IS OUT-FNAME.
17   01 OUT-REC PIC X(70).
18   WORKING-STORAGE SECTION.
19   01 HEADER-LINE.
20    05 H1        PIC X(7)       VALUE "PAYMENT".
21    05 FILLER    PIC X(4)         VALUE SPACES.
22    05 H2        PIC X(8)       VALUE "INTEREST".
23    05 FILLER    PIC X(6)         VALUE SPACES.
24    05 H3        PIC X(9)       VALUE "PRINCIPAL".
25    05 FILLER    PIC X(8)         VALUE SPACES.
26    05 H4        PIC X(7)       VALUE "BALANCE".
27    05 FILLER    PIC X(5)         VALUE SPACES.
28    05 H5        PIC X(12)      VALUE "TOT INTEREST".
29   01 PAGE-LINE.
30    05 FILLER PIC X(50) VALUE SPACES.
31    05 H1    PIC X(6)  VALUE "PAGE  ".
32    05 P-NO   PIC X(4).
33   01 DESCRIP-LINE.
34    05 FILLER PIC X(3) VALUE SPACES.
35    05 DESCRIP  PIC X(16).
36    05 FILLER   PIC X(2)          VALUE ": ".
37    05 NVAL     PIC Z(8).9(2).
38    05 TVAL     REDEFINES NVAL PIC Z(11).
39   01 DASH-LINE PIC X(70) VALUE ALL "--".
40   01 MONTHLY-RATE PIC S9(4)V9(6) COMP-3.
41   01 TERM-IN-MONTHS PIC S9(4) COMP-3.
42   01 LOAN PIC S9(8)V9(2).
```

```
43   01 TERM PIC S9(4).
44   01 RATE PIC S9(4)V9(6).
45   01 MP PIC S9(6)V9(2).
46   01 TWELVE PIC S9(4) COMP-3 VALUE 12.0.
47   01 C-ONE PIC S9(4) COMP-3 VALUE 1.0.
48   01 CURRENT-BALANCE PIC S9(8)V9(2) COMP-3.
49   01 THIS-INTEREST   PIC S9(8)V9(2) COMP-3.
50   01 THIS-PRINCIPAL  PIC S9(8)V9(2) COMP-3.
51   01 TOTAL-INTEREST  PIC S9(8)V9(2) COMP-3 VALUE 0.0.
52   01 PAYMENT-NUMBER  PIC S9(4) COMP VALUE 1.
53   01 OUT-LINE.
54    05 OUT-PAYMENT-NUMBER         PIC Z(3).
55    05 FILLER                     PIC X(4)  VALUE SPACES.
56    05 OUT-THIS-INTEREST          PIC Z(8).9(2).
57    05 FILLER                     PIC X(4)  VALUE SPACES.
58    05 OUT-THIS-PRINCIPAL         PIC Z(8).9(2).
59    05 FILLER                     PIC X(4)  VALUE SPACES.
60    05 OUT-CURRENT-BALANCE        PIC Z(8).9(2).
61    05 FILLER                     PIC X(4)  VALUE SPACES.
62    05 OUT-TOTAL-INTEREST         PIC Z(8).9(2).
63   77 PAGE-COUNT     PIC S9(4) COMP      VALUE 0.
64   77 LINE-COUNT     PIC S9(4) COMP      VALUE 0.
65   77 MAX-LINES      PIC S9(4) COMP      VALUE 50.
66   77 OUT-FNAME      PIC X(40)           VALUE "OUTALL".
67   PROCEDURE DIVISION.
68   S1 SECTION.
69   P1.

71    DISPLAY "ENTER LOAN AMOUNT: " NO ADVANCING.
72    ACCEPT LOAN.
73    DISPLAY "ENTER TERM IN YEARS: " NO ADVANCING.
74    ACCEPT TERM.
75    DISPLAY "ENTER INTEREST RATE: " NO ADVANCING.
76    ACCEPT RATE.
77
78    CALL "MPCALC" USING LOAN, TERM, RATE, MP.
79    OPEN OUTPUT OUT-FILE.
80    PERFORM PAGE-HEADER.
81    PERFORM WRITE-DESCRIP.
82    PERFORM DETAIL-HEADER.
83
84    COMPUTE MONTHLY-RATE = RATE / TWELVE.
85    COMPUTE TERM-IN-MONTHS = TERM * TWELVE.
86    MOVE LOAN TO CURRENT-BALANCE.
87    PERFORM ONE-PAYMENT-CALC TERM-IN-MONTHS TIMES.
88    STOP RUN.
89
90   ONE-PAYMENT-CALC.
91    COMPUTE THIS-INTEREST ROUNDED
92         = CURRENT-BALANCE * MONTHLY-RATE.
93    COMPUTE THIS-PRINCIPAL = MP - THIS-INTEREST.
94    COMPUTE CURRENT-BALANCE = CURRENT-BALANCE - THIS-PRINCIPAL.
95    COMPUTE TOTAL-INTEREST = TOTAL-INTEREST + THIS-INTEREST.
96    PERFORM WRITE-ONE-LINE.
97    ADD 1 TO PAYMENT-NUMBER.
```

