CRISP ®/32
Logic Debugger
User's Manual

CRISP *Software Products*

SQUARE D
GROUPE SCHNEIDER

CRISP®/32
Logic Debugger User's Manual
Document number: 500 071 - 001, Rev.1

Document History

Revision	Date	Pages affected/Description of change
1	12/17/93	Initial Release. ECN # 4318

Software Version CRISP/32 Rev. 3.0 and Later

Table of Contents

Introduction

General

The CRISP Logic Debugger is a tool that helps you locate logic errors in your CRISP Application Logic program. You can use the debugger with a logic that compiles and links successfully, but does not run correctly.

You locate these errors with the debugger by observing and manipulating your logic interactively as it executes. By entering debugger commands at the terminal, you can do the following.

- Control the logic execution. Start your logic, stop at points of interest, step through lines of code, resume execution, and stop your logic.
- Examine and modify variables in the CRISP Database.
- Affect the behavior of the debugger itself, and how you use it.

Notes:

Functional Features

The CRISP Logic Debugger interfaces exclusively with a CRISP/32 Application Logic Program, and provides the following features.

- Symbolic Debugging. The CRISP Logic Debugger is a symbolic debugger. You can refer to any symbol that appears in the declarations section of your logic using its declared name. Array elements can be referenced using one level of subscripting. The subscript must be specified as a numeric integer constant; named Constants are not supported.

- Support for all data types. The debugger understands all data types generated by the CRISP/32 Compiler. Numerics and Longs are displayed as integers. Floats are displayed as floating-point numbers. Timers and Counters are displayed as multi-element structures. Strings are displayed as ASCII text.

- Flexible Data Format. The SET RADIX command allows you to change the default display method to unsigned binary, unsigned octal, signed decimal, unsigned decimal, or unsigned hexadecimal. This allows you to examine and modify any CRISP variable in any of these formats.

- Breakpoints. The SET BREAK command allows you to set breakpoints on any line in your logic that contains an executable CRISP statement. A breakpoint suspends execution of the logic and allows you to interact with the debugger at that point.

- Starting or Resuming logic execution. The GO and STEP commands allow you to control the execution of your logic. GO causes the logic to execute until it reaches the next breakpoint. STEP causes the current statement to be executed and automatically breaks before executing the next statement.

- Manipulation of CRISP Database variables. The EXAMINE command allows you to examine the current value of a CRISP variable. The DEPOSIT command allows you to modify the value of a CRISP variable.

Convenience Features

The following features have been added to the CRISP Logic Debugger that make it easier to use.

- Keypad Mode. Some of the most commonly used commands are assigned by default to keys on the numeric keypad on a DEC VT terminal. This keypad mode can be disabled if you would rather use the numeric keypad for the entry of numerical data.

Convenience Features (cont)

- Command Line editing. The debugger makes full use of the VMS command line recall and command line editing features. The previous 20 command lines are retained and can be recalled by pressing the up and down arrows on a VT terminal. The command line can be edited by using the left and right arrows to position the cursor in the line. The backspace key (F12) positions the cursor at the beginning of the line. The Ctrl/E key positions the cursor at the end of the line. The "delete word" key (F13 or linefeed) deletes the word immediately to the left of the cursor. Ctrl/A and F14 toggle between insert and overstrike mode.

  On hardcopy terminals, commands may be recalled with Ctrl/B. Editing is limited to deleting characters from the end of the command and entering changes.

Conventions

The following list identifies the conventions used in this manual.

Ctrl/? The symbol Ctrl/?, where ? represents a terminal control character, is generated by holding down the Ctrl key while pressing the key of the specified terminal character.

option, ... A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma preceding the ellipsis indicates that successive items must be separated by commas.

[option] Square brackets in command format definitions indicate that a syntactic element is optional. Square brackets are not optional, however, when used to delimit a directory name in a VMS file specification.

{a | b} Braces surrounding two or more items separated by a vertical bar indicate a choice; you must choose one of the syntactic elements presented. Choose wisely.

General

Using the CRISP Logic Debugger is simple. You will compile and run your logic in the usual way with the additional /DEBUG qualifier.

Building your Logic

In a CRISP logic, the first 16 logicals are reserved for CRISP system use. The debugger makes use of two new logicals, DEBUG and DEBUG_BREAK. You should add these new logicals immediately following the PAUSE logical. The DEBUG logical is set TRUE on each pass of logic if that logic was compiled with /DEBUG. It is set FALSE on each pass of logic if that logic was NOT compiled /DEBUG. The DEBUG_BREAK logical is set TRUE by the debugger whenever it is prompting you for input. It is set FALSE when you instruct the debugger to allow the logic to run.