```
99   WRITE-ONE-LINE.
100    MOVE PAYMENT-NUMBER TO OUT-PAYMENT-NUMBER.
101    MOVE THIS-INTEREST TO OUT-THIS-INTEREST.
102    MOVE THIS-PRINCIPAL TO OUT-THIS-PRINCIPAL.
103    MOVE CURRENT-BALANCE TO OUT-CURRENT-BALANCE.
104    MOVE TOTAL-INTEREST TO OUT-TOTAL-INTEREST.
105    MOVE OUT-LINE TO OUT-REC.
106    PERFORM WRITE-DETAIL.
107
108   PAGE-HEADER.
109    ADD 1 TO PAGE-COUNT.
110    MOVE PAGE-COUNT TO P-NO IN PAGE-LINE.
111    WRITE OUT-REC FROM PAGE-LINE AFTER ADVANCING PAGE.
112    MOVE 0 TO LINE-COUNT.
113
114   WRITE-DESCRIP.
115    MOVE "LOAN"              TO DESCRIP IN DESCRIP-LINE.
116    MOVE LOAN                TO NVAL IN DESCRIP-LINE.
117    WRITE OUT-REC FROM DESCRIP-LINE
118    AFTER ADVANCING 2.
119    MOVE "INTEREST RATE"     TO DESCRIP IN DESCRIP-LINE.
120    MOVE RATE                TO NVAL IN DESCRIP-LINE.
121    WRITE OUT-REC FROM DESCRIP-LINE.
122    MOVE "MONTHLY PAYMENT"   TO DESCRIP IN DESCRIP-LINE.
123    MOVE MP                  TO NVAL IN DESCRIP-LINE.
124    WRITE OUT-REC FROM DESCRIP-LINE.
125    MOVE "TERM (YEARS)"      TO DESCRIP IN DESCRIP-LINE.
126    MOVE TERM                TO TVAL IN DESCRIP-LINE.
127    WRITE OUT-REC FROM DESCRIP-LINE.
128    MOVE SPACES TO OUT-REC.
129    WRITE OUT-REC AFTER ADVANCING 2.
130    ADD 7 TO LINE-COUNT.
131
132   WRITE-DETAIL.
133    WRITE OUT-REC.
134    ADD 1 TO LINE-COUNT.
135    IF LINE-COUNT GREATER MAX-LINES
136      PERFORM PAGE-HEADER
137      PERFORM DETAIL-HEADER.
138
139   DETAIL-HEADER.
140     WRITE OUT-REC FROM HEADER-LINE AFTER ADVANCING 2.
141     WRITE OUT-REC FROM DASH-LINE.
142
```

Source File: mpcalc.cob
Compiled: 01-Jan-85 12:00:01 by LPI-COBOL, Rev 1.00.00
Options: deb opt 2 l

Compiler Runtime Library Products,
Copyright(c) Language Processors,Inc. 1986.

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. MPCALC.
3   ENVIRONMENT DIVISION.
4   CONFIGURATION SECTION.
5    SOURCE-COMPUTER. XXX.
6    OBJECT-COMPUTER. XXX.
7   DATA DIVISION.
8   WORKING-STORAGE SECTION.
9   01 TWELVE PIC S9(4) COMP-3 VALUE 12.0.
10  01 C-ONE PIC S9(4) COMP-3 VALUE 1.0.
11  01 FACTOR1 PIC S9(1)V9(8) COMP-3.
12  01 FACTOR2 PIC S9(4)V9(8) COMP-3.
13  01 FACTOR3 PIC S9(4)V9(8) COMP-3.
14  01 FACTOR4 PIC S9(2)V9(8) COMP-3.
15  01 FACTOR5 PIC S9(7)V9(7) COMP-3.
16  01 FLOAT2 COMP-2.
17  LINKAGE SECTION.
18  01 LOAN PIC S9(8)V9(2).
19  01 TERM PIC S9(4).
20  01 RATE PIC S9(4)V9(6).
21  01 MP PIC S9(6)V9(2).
22  PROCEDURE DIVISION USING LOAN, TERM, RATE, MP.
23  S1 SECTION.
24  P1.
25
26    COMPUTE FACTOR1 = RATE / TWELVE.
27    COMPUTE FLOAT2 = ( C-ONE + FACTOR1) ** ( TERM * TWELVE).
28    MOVE FLOAT2 TO FACTOR2.
29    COMPUTE FACTOR3 = FACTOR2 - C-ONE.
30    COMPUTE FACTOR4 = ( (FACTOR1 * FACTOR2) ) / FACTOR3.
31    COMPUTE MP = LOAN * FACTOR4.
32
33  P2.
34    EXIT PROGRAM.
35
```

Chapter 8: Debugging LPI-FORTRAN Programs

Chapter 8: Debugging LPI-FORTRAN Programs

Specific Ways to Use CodeWatch Features

This section describes CodeWatch features that relate to debugging FORTRAN programs. The second section contains a sample debugging session of an LPI-FORTRAN program. The third section contains the listing of the program used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. A block in FORTRAN is a main program or a subroutine or function subprogram. An unnamed main program block is referred to by the name MAIN. A subroutine or function subprogram is referred to by the external name of that subroutine or function.

Referencing Arrays and Aggregate Structures

To reference common block names in the TYPE command, the name must be surrounded by slashes (/). For example,

```
TYPE /common1/
```

Common block names cannot be referenced by the EVALUATE command. Individual array elements can be referenced, but not slices of arrays.

Built-in Function Support

All FORTRAN built-in functions including the scientific functions such as sine and cosine, are available when debugging a FORTRAN program. Refer to the *LPI-FORTRAN Language Reference Manual* for information on built-in functions.

Sample CodeWatch Session for LPI-FORTRAN Program

This debugging session illustrates how to use the commands and features of CodeWatch when debugging FORTRAN programs. The source listing of the sample program follows the session.

The sample program, primes.ftn, calculates the number of prime numbers within a given range. The program has been compiled using the -deb option to produce the necessary information for the debugger, the -l option to produce a listing file and has been linked. For example,

```
lpifortran primes.ftn -deb -l
lpild primes.o -o primes
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild primes.obj -o primes
```

In this sample session, comments (which are not part of the session) are the bulleted items. The system prompt is $. For clarity, the abbreviated form of the commands is used only after the command has been previously spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
$ codewatch primes
CodeWatch setting up "primes". Wait ...

************************************************************
* CodeWatch,  Revision 4.2.0                               *
* -------------------------                                *
*    Copyright(c) Language Processors, Inc. 1987           *
************************************************************
Evaluation environment is PRIMES:(inactive)
```

- Find the string "Main."

```
DEB> FIND Main
   6: *       Main Program
```

- Print 10 lines.

```
DEB> PRINT 10
   6  *      Main Program
   7  *      ============================
   8  *
   9  10     CALL READ_INPUT(N, MAX_PRIMES)
  10         IF (N .LE. 0) GOTO 20
  11         CALL SIFT(N)
  12         GOTO 10
  13  20     STOP
  14         END
  15  *
```

- Set a breakpoint at line 10.

```
DEB> BREAKPOINT 10
```

- Set a breakpoint at line 92 of the sift routine.

```
DEB> B 92
```

- Enable entry tracing and begin program execution.

```
DEB> TRACE ENTRY
DEB> CONTINUE
**** PRIMES\%ENTRY
**** READ_INPUT\%ENTRY
Input upper boundary:
10
**** READ_INPUT\%EXIT
Break at PRIMES\10
```

- Continue program execution.

```
DEB> C
**** SIFT\%ENTRY
Break at SIFT\92
```

- Evaluate data items in the sift routine.