A CRISP logic is compiled and linked using the LGBUILD command. This command allows you to optionally compile, build a database, and build a logic executable for your CRISP/32 program. To use the debugger, you will compile your logic with the /DEBUG qualifier. In the following example, a logic TEST is built for debugging.

```
$ LGBUILD

        +----------------------------------------+
        | CRISP/32 user logic build procedure    |
        +----------------------------------------+

Enter logic filespec . . . . . []: TEST
  Do you want to compile TEST . . . . . [Yes]?
  Do you want to build the TEST database  . . [Yes]?
  Do you want to assemble the TEST logic  . . [Yes]?
  Do you want to build the TEST logic . . . . [Yes]?
  Enter CRISP compiler qualifiers: /DEBUG

%LGBUILD-I-START, CRISP logic build started at
                  31-MAY-1991 14:27:09.17

$ Crisp /list /DEBUG DISK$USER:[CRISP.LOG]TEST.C32;
$ Macro  TEST
 This CRISP logic file was compiled /DEBUG
$ Link /share=TEST.DBE TEST.DBO
$ Link /map /full /notraceback TEST

%LGBUILD-I-DONE, CRISP logic build is finished at
                 31-MAY-1991 14:28:44.97
```

Note that the qualifier /DEBUG was entered for the CRISP compiler. A message was printed by the VMS MACRO Assembler indicating that the CRISP logic was compiled with debug.

Building your Logic (cont)

The LGBUILD command can also be invoked all on one line as follows.

```
$ LGBUILD TEST Y Y Y Y /DEBUG

        +----------------------------------------+
        | CRISP/32 user logic build procedure    |
        +----------------------------------------+


%LGBUILD-I-START, CRISP logic build started at
                  31-MAY-1991 14:32:44.29

$ Crisp /list /debug DISK$USER:[CRISP.LOG]TEST.C32;
$ Macro  TEST
 This CRISP logic file was compiled /DEBUG
$ Link /share=TEST.DBE TEST.DBO
$ Link /map /full /notraceback TEST

%LGBUILD-I-DONE, CRISP logic build is finished at
                 31-MAY-1991 14:34:16.41
```

In the one-line command, the answers to the questions are entered as parameters to the command file.

For more information on the LGBUILD command, refer to the CRISP/32 Utilities Reference Manual.

Building your Logic (cont)

NOTE

The debugger currently imposed a size limitation on CRISP label names of 29 characters. This is 1 less than the normal CRISP limitation of 30 characters. If you have any lengthy LABEL; names that exceed 29 characters you will get errors during the "Macro" phase of LGBUILD, and the build procedure will terminate. In that case. you must determine which names are too long and shorten them. Here's how to do it.

1. The MACRO assembler will tell you which line numbers the long name occurred on, but not the name itself. This line number is NOT for your CRISP source program, but rather for the resulting MACRO source.

2. Edit PROG.MAR, where "prog" is the name of your CRISP source program. Go to the line numbers identified by the MACRO assembler. (If you are using EVE or TPU, press the Do key, and type `Line x`, where x is the line number.) You will now be on the line referencing the offending name. Write it down, and write down an acceptable shortened version.

3. Now edit your CRISP source program, PROG.C32, search for all occurrences of each name you wrote down and replace them with the shortened name.

You are now ready to execute the LGBUILD command again.

Invoking the Debugger

You start a CRISP logic with debug using the LGINSTALL command, supplying the /DEBUG qualifier. In the following example, the TEST logic built above is started with the debugger.

```
$ LGINSTALL START TEST /DEBUG
```

When you start the logic with debug, control will not be returned to the DCL or LGINSTALL prompt right away. Instead, the CRISP Logic Debugger will use the terminal that you issued the LGINSTALL START command on as the debugger terminal. The debugger display will be initialized, and the initial CSPDBG> prompt will be issued. At this point, any of the debugger commands may be executed.

Invoking the Debugger (cont)

NOTE

DO NOT instruct CRISPmon to monitor a logic that you are debugging. If you do, it will complain that the logic has stopped whenever the debugger is prompting you for input.

Ending a Debugging Session

The debugging session can be terminated at any time by entering the EXIT command (or pressing the F10 or Ctrl/Z keys) in response to the CSPDBG> prompt. This will cause the CRISP logic being debugged to terminate. The CRISP Logic Executive (CLE) will signal an error message indicating that the logic has stopped running.

Interrupting Logic Execution

The application logic can be interrupted at any time by pressing Ctrl/C. This is useful when your logic is running and either you do not have any breakpoints set or none of your breakpoints are being encountered. Pressing Ctrl/C causes the logic to break at the next CRISP logic statement.

Aborting Debugger Commands

Ctrl/C can also be used to abort certain debugger commands. For example, suppose LNG_AR is declared to be an array of 1000 longwords. Entering EXAMINE LNG_AR will cause the debugger to display the current values of every element of the array. You may press Ctrl/C at any time to terminate this command.

General

If you are running on a hardcopy terminal, the debugger simply prompts you for input using the CSPDBG> prompt.

If you are running on a DEC VT terminal, the debugger divides the screen into two areas, a source display area and an output area. Display of the source code is not currently implemented. The output area is where the debugger prompt is displayed. Any commands you type will be displayed there. All debugger output is displayed there.

This is the initial appearance of the debugger screen for the TEST logic.

```
-Source-----------------------------------------------------
Source Code display will be available in a future version of the
CRISP Logic Debugger.

-Output-----------------------------------------------------
CRISP Logic Debugger active for logic TEST
CSPDBG> _
```

Notes:

General

You interact with the CRISP Logic Debugger by typing commands in response to the CSPDBG> prompt. The general format of a debugger command is the following.

Verb [parameter ...] [/qualifier ...] [;]

A command verb is always required. Depending on the verb, there may be any number of additional parameters, qualifiers, and other information. These are discussed on a per-command basis below.

Debugger commands are terminated with a semi-colon (;). This is optional for a single command. However, a semi-colon can be used to separate multiple commands entered on one line.

The debugger does not distinguish between upper and lower case letters for command verbs, parameters, or qualifiers.

Debugger commands can span multiple lines if necessary. To continue a long command, enter a hyphen (-) as the last character on the line. The debugger prompt will change so that it has a leading underscore (_) character, indicating that the command is being continued. This action will continue until a line is entered without a trailing hyphen. That indicates the end of the command. All the lines are concatenated (without the hyphens), and the command is parsed as if it had been typed all on one line.

A debugger command can be up to 1024 characters long.

The command line editing features described earlier are limited to a single line. If a debugger command is recalled, and it is longer than the width of the terminal, it wraps around to the next line and line editing is limited to the last line only.

Some commands are bound to keys on the numeric keypad. These will be identified in the individual command descriptions below.

All command verbs, parameters, and qualifiers can be abbreviated for quicker entry. The rule of thumb is that the abbreviation must define an unambiguous segment of the word, and that this usually requires a minimum of three characters. There are some exceptions to this rule; they are identified as each command is discussed below.

Controlling and Monitoring Logic execution

These commands are used to control the execution of the application logic. They enable you to set and cancel breakpoints, execute CRISP statements one at a time, or execute to the next breakpoint.

Controlling and Monitoring Logic execution (cont)

In general, a CRISP statement can be represented by the following format.

[Keyword; [Cond],] Action

Where:

Keyword;	is an optional keyword, and can be LET;, SET;, CALL;, RECALL;, or MESSAGE;
Cond	is an optional conditional expression. If present, it determines whether the Action statement is executed.
Action	is a mathematical or boolean equation, or a list of call arguments.

(The actual syntax for a CALL statement is slightly different, but that is not important for the purposes of this discussion. For details, refer to the CRISP/32 Language Reference Manual.)

Setting Breakpoints

A breakpoint causes the logic to stop executing. The debugger displays the CSPDBG> prompt, and awaits your command.

When the logic stops at a breakpoint, it always stops after the Conditional expression has been evaluated, but before the Action statement is executed. The debugger prints information about the line at which the logic stopped, depending upon the type of CRISP statement present.

In all cases, the line number and the pass number are displayed. The line number is the physical line number of that statement in the CRISP source code. This line number corresponds to the line numbers which appear on the list file created by the CRISP compiler if the /LIST qualifier is used.