```
DEB> EVALUATE count
      5 {INTEGER*2}
DEB> E flags(count)
      .TRUE. {LOGICAL*1}
```

- Evaluate a logical expression.

```
DEB> E flags(count) .AND. .FALSE.
      .FALSE. {LOGICAL*4}
```

- Evaluate an equivalenced integer to a logical value (represented by the variable LVALUE).

```
DEB> E LVALUE
      -1 {INTEGER*4}
```

- Display declaration information about various data items.

```
DEB> TYPE this_prime
THIS_PRIME
Class = variable    Size =  2     INTEGER*2 <auto>
DEB> TY primes
PRIMES
Class = array       Size =  2000  INTEGER*2 <auto>
```

- Display stack frame information.

```
DEB> STACK /ALL

Stack contains 4 frames.
Current execution point is SIFT\92

   4: Owner is "SIFT"
     Called from PRIMES\11

   3: Owner is "PRIMES"
     Main program
```

- Set a breakpoint in the print_out routine and continue program execution.

```
DEB> B 62
DEB> C
**** PRINT_OUT\%ENTRY
**** ISPRIME\%ENTRY
**** ISPRIME\%ENTRY
Number of primes found was 5 (prime)
Break at PRINT_OUT\62
```

- Evaluate the first 24 characters in the variable line.

```
DEB> E line(1:24)
'      1      2      3' {CHARACTER*24}
```

- Set the default stepping mode to step in and the stepping action list to print the current source line.

```
DEB> DSTEP IN [P]
```

- Step one statement.

```
DEB> STEP
      1      2      3      5      7
Step at PRINT_OUT\63
  63:         J = 1
```

- Step two statements.

```
DEB> S 2
Step at PRINT_OUT\65
  65: 40        CONTINUE
```

- Set a breakpoint at the entry point of the READ_INPUT procedure with an action list evaluating the value of MAXV. Continue program execution.

```
DEB> B read_input\%ENTRY [E MAXV]
DEB> C
**** PRINT_OUT\%EXIT
**** SIFT\%EXIT
Break at READ_INPUT\%ENTRY
         10 {INTEGER*4}
```

- Modify the value of MAXV.

```
DEB> LET MAXV = MAXV/2
```

- Change the current evaluation environment to primes and evaluate the new value of N.

```
DEB> ENVIRONMENT primes
DEB> E n
          5 {INTEGER*4}
```

- Remove all previously set breakpoints, set a breakpoint at the entry point of the isprime procedure, and continue execution.

```
DEB> NB /ALL
DEB> B isprime\%entry
DEB> C
Input upper boundary:
10
**** READ_INPUT\%EXIT
**** PRIMES.SIFT\%ENTRY
**** PRIMES.PRINT_OUT\%ENTRY
Break at PRIMES.ISPRIME\%ENTRY
```

- Set the return value of isprime to 15.

```
DEB> RETURN 15
```

- List the return value of isprime.

```
DEB> LRETURN
Return value for PRIMES.ISPRIME is          15 {INTEGER*4}
```

- Define a macro that lists all breakpoints, the current evaluation environment, all macros, the current execution point, and invokes the shell for a directory listing.

```
DEB> MACRO info = [LB /A; LENV; LMA /A; WHERE]
```

- Use the macro.

```
DEB> info
Break set at ISPRIME\%ENTRY (count = 1)
Evaluation environment is ISPRIME
  info = [LB /A; LENV; LMA /A; WHERE]
Current execution point is ISPRIME\%EXIT
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: primes.ftn
Compiled: 01-Jan-85 12:00:01 by LPI-FORTRAN, Rev 1.01.00
Options: deb opt 2 l

Compiler Runtime Library Products,
Copyright (c) Language Processors, Inc. 1985.

```
1  *
2         PROGRAM PRIMES
3  *
4         PARAMETER (MAX_PRIMES=1000)
5  *
6  *      Main Program
7  *      ============================
8  *
9  10     CALL READ_INPUT(N, MAX_PRIMES)
10        IF (N .LE. 0) GOTO 20
11        CALL SIFT(N)
12        GOTO 10
13 20      STOP
14        END
15 *
16 *      Read Input from Terminal
17 *      ============================
18 *
19        SUBROUTINE READ_INPUT(MAXV,MAX_VALUE)
20 30      PRINT *, ' Input upper boundary: '
21        READ *, MAXV
22        IF (MAXV .LE. MAX_VALUE) RETURN
23 *
24        PRINT *, ' Value too big. Try again. '
25        GOTO 30
26        END
27 *
28 *      Determine if the given number is prime
29 *      ============================
30 *
31        INTEGER*4 FUNCTION ISPRIME(NUMBER, VALUES, TOTAL)
32        INTEGER*2 NUMBER, VALUES, TOTAL
33        DIMENSION VALUES (MAX_PRIMES)
34        INTEGER*4  I
35        DO 35 I = 1, TOTAL
36           IF (NUMBER .EQ. VALUES(I)) THEN
37              ISPRIME = NUMBER
```

```
38              RETURN
39            END IF
40  35       CONTINUE
41           ISPRIME = -1
42           END
43  *
44  *        Print out Prime Numbers from Sieve
45  *        ==========================
46  *
47           SUBROUTINE PRINT_OUT(VALUES, TOTAL)
48           INTEGER*2 VALUES, TOTAL
49           DIMENSION VALUES (MAX_PRIMES)
50           CHARACTER*80 LINE
51           INTEGER*4   I, J
52           LINE = ""
53           IF (ISPRIME(TOTAL,VALUES,TOTAL) .GE. 0) THEN
54              LINE(1:8) = " (prime)"
55           END IF
56           PRINT *, " Number of primes found was ", TOTAL, LINE
57           J = 1
58           DO 40 I = 1, TOTAL
59              WRITE(LINE(J: J + 7), '(I8)') VALUES(I)
60              J = J + 8
61              IF ((MOD(I, 9) .EQ. 0) .OR. (I .EQ. TOTAL)) THEN
62                 PRINT *, LINE(1:J-1)
63                 J = 1
64              END IF
65  40       CONTINUE
66           END
67  *
68  *        Sift the Sieve and find prime numbers
69  *        ==========================
70  *
71           SUBROUTINE SIFT(N)
72           PARAMETER (MAX_VALUE=1000)
73           LOGICAL*1 FLAGS(0:MAX_VALUE)
74           EQUIVALENCE (LVALUE, FLAGS)
75           INTEGER*2 SIZE,I,K,THIS_PRIME,COUNT
76           INTEGER*2 PRIMES
77           DIMENSION PRIMES(MAX_VALUE)
78           DO 50 I = 0, N
79              FLAGS(I) = .TRUE.
80  50       CONTINUE
81           COUNT = 1
82           PRIMES(1) = 1
83           DO 90 I = 2, N
84              IF (FLAGS(I)) THEN
85                 COUNT = COUNT + 1
86                 PRIMES(COUNT) = I
```

```
87                DO 80 K = I*2, N, I
88                   FLAGS(K) = .FALSE.   ! Cancel all multiples
89  80            CONTINUE
90              END IF
91  90        CONTINUE
92            CALL PRINT_OUT(PRIMES, COUNT)
93            END
94
```