The pass number is the number of times the logic has executed a complete pass through the logic to the END; statement. When the debugger session begins, the pass number starts at zero.

```
CSPDBG> Go
Break at TEST\%line 104;        pass 1
CSPDBG> _
```

If the statement contains a conditional expression, the conditional expression result is also displayed, along with a message indicating whether the Action statement will execute.

```
CSPDBG> Go
Break at TEST\%line 107;        pass 1
Conditional is TRUE - Statement WILL execute
CSPDBG> _
```

Setting Breakpoints (cont)

If the keyword is a SET; or a CALL;, and a conditional expression is present, then the Action is edge-triggered. The debugger will display the conditional expression result, the conditional expression result history, and a message indicating whether the Action statement will execute.

```
CSPDBG> Go
Break at TEST\%line 136;        pass 2
Conditional is FALSE, History is FALSE - Statement will NOT execute
CSPDBG> _
```

Breakpoints are set using the following command.

```
SET BREAK %LINE n
```

Where:

n	is a line number at which a CRISP statement appears.

The SET verb cannot be abbreviated. (S is short for STEP).

%LINE is a special debug operator which specifies that the numeric literal that follows is a line number in your program.

If a CRISP statement spans multiple lines (using the line continuation (\) operator), you set a breakpoint using the line number of the first line.

Breakpoints cannot be set on lines that do not contain executable CRISP statements. Thus, a line containing only a comment is not a valid target for a breakpoint. In these cases, the debugger informs you that the target you specified is not a valid line number, and tells you what the previous and next valid line numbers are.

In the following example, a breakpoint is set on line 126.

```
CSPDBG> SET BRE %LINE 123
No line 123, Previous line is 122, Next line is 126
CSPDBG> SET BRE %LINE 126
Breakpoint set at TEST\%line 126
CSPDBG> _
```

When a breakpoint is successfully set, the debugger responds with a confirmation message. There is no limit to the number of breakpoints that can be set.

Showing current Breakpoints

All existing breakpoints can be displayed using the SHOW BREAK command. This causes the debugger to display all the line numbers where breakpoints have been set. If there are no breakpoints, the debugger prints a message to that effect.

```
CSPDBG> SHO BRE
Breakpoint set at TEST\%line 120
Breakpoint set at TEST\%line 145
There are 2 breakpoints set
CSPDBG> _
```

Canceling Breakpoints

Canceling a breakpoint removes it from the application logic so that once the logic resumes execution, it no longer breaks at that line. Breakpoints can be canceled individually, or en mass.

To cancel a single breakpoint, use the following command.

```
CANCEL BREAK %LINE n
```

Where:

n	is a line number at which a breakpoint had been previously set.

%LINE is a special debug operator which specifies that the numeric literal that follows is a line number in your program.

If you attempt to cancel a breakpoint on a line that does not contain a breakpoint, the debugger informs you that no breakpoint is present on that line. If you attempt to cancel a breakpoint on a line that does not contain an executable CRISP statement, the debugger informs you that the target you specified is not a valid line number, and tells you what the previous and next valid line numbers are.

In the following example a breakpoint at line 126 is canceled.

```
CSPDBG> CAN BRE %LINE 122
No such breakpoint
CSPDBG> CAN BRE %LINE 123
No line 123, Previous line is 122, Next line is 126
CSPDBG> CAN BRE %LINE 126
Breakpoint canceled at TEST\%line 126
CSPDBG> _
```

When a breakpoint is successfully canceled, the debugger prints a confirmation message.

To cancel all current breakpoints, use the following command.

```
CANCEL BREAK /ALL
```

This causes all the breakpoints to be canceled. The debugger displays a confirmation message for each breakpoint canceled.

```
CSPDBG> CAN BRE /all
Breakpoint canceled at TEST\%line 120
Breakpoint canceled at TEST\%line 145
There are no breakpoints set
CSPDBG> _
```

Resuming Logic execution

Resuming Logic execution means that you instruct the debugger to allow the logic to continue running. You do this with one of two commands, GO and STEP, depending on how much logic you want to execute.

The GO command allows the logic to run until it encounters the next breakpoint. It will complete execution of the current statement, and evaluate and execute all subsequent statements up to the line at which the next breakpoint is set. At that time, it will break as described earlier, and the debugger will issue the CSPDBG> prompt. If there are no breakpoints set, the logic will continue running indefinitely. The GO command can be abbreviated as "G". In addition, it is bound to the comma (,) key on the numeric keypad. The debugger does not print any response to the GO command.

The STEP command instructs the debugger to complete execution of the current statement, and break at the next line. This can be thought of as a temporary breakpoint. Using the STEP command, it is possible to single step through your entire logic, executing statements one at a time. Any abbreviation of STEP, down to the single letter S, is accepted. In addition, the STEP command is bound to the zero (0) key on the numeric keypad on a DEC VT terminal.

The STEP command accepts an integer as an optional argument, which instructs the debugger to take that many steps. The default is one. Multiple STEPs can be interrupted by pressing Ctrl/C. The format of the STEP command is the following.

STEP [n]

The optional argument n must be a positive integer.

```
CSPDBG> STEP
Step to TEST\%line 137; pass 2
Conditional is FALSE, History is FALSE - Statement will NOT execute
CSPDBG> _
```

Resuming Logic execution (cont)

If you step to a line at which a breakpoint has been set, the breakpoint takes precedence over the step function. In this case, the message printed by the debugger will reflect the breakpoint rather than the step. In addition, multiple stepping ceases if a breakpoint is encountered.

```
CSPDBG> STEP 100
Step to TEST\%line 137; pass 2
Conditional is FALSE, History is FALSE - Statement will NOT execute
Step to TEST\%line 138; pass 2
Conditional is FALSE, History is FALSE - Statement will NOT execute
Step to TEST\%line 139; pass 2
Conditional is FALSE, History is FALSE - Statement will NOT execute
Break at TEST\%line 140; pass 2
Conditional is TRUE, History is FALSE - Statement WILL execute
CSPDBG> _
```

There are three special cases of the STEP command results.

- Initialization. When the debugger issues its initial CSPDBG> prompt, it is not on any line of your logic yet. If you step from here, the debugger advances to the first CRISP statement after the TABLES; statement. This is usually initialization code before the RESTART; statement.

- JUMP; statements. When the debugger is stopped at a JUMP; statement, there are two possible jump targets, depending upon the conditional expression result. If you step from here, the debugger evaluates the conditional expression result, and steps to the appropriate jump target. This will either be the next physical statement, or the first CRISP statement immediately following the LABEL; statement referenced in the JUMP;.

- End of Logic. When the debugger is stopped on the last statement in your logic, stepping from there causes the debugger to advance to the first statement following the RESTART; statement. All the normal end of logic and start of logic functions occur, i.e. I/O scans, database transfers, synchronous database updates, etc. When the debugger announces the next step, you will notice that the pass count has been incremented.

Interrupting Logic execution

The application logic can be interrupted at any time by pressing Ctrl/C. This is useful when your logic is running and you do not have any breakpoints set, or none of your breakpoints are being encountered. Pressing Ctrl/C causes the logic to break at the next CRISP logic statement.

If your logic is not running at the instant you press Ctrl/C, then your logic will be interrupted the next time CLE schedules it to run. (This is controlled by the /INTERVAL qualifier of the LGINSTALL START command.) The logic will then stop at the first executable CRISP statement following the RESTART; statement. In this case, there is usually a perceptible delay between pressing Ctrl/C and the response from the debugger.

Stopping Logic Execution

The debugging session can be ended at any time by entering the EXIT command (or pressing the F10 or Ctrl/Z keys) in response to the CSPDBG> prompt. This will cause the CRISP logic being debugged to terminate. The CRISP Logic Executive (CLE) will signal an error message indicating that the logic has stopped running. When you exit the debugger, control will be returned to DCL or LGINSTALL, depending on how you invoked LGINSTALL. If you invoked it using a one-line command, control will be returned to DCL.

```
$ LGINSTALL START TEST /DEBUG

CRISP Logic Debugger active for logic TEST
CSPDBG>  EXIT
CRISP Logic TEST is now exiting.
%CLE-E-LGDIED, Logic "!AS" is no longer running on !%D
$ _
```

If you first invoked LGINSTALL without a command line, and started the logic in response to the LGINSTALL> prompt, control will be returned to LGINSTALL.

```
   $ LGINSTALL

LGINSTALL> START TEST /DEBUG

CRISP Logic Debugger active for logic TEST
CSPDBG>  EXIT
CRISP Logic TEST is now exiting.
%CLE-E-LGDIED, Logic "!AS" is no longer running on !%D
LGINSTALL> _
```

Examining and Modifying Database Variables

You can examine and modify any declared variable in the CRISP database against which your logic is running.