Chapter 9: Debugging LPI-PASCAL Programs

Chapter 9: Debugging LPI-PASCAL Programs

Specific Ways to Use CodeWatch Features

The first section describes CodeWatch features that relate to debugging LPI-PASCAL programs. The second section contains a sample debugging session of an LPI-PASCAL program. The third section contains the listing of the program used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. An LPI-PASCAL program block is a procedure block or a begin block.

Block Names

A procedure block is referred to by the name of that procedure.

Referencing Nested Blocks

LPI-PASCAL program blocks may be nested. Rules for naming nested blocks are as follows:

1. If a block defined within the compilation unit is contained within or contains the debugger's current evaluation environment, the block may be referred to simply by its name. The name may need to be qualified to make it unique within the external procedure.

   If the block contains the debugger's current evaluation environment, the search for the referenced block begins from the current environment and continues through each successive containing (parent) block until the referenced block is found.

2. If a block is in some other external procedure, it must be qualified with at least the external procedure name, and it may require further qualification. Furthermore, a fully qualified name to a block in some other external procedure may be the same as a partially qualified block name in the current procedure. To force the debugger to search outside the current procedure, qualify the block name with "%EXTERN". (This is seldom necessary if ambiguous naming is avoided.)

For example:

```
procedure A;              (* 1 *)
   procedure B;           (* 2 *)
      procedure C;        (* 3 *)
         procedure D;     (* 4 *)
         end;
      end;
   end;
   procedure C;           (* 5 *)
      procedure B;        (* 6 *)
         procedure C;     (* 7 *)
         end;
      end;
   end;
end;
```

```
procedure B;                 (* 8 *)
   procedure C;              (* 9 *)
      procedure D;           (* 10 *)
      end;
   end;
   procedure B;              (* 11 *)
      procedure A;           (* 12 *)
         procedure A;        (* 13 *)
            procedure B;     (* 14 *)
            end;
         end;
      end;
   end;
end;
```

The following table describes how each block can be referenced given the current evaluation environment is in the external procedure "%EXTERN.A" (block 1).

BLOCK	REFERENCE
1	"A" or "%EXTERN.A"
2	"A.B", "B", or "%EXTERN.A.B"
3	"B.C", "A.B.C", or "%EXTERN.A.B.C".
4	"D", "C.D", "B.C.D", "A.B.C.D", "%EXTERN.A.B.C.D", "B.D", "A.B.D", "%EXTERN.A.B.D", "A.D", "%EXTERN.A.D"
5	"C", "A.C", "%EXTERN.A.C"
6	"C.B", "A.C.B", "%EXTERN.A.C.B"

BLOCK	REFERENCE
7	"C.C", "C.B.C", "A.C.C", "%EXTERN.A.C.C", "A.C.B.C", "%EXTERN.A.C.B.C"
8	"%EXTERN.B"
9	"%EXTERN.B.C"
10	"%EXTERN.B.D", "%EXTERN.B.C.D"
11	"B.B" or "%EXTERN.B.B"
12	"B.B.A" or "%EXTERN.B.B.A"
13	"B.A.A", "B.B.A.A", "%EXTERN.B.A.A", "%EXTERN.B.B.A.A"
14	"B.B.B" "B.A.B", "B.A.A.B", "B.B.A.B", "B.B.A.A.B", "%EXTERN.B.B.B", "%EXTERN.B.A.B", "%EXTERN.B.A.A.B", "%EXTERN.B.B.A.B", or "%EXTERN.B.B.A.A.B"

If the current evaluation environment is in the internal procedure block "%EXTERN.A.B.C.D" (block 4), then the following is true in the debugger:

— "B"	refers to block 2
— "B.C"	refers to block 3
— "C"	refers to block 3
— "%EXTERN.B.A"	is an ambiguous reference (block 12 or 13)
— "%EXTERN.B.B.A"	refers to block 12
— "%EXTERN.B.C"	refers to block 9

If the current evaluation environment is in the external procedure block "%EXTERN.B" (block 8), then the following is true in the debugger:

— "A.B.C.D"	refers to block 1
— "A"	is an ambiguous reference (block 12 or 13)

— "B" refers to block 11

— "B.A" refers to block 12

— "A.B" refers to block 14

— "B.A.B" refers to block 14

Built-in Function Support

CodeWatch does not support any of the LPI-PASCAL built-in functions.

Referencing Arrays and Aggregate Structures

Arrays and aggregates structures can be referenced in their entirety by referring to them by their array or record name, or individual subfields or members can be referenced. A subfield reference must be fully qualified. Array slices can be referenced as described in the "EVALUATE" section of Chapter 4.

Enumerated types can be referenced directly. Sets cannot be manipulated.

Sample CodeWatch Session Using LPI-PASCAL

This debugging session illustrates how to use the commands and features of CodeWatch when debugging LPI-PASCAL programs. Following the session is the source listing of the sample program.

The sample program, primes.pas, calculates the number of prime numbers within a given range. The program has been compiled using the -deb option to produce the necessary information for the debugger and the -l option to produce a listing file. The program has also been linked. For example,

```
lpipascal primes.pas -deb -l
lpild primes.o -o primes
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild primes.obj -o primes
```

In this sample session, explanatory comments (which are not part of the session) are the bulleted items. The system prompt is $. The abbreviated form of the commands is used after the command has been previously spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
codewatch primes
CodeWatch setting up "primes". Wait ...

*********************************************************
* CodeWatch,  Revision 4.2.0                            *
* ------------------------                              *
*    Copyright(c) Language Processors, Inc. 1987        *
*********************************************************
Evaluation environment is MAIN:(inactive)
```

- Find the string "main program".