Examining CRISP Symbols

The EXAMINE command causes the debugger to display information about any symbol that was declared in the declarations section of your logic. The debugger will display the symbol type, name, and current value of the symbol you specify. The current values are displayed in a format which depends upon the symbol type and the current radix. When the debug session begins, the radix is set to the CRISP default radix. (See the SET RADIX command for further information.)

The EXAMINE command has the following format.

EXAMINE symbol

Examining CRISP Symbols (cont)

Where:

symbol is the name of a declared variable or constant in the declarations section of your logic.

Any abbreviation of EXAMINE, down to the single letter E, is accepted.

If you specify a symbol name that was not declared in your logic, the debugger displays a "no such symbol" error message.

If you specify the name of an array, and you do not specify a subscript, the debugger will display the values of all the elements in the array. This function can be aborted by pressing Ctrl/C.

To specify a single element of an array, specify a subscript as a numeric integer constant. You cannot specify a symbol name, either variable or constant, for a subscript.

Examining Named Constants

CRISP Named Constants are declared using the CONSTANT; keyword. This allows you to declare a symbolic constant that can be used throughout your logic. Constants cannot be assigned new values. When you examine a Constant, the debugger displays the following information.

```
CSPDBG> E PI
Constant; PI = 3.14153
CSPDBG> _
```

A CRISP Constant is treated by the CRISP compiler as a Float, Numeric, or Long, depending upon its declared value. If it contains a decimal point, it is a Float. If its value is between -32768 and 32767, it is a Numeric. If its value is less than -32768 or greater than 32767, it is a Long.

Examining Counters

CRISP Counters are declared using the COUNTER; keyword. The debugger treats Counters as structures consisting of two elements: Reset and Count. In the CRISP default radix, these are displayed as unsigned decimal. When you examine a Counter, the debugger displays the following information.

```
CSPDBG> E C001
Counter; C001 .Reset =      1; .Count =      1
CSPDBG> _
```

Examining Floats

CRISP Floats are declared using the FLOAT; keyword. In the CRISP default radix, these are displayed as single-precision floating-point values in standard floating-point format. When you examine a Float, the debugger displays the following information.

```
CSPDBG> E FLT1
Float; FLT1 = 1.1
CSPDBG> _
```

The debugger will automatically use exponential notation to display very large or very small values.

Examining Logicals

CRISP logicals are declared using the LOGICAL; keyword. These are boolean variables, having only two possible states: True or False. Internally, if the low order bit is true, the logical is considered true. In the CRISP default radix, the debugger displays the value of logicals as "<True>" or "<False>".

When you examine a Logical, the debugger displays the following information.

```
CSPDBG> E DO_EDIT
Logical; DO_EDIT = <True>
CSPDBG> _
```

Examining Longs

CRISP Longs are declared using the LONG; keyword. In the CRISP default radix, these are displayed as signed decimal integer values, ranging from -2147483648 to 2147483647.

When you examine a Long, the debugger displays the following information.

```
CSPDBG> E LNG1
Long; LNG1 =       250549
CSPDBG> _
```

Examining Numerics

CRISP Numerics are declared using the NUMERIC; keyword. In the CRISP default radix, these are displayed as signed decimal integer values, ranging from -32768 to 32767.

When you examine a Numeric, the debugger displays the following information.

```
CSPDBG> E NUM1
Numeric; NUM1 =   2001
CSPDBG> _
```

Examining Strings

CRISP Strings are declared using the STRING; keyword. When you examine a String, the debugger displays its current value using the same rules that apply for supplying initial values for Strings when they are declared by the following.

- The text is bounded by quote (") characters.
- Embedded quote characters are specified as double quotes ("").
- Control characters are specified using the @C construct.

Refer to the CRISP/32 Language reference manual for details.

The display of String values is always ASCII text; it is not affected by the default radix.

If you examine a simple String, having no embedded quotes or control characters, the debugger displays the following information.

```
CSPDBG> E STR1
String; STR1 = "this is simple"
CSPDBG> _
```

If you examine a String which contains embedded quotes, the debugger displays the following information.

```
CSPDBG> E STR1
String; STR1 = "the ""right"" one"
CSPDBG> _
```

Note that if the embedded quotes surround the first or last word in the String, the debugger would display three quotes (""") at the beginning or end of the String, respectively.

If you examine a String which contains control characters, the debugger displays the following information.

```
CSPDBG> E STR1
String; STR1 = "Message goes here@M@J"
CSPDBG> _
```

Examining Timers

CRISP Timers are declared using the TIMER; keyword. The debugger treats Timers as structures consisting of three elements: Reset, Tickdown, and Status. In the CRISP default radix, Reset and Tickdown are displayed as unsigned decimal, and Status is displayed as a named state.