```
DEB> FIND main program
  128: begin   (* main program *)
```

- Print 11 lines.

```
DEB> PRINT 11
  128: begin   (* main program *)
  129:
  130:    writeln (' *** Sieve of Erathosthenes ***');
  131:    writeln;
  132:
  133:    read_input(n);
  134:
  135:   while (n > 1) do begin
  136:      sift (n);
  137:      read_input(n);
  138:   end;
```

- Set a conditional breakpoint when the variable n is greater than 1, at line 135 of the main program.

```
DEB> BREAKPOINT 135 /IF= n > 1
```

- Enable entry tracing and begin program execution.

```
DEB> TRACE ENTRY
DEB> CONTINUE
**** MAIN\%ENTRY
 *** Sieve of Erathosthenes ***

**** MAIN.READ_INPUT\%ENTRY
 Input upper boundary:
 10
**** MAIN.READ_INPUT\%EXIT
Break at MAIN\135
```

- Set a breakpoint at line 120 of the sift routine and continue program execution.

```
DEB> B sift\120
DEB> C
**** MAIN.SIFT\%ENTRY
Break at MAIN.SIFT\120
```

- Evaluate data items in the sift routine.

```
DEB> EVALUATE count
      COUNT =        5 INTEGER
DEB> E flags[count]
      TRUE BOOLEAN
```

- Display type information about various data items.

```
DEB> TYPE primes
PRIMES(1:1000) RESULTS
DEB> TY results
 ARRAY OF INTEGER
```

- Display the entire structure or record.

```
DEB> TY /FULL primes
  PRIMES(1:1000) automatic
   INTEGER INTEGER
```

- Display stack frame information including argument information.

```
DEB> STACK /ARGUMENTS /ALL

Stack contains 4 frames.
Current execution point is MAIN.SIFT\120

   4: Owner is "MAIN.SIFT"
      Arguments:
         N =          10 INTEGER
      Called from MAIN\136

   3: Owner is "MAIN"
      Arguments: None
      Main program
```

- Display the arguments to sift.

```
DEB> ARGUMENTS
N =      10 INTEGER
```

- Set the default stepping mode to step in and the stepping action list to print the current source line.

```
DEB> DSTEP IN [P]
```

- Step one statement.

```
DEB> STEP
Step at MAIN.PRINT_OUT\%ENTRY
  67:   procedure print_out(values: results; total: integer);
```

- Step out of the print_out routine.

```
DEB> STEP OUT
Number of primes found was    (prime)   5

     1     2     3     5     7

Step at MAIN.PRINT_OUT\%EXIT
  86:   end;  (* print_out *)
```

- Set a breakpoint at the entry point of the read_input procedure with an action list evaluating the value of maxv. Continue program execution.

```
DEB> B read_input\%ENTRY [E maxv]
DEB> C
**** MAIN.SIFT\%EXIT
Break at read_input\%ENTRY
MAXV =       10 INTEGER
```

- Modify the value of maxv.

```
DEB> LET maxv = maxv/2
```

- Change the current evaluation environment to primes and evaluate the new value of N. Continue program execution.

```
DEB> ENVIRONMENT main
DEB> E n
N =          5 INTEGER
```

- Reset the current evaluation environment to the previous evaluation environment by using the ENVIRONMENT command without an argument.

```
DEB> ENV
Environment reset to MAIN.READ_INPUT
```

- Go to line 26 and continue program execution.

```
DEB> GOTO 26
Execution point is now MAIN.READ_INPUT\26
DEB>  C
 Input upper prime boundary: 10
**** MAIN.READ_INPUT\%EXIT
Break at main\135
```

- Remove the breakpoint at line 120, set a new breakpoint at the entry point of the isprime procedure, and continue execution.

```
DEB> NB 120
```

```
DEB> B isprime\%entry
DEB> C
**** MAIN.SIFT\%ENTRY
**** MAIN.PRINT_OUT\%ENTRY
Break at MAIN.ISPRIME\%ENTRY
```

- Set the return value of isprime to 15.

```
DEB> RETURN 15
```

- List the return value of isprime.

```
DEB> LRETURN
Return value for MAIN.ISPRIME is          15 INTEGER
```

- Define a macro that removes all breakpoints, removes the default stepping action list, disables tracing, moves the current source file pointer to line 1 of the source program, and prints 10 lines.

```
DEB> MACRO fresh = [NB /ALL; DS []; NTR E; PO 1; P 10]
```

- Use the macro.

```
DEB> fresh

  1: (* Sieve of Erathoshenes *)
  1: (* Sieve of Erathoshenes *)
  2:
  3: program main;
  4:
  5:   const
  6:     max_value = 1000;
  7:     max_primes = 1000;
  8:
  9:   type
 10      sieve = ARRAY [0..max_value] of boolean;
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: primes.pas
Compiled: 12-May-87 14:46:43 by LPI-Pascal, Rev 02.04.03
Options: deb opt 2 l


```
1   (* Sieve of Eratosthenes *)
2
3   program main;
4
5       const
6           max_value       = 1000;
7           max_primes      = 1000;
8
9       type
10          sieve           = ARRAY [0..max_value] of boolean;
11          results         = ARRAY [1..max_primes] of integer;
12
13      var
14          n               : integer;
15
16*     (* ------------------------------------------------------------
17*      * PROCEDURE read_input
18      *)
19
20      procedure read_input (var maxv: integer);
21
22          var ok : boolean;
23
24      begin
25
26          repeat
27              writeln ('Input upper prime boundry');
28              readln (maxv);
29              if (maxv > max_value) then
30                  writeln ('Value too big.  Try again.')
31              else
32                  ok := true
33          until (ok);
34
35      end;  (* read_input *)
36
35
```

```
37*     (* ------------------------------------------------------------
38*      * FUNCTION isprime
39      *)
40
41      function isprime (number: integer; values: results; total: integer):integer;
42
43          var n : integer;
44
45          label 999;
46
47      begin
48
49          for n := 1 to total do begin
50              if number = values[n] then begin
51                  isprime := number;
52                  goto 999
53              end;
54          end;
55
56          isprime := -1;
57
58      999:
59
60      end;  (* isprime *)
61
62
63*     (* ------------------------------------------------------------
64*      * PROCEDURE print_out
65      *)
66
67      procedure print_out (values: results; total: integer);
68
69          var i : integer;
70
71      begin
72          write ('Number of primes found was ');
73          if isprime (total,values,total) >= 0 then
74              write (' (prime)');
75          writeln (total);
76          writeln;
77
78          for i := 1 to total do begin
79              write (values[i]);
80                  if (i mod 10) = 0 then
81                      writeln;
82          end;
83          writeln;
84          writeln;
85
```

```
86      end;  (* print_out *)
87
88*     (* ------------------------------------------------------------
89*      * PROCEDURE sift
90      *)
91
92      procedure sift (n: integer);
93
94         var
95            i, k, count : integer;
96            flags       : sieve;
97            primes      : results;
98            this_prime  : integer;
99
100     begin
101
102        for i := 0 to n do
103           flags[i] := true;
104
105        count := 1;
106        primes[1] := 1;
107
108        for i := 2 to n do begin
109           if flags[i] then begin
110              this_prime := i;
111              count := count + 1;
112              primes[count] := this_prime;
113              k := i + this_prime;
114              while (k <= n) do begin
115                 flags[k] := false;
116                 k := k + this_prime;
117              end;
118           end;
119        end;
120        print_out (primes, count);
121
122     end;  (* sift *)
123
124*  (* ------------------------------------------------------------
125*   * PROGRAM main
126    *)
127
128  begin  (* main program *)
129
130     writeln ('*** Sieve of Erathosthenes ***');
131     writeln;
132
133     read_input (n);
134
```

```
135       while (n > 1) do begin
136          sift (n);
137          read_input (n);
138       end;
139
140    end.  (* main program *)
```

Chapter 10: Debugging LPI-PL/I Programs

Chapter 10: Debugging LPI-PL/I Programs

Specific Ways to Use CodeWatch Features

This section describes CodeWatch features that relate to debugging PL/I programs. The second section contains a sample debugging session of a LPI-PL/I program. The third section contains the listing of the program used in the sample debugging session.

Program Blocks

Program blocks are units of code that provide scope and context for the debugger. An LPI-PL/I program block is a procedure block or a begin block.

Block Names

A procedure block is referred to by the name of that procedure. A begin block is referred to by its label (if present) or by the string "%BEGIN" followed immediately by the line number on which the the block begins. For example, an unlabeled begin block which starts on line 119 of the source file is referred to as "%BEGIN119".

Referencing Nested Blocks

LPI-PL/I program blocks may be nested. Rules for naming nested blocks are as follows:

1. If a block defined within the compilation unit is contained within or contains the debugger's current evaluation environment, the block may be referred to simply by its name. The name may need to be qualified to make it unique within the external procedure.

If the block contains the debugger's current evaluation environment, the search for the referenced block begins from the current environment and continues through each successive containing (parent) block until the referenced block is found.

2. If a block is in some other external procedure, it must be qualified with at least the external procedure name, and it may require further qualification. Furthermore, a fully qualified name to a block in some other external procedure may be the same as a partially qualified block name in the current procedure. To force the debugger to search outside the current procedure, qualify the block name with "%EXTERN". (This is seldom necessary if ambiguous naming is avoided.)

For example:

```
A: proc;              /* 1 */
   B: proc;           /* 2 */
      C: proc;        /* 3 */
         D: proc;     /* 4 */
         end D;
      end C;
   end B;
   C: proc:           /* 5 */
      B: proc;        /* 6 */
         C: proc;     /* 7 */
         end C;
      end B;
   end C;
end A;
```

```
B: proc;                 /* 8 */
   C: proc;              /* 9 */
      D: proc;           /* 10 */
      end D;
 end C;
  B: proc;               /* 11 */
     A: proc;            /* 12 */
        A: proc;         /* 13 */
           B: proc;      /* 14 */
           end B;
        end A;
     end A;
  end B;
end B;
```

The following table describes how each block can be referenced given the current evaluation environment is in the external procedure "%EXTERN.A" (block 1).

BLOCK	REFERENCE
1	"A" or "%EXTERN.A"
2	"A.B", "B", or "%EXTERN.A.B"
3	"B.C", "A.B.C", or "%EXTERN.A.B.C".

BLOCK	REFERENCE
4	"D", "C.D", "B.C.D", "A.B.C.D", "%EXTERN.A.B.C.D", "B.D", "A.B.D", "%EXTERN.A.B.D", "A.D", "%EXTERN.A.D"
5	"C", "A.C", "%EXTERN.A.C"
6	"C.B", "A.C.B", "%EXTERN.A.C.B"
7	"C.C", "C.B.C", "A.C.C", "%EXTERN.A.C.C", "A.C.B.C", "%EXTERN.A.C.B.C"
8	"%EXTERN.B"
9	"%EXTERN.B.C"
10	"%EXTERN.B.D", "%EXTERN.B.C.D"
11	"B.B" or "%EXTERN.B.B"
12	"B.B.A" or "%EXTERN.B.B.A"
13	"B.A.A" "B.B.A.A", "%EXTERN.B.A.A", or "%EXTERN.B.B.A.A"
14	"B.B.B" "B.A.B" "B.A.A.B", "B.B.A.B", "%EXTERN.B.B.B", "%EXTERN.B.A.B", "%EXTERN.B.A.A.B", "%EXTERN.B.B.A.B", or "%EXTERN.B.B.A.A.B"

If the current evaluation environment is in the internal procedure block "%EXTERN.A.B.C.D" (block 4), then the following is true in the debugger:

— "B"	refers to block 2
— "B.C"	refers to block 3
— "C"	refers to block 3
— "%EXTERN.B.A"	is an ambiguous reference (block 12 or 13)
— "%EXTERN.B.B.A"	refers to block 12

— "%EXTERN.B.C" refers to block 9

If the current evaluation environment is in the external procedure block "%EXTERN.B" (block 8), then the following is true in the debugger:

— "A.B.C.D"	refers to block 1
— "A"	is an ambiguous reference (block 12 or 13)
— "B"	refers to block 11
— "B.A"	refers to block 12
— "A.B"	refers to block 14
— "B.A.B"	refers to block 14

Built-In Function Support

The following PL/I built-in functions are supported by CodeWatch:

ADDR	DECIMAL	INDEX	SIZE
BINARY	DIMENSION	LBOUND	SUBSTR
BIT	FIXED	LENGTH	TRIM
BYTE	FLOAT	NULL	UNSPEC
CHAR	HBOUND	RANK	VERIFY

Referencing Arrays and Aggregate Structures

Arrays and aggregate structures can be referenced in their entirety by referring to them by their array or structure name, or individual subfields or members can be referenced using the conventional PL/I syntax as in "REC_NAME.FIELD". A subfield can be referenced directly as long as its name is unique.

Array slices can be referenced as described in the "EVALUATE" section in Chapter 4.

Sample CodeWatch Session Using LPI-PL/I

This debugging session illustrates how to use the commands and features of CodeWatch when debugging PL/I programs. Following the session is the source listing of the sample program.

The sample program, primes.pl1, calculates the number of prime numbers within a given range. The program has been compiled using the -deb option to produce the necessary information for the debugger, and the -l option to produce a listing file. The program has also been linked. For example,

```
lpipl1 primes.pl1 -deb -l
lpild primes.o -o primes
```

Object files are given the suffix .obj on systems running under MS-DOS. For example, the following link line is applicable to MS-DOS systems.

```
lpild primes.obj -o primes
```

In this sample session, explanatory comments (which are not part of the session) are the bulleted items. The system prompt is $. For clarity, the abbreviated form of the commands is used only after the command has been previously used spelled out in its entirety.

- Invoke CodeWatch at the system prompt.

```
$ codewatch primes
CodeWatch setting up "primes". Wait ...

************************************************************
* CodeWatch,  Revision 4.2.0                               *
* -------------------------                                *
*    Copyright(c) Language Processors, Inc. 1987           *
************************************************************
Evaluation environment is PRIMES:(inactive)
```

- Find the string "main".

```
DEB> FIND main
  108:      /* main procedure */
```

- Print 15 lines.

```
DEB> PRINT 15
  108:      /* main procedure */
  109:
  110:      declare n fixed binary(31);
  111:
  112:      put skip;
  113:      put list (' *** Sieve of Eratosthenes ***');
  114:      put skip (2);
  115:
  116:      call read_input(n);
  117:
  118:      do while (n > 1);
  119:         call sift(n);
  120:         call read_input(n);
  121:      end;
  122:
```

- Set a breakpoint at line 118 of the main program.

```
DEB> BREAKPOINT 118
```

- Enable entry tracing and begin program execution.

```
DEB> TRACE ENTRY
DEB> CONTINUE
**** PRIME\%ENTRY

 *** Sieve of Eratosthenes ***

**** PRIMES.READ_INPUT\%ENTRY
 Input maximum prime boundary: 10
**** PRIMES.READ_INPUT\%EXIT
Break at PRIMES\118
```

- Set a breakpoint at line 94 of the sift routine to break when the variable count equals itself and continue program execution.

```
DEB> B SIFT\94 /IF= COUNT = COUNT
DEB> C
**** PRIMES.SIFT\%ENTRY
Break at PRIMES.SIFT\94
```

- Evaluate the value of data items in the sift routine.

```
DEB> EVALUATE THIS_PRIME
THIS_PRIME =          2 {fixed binary (31)}
DEB> E PRIMES(COUNT)
PRIMES(1) =           1 {fixed binary (31)}
```

- Display stack frame information including argument information.

```
DEB> STACK /ARGUMENT /ALL

Stack contains 4 frames.
Current execution point is PRIMES.SIFT\94

   4: Owner is "PRIMES.SIFT"
     Arguments:
        N =             10   {fixed binary (31)}
     Called from PRIMES\119

   3: Owner is "PRIMES"
     Arguments: None
     Main program
```

- Display declaration information about various data items.

```
DEB> TYPE COUNT
COUNT  fixed binary (31) automatic
DEB> TY FLAGS
FLAGS(1:1000)  bit (1) automatic
```

- Set the default stepping mode to step in and the stepping action list to print the current source line.

```
DEB> DSTEP IN [P]
```

- Step one statement.

```
DEB> STEP
Step at PRIMES.SIFT\95
  95:        primes(count) = this_prime;
```

- Step two statements.

```
DEB>  S 2
Step at PRIMES.SIFT\97
  97:        do while (k < n);
```

- Remove the breakpoint at line 94 and step out of the sift routine.

```
DEB> NB 94
DEB> STEP OUT
Number of primes found was      (prime)   5
    1     2     3     5     7

Step at PRIMES.SIFT\%EXIT
  106: end sift;
```

- Set a breakpoint at the entry point of the read_input procedure with an action list evaluating the value of MAXV. Continue program execution.

```
DEB> B read_input\%ENTRY [E MAXV]
DEB> C
Break at PRIMES.READ_INPUT\%ENTRY
MAXV =            10   {fixed binary (31)}
```

- Modify the value of MAXV.

```
DEB> LET MAXV = MAXV/2
```

- Change the current evaluation environment to primes and evaluate the new value of N. Continue program execution.

```
DEB> ENVIRONMENT PRIMES
DEB> E N
N =            5   {fixed binary}
```

- Reset the current evaluation environment to the previous evaluation environment by using the ENVIRONMENT command without an argument.

```
DEB> ENV
Environment reset to PRIMES.READ_INPUT
```

- Go to line 21 and continue program execution.

```
DEB> GOTO 21
Execution point is now PRIMES.READ_INPUT\21
DEB>  C
Input maximum prime boundary: 10
**** READ_INPUT\%EXIT
Break at PRIMES\118
```

- Set a breakpoint at the entry point of the isprime procedure and continue execution.

```
DEB> B isprime\%entry
DEB> C
**** PRIMES.SIFT\%ENTRY ****
**** PRIMES.PRINT_OUT\%ENTRY ****
 Number of primes found was
Break at PRIMES.ISPRIME\%ENTRY
```

- Set the return value of isprime to 15.

```
DEB> RETURN 15
```

- List the return value of isprime.

```
DEB> LRETURN
Return value for PRIMES.ISPRIME is    15 {fixed binary (31)}
```

- Define a macro that lists all breakpoints, the current evaluation environment, all macros, and the current execution point.

```
DEB> MACRO info = [LB /A; LENV; LMA /A; WHERE]
```

- Use the macro.

```
DEB> info
Break set at PRIME\118 (count = 2)
Break set at PRIMES.READ_INPUT\%ENTRY
    (count = 1) [E MAXV;]
Break set at PRIME.ISPRIME\%ENTRY (COUNT = 1)
Evaluation environment is PRIMES.ISPRIME
  info = [LB /ALL; LENV; LMA /ALL; WHERE]
Current execution point is PRIMES.ISPRIME\EXIT
```

- Quit the debugging session.

```
DEB> QUIT
CodeWatch Quit ... Bye!
```

Program Listings

Source File: primes.pl1
Compiled: 1-Jan-85 12:00:01 by LPI-PL/I, Rev 01.00.00
Options: deb opt 2 l


```
1     /* Sieve of Eratosthenes */
2
3     primes: procedure;
4
5     %replace FALSE          by '0'B;
6     %replace TRUE           by '1'B;
7
8     %replace MAX_VALUE      by 1000;
9     %replace MAX_PRIMES     by  500;
10
11
12     read_input: procedure (maxv);
13
14           declare maxv fixed binary(31);
15           declare instring char(128) varying;
16
17           declare ok bit(1);
18
19           ok = FALSE;
20
21           do while (^ok);
22                 put list (' Input maximum prime boundary:');
23                 get list (instring);
24                 maxv = decimal(instring);
25                 if maxv > MAX_VALUE then do;
26                       put list (' Value too big.  Try again.');
27                       put skip;
28                 end;
29                 else do;
30                       ok = TRUE;
31                 end;
32           end;
33
34     end read_input;
35
```

```
36      isprime: procedure (number,values,total) returns (fixed binary (31));
37
38              declare number              fixed binary (31),
39                      values (1:MAX_PRIMES) fixed binary (31),
40                      total               fixed binary (31);
41              declare n                   fixed binary (31);
42
43              do n = 1 to total;
44                      if number = values(n) then
45                              return (number);
46              end;
47
48              return (-1);
49
50      end isprime;
51
52      print_out: procedure (values,total);
53
54              declare values(1:MAX_PRIMES) fixed binary(31),
55                      total fixed binary(31);
56
57              declare i fixed binary(15);
58
59              put list (' Number of primes found was');
60              if isprime (total,values,total) >= 0 then
61                      put list (' (prime)');
62              put edit (total) (F(4));
63              put skip (2);
64
65              do i = 1 to total;
66                      put edit (values(i)) (F(7));
67                      if mod(i,10) = 0 then do;
68                              put skip;
69                      end;
70              end;
71
72              put skip (2);
73
74      end print_out;
75
76      sift: procedure (n);
77
78              declare n fixed binary(31);
79
```

```
80          declare (i, k, count, this_prime) fixed binary(31),
81                flags(1:MAX_VALUE) bit(1),
82                primes(1:MAX_PRIMES) fixed binary(31);
83
84          do i = 1 to n;
85                flags(i) = TRUE;
86          end;
87
88          count = 1;
89          primes(1) = 1;
90
91          do i = 1 to n;
92                if flags(i) = TRUE then do;
93                      this_prime = i + 1;
94                      count = count + 1;
95                      primes(count) = this_prime;
96                      k = i + this_prime;
97                      do while (k < n);
98                            /* cancel all multiples */
99                            flags(k) = FALSE;
100                            k = k + this_prime;
101                     end;
102               end;
103         end;
104         call print_out(primes,count-1);
105
106   end sift;
107
108         /* main procedure */
109
110         declare n fixed binary(31);
111
112         put skip;
113         put list (' *** Sieve of Eratosthenes ***');
114         put skip (2);
115
116         call read_input(n);
117
118         do while (n > 1);
119               call sift(n);
120               call read_input(n);
121         end;
122
123   end;
```

Glossary

absolute activation number — A positive integer that specifies the exact activation of a procedure.

action list — A set of one or more CodeWatch commands that are separated by semicolons and enclosed within square brackets (for example, [PRINT 10; LENVIRONMENT]).

activation number — An integer that specifies a particular activation of a procedure when one or more such activations exist.

active environment — An environment that exists on the current stack.

breakpoint — A point set by the BREAKPOINT command at which program execution pauses to allow the debugger to process an action list or accept additional commands.

command interpreter — A system program that acts an an intermediary between a user (or a process) and the operating system. It reads, interprets, and (in some cases) executes commands.

default evaluation environment — The environment that contains the current point of execution.

display mode — The mode in which the value of an expression is printed, either ASCII, BIT, FLOAT, HEX, INTEGER, OR OCTAL.

entry point	The location during execution where a procedure has been called, but before variable size local storage has been allocated and before other prelude code has been run.
entry tracing	The printing of messages identifying the entry point of each procedure as it is called during program execution.
environment	A program block that establishes a frame of reference for identifying specific instances of variables and statements.
evaluation environment	The environment that currently supplies scope and context to the debugger for referencing variables and statements.
evaluation language	The source language in which the current evaluation environment's program was written.
execution pointer	A debugger pointer that tracks the point of program execution.
exit point	The location in a procedure after which a return value, if any, has been computed, but before the return statement has been executed.
inactive environment	An environment that does not exist on the current stack.
line number offset	A plus sign followed by a positive integer after a source line number or a statement label. It indicates the number of physical source lines to count from the source line number or the label to locate another statement when its label or source line number is unknown. The current statement has a line number offset of zero. For example, line 15 of a program may be referred to by "11+4".

macro	A name defined by the MACRO command as a shorthand for an action list.
program block	A unit of code that supplies scope and context for the debugger. Program blocks are defined according to the source language.
program block name	The name used to refer a program block.
relative activation number	Either zero or a negative integer. Zero specifies the current activation of a procedure. A negative integer specifies an activation of a procedure by counting the number of activations prior to the current activation.
shell	The UNIX command interpreter, a system program that reads, interprets, and (in some cases) executes commands.
skip count	The number of times a breakpoint or a watchpoint is to be skipped.
source file pointer	Pointer to the current line in the current source file.
source line number	The physical line number in the source file on which a statement begins.
statement identifier	A source line number, a source line number and a statement offset, or a statement label.
statement label	A source-language-defined mark for identifying statements.
statement offset	A period followed by a positive integer after a line number offset indicating the number of statements to count from the first statement on a line. The first statement has a statement offset of zero. A plus sign must precede the period if the line number offset equals zero.

statement tracing	The printing of messages identifying each statement prior to its execution.
variable name	A source-language-defined identifier for a variable. In the debugger, an environment name followed by a backslash (\) can precede the variable name to identify a variable in another program block.
watchpoint	A point set by the WATCH command that is used to monitor any changes to data When a watched variable changes, program execution stops and control is returned to the debugger.

Index

M

N

S

INTERACTIVE

A Kodak Company

DOC0114-2Z