When you examine a Timer, the debugger displays the following information.

```
CSPDBG> E T001
Timer; T001 .Reset =      1; .Tickdown =      1; .Status = reset
CSPDBG> _
```

Examining Timers (cont)

The Status element shows the current state of the Timer. The following are four possible states that a Timer can be in.

- Reset. A Timer is reset when Tickdown equals Reset, and the Timer is not running. That is, the last time the Timer was evaluated in the logic, the second (enable) argument of the Timer expression was false.
- Running. A Timer is running when the last time the Timer was evaluated in the logic, the second (enable) argument of the Timer expression was true.
- Stopped. A Timer is stopped when Tickdown does not equal Reset, and the Timer is not running.
- Expired. A Timer is expired when Tickdown equals zero.

Examining Timers (cont)

NOTE

Timers are decremented by the Database Controller, which is independent of the logic. Thus, if a Timer is running, it will continue running even though the logic may be stopped in the debugger.

Modifying CRISP Variables

The DEPOSIT command allows you to modify the current value of any declared variable in your database. The value you supply must be appropriate for the symbol type and the current radix. When the debug session begins, the radix is set to the CRISP default radix. (See the SET RADIX command for further information.)

The DEPOSIT command has the following format.

DEPOSIT variable = value

Where:

variable is the name of a declared variable in the declarations section of your logic.

value is the desired value.

Any abbreviation of DEPOSIT, down to the single letter D, is accepted.

Any whitespace before or after the equals sign is ignored.

After modifying the value of a variable, the debugger displays the same information for that variable as the EXAMINE command.

If you specify a variable name that was not declared in your logic, the debugger prints a "no such symbol" error message.

If you specify a variable which is an array, you must specify a numeric integer constant subscript as well. You cannot specify a symbol name, either variable or constant, for a subscript.

Modifying Named Constants CRISP Named Constants are declared using the CONSTANT; keyword. Although Named Constants are declared symbols, they are not variables. They are assigned values at compile time, and cannot be changed. If you attempt to modify a Constant, the debugger issues the following error message.

```
CSPDBG> D PI=3
Constant; PI = 3.14153
%CSPDBG-E-CNSTNOMOD, Symbols declared as Constants cannot be modified.
CSPDBG> _
```

Modifying Counters CRISP Counters are declared using the COUNTER; keyword. The debugger treats Counters as structures consisting of two elements: Reset, and Count. In the CRISP default radix, these are displayed as unsigned decimal. You must specify which element you wish to modify using this format.

DEPOSIT Counter.element = value

In the following example, a Counter "reset" value is modified.

```
CSPDBG> E C001
Counter; C001 .Reset =      1; .Count =      1
CSPDBG> D C001.RES = 2
Counter; C001 .Reset =      2; .Count =      1
CSPDBG> _
```

The element names can be abbreviated. If you do not specify an element name, or specify an element that is not part of the Counter structure, the debugger will print an error message.

Modifying Floats CRISP Floats are declared using the FLOAT; keyword, and are single-precision floating-point values in standard floating-point format.

To modify a Float, supply a new value in the form of a floating-point number.

DEPOSIT variable = [{+|-}]int[.frac][E[{+|-}]int]

Where:

E is 10 raised to the power specified by the + or - int which follows it.

For example:

```
CSPDBG> D FLT1 = 6.023E23
Float; FLT1 = 6.023000E+23
CSPDBG> _
```

Modifying Logicals

CRISP logicals are declared using the LOGICAL; keyword. These are boolean variables, having only two possible states: True or False. Internally, if the low order bit is true, the logical is considered true. In the CRISP default radix, the debugger displays the value of logicals as "<True>" or "<False>".

To modify a logical, you supply a value in one of two ways. To set the logical false, you can supply a boolean state <False>, or a numeric value of zero. To set the logical true, you can supply a boolean state <True>, or a numeric value of one.

```
CSPDBG> D DO_EDIT = 0
Logical; DO_EDIT = <False>
CSPDBG> D DO_EDIT = 1
Logical; DO_EDIT = <True>
CSPDBG> D DO_EDIT = <F>
Logical; DO_EDIT = <False>
CSPDBG> D DO_EDIT = <T>
Logical; DO_EDIT = <True>
CSPDBG> _
```

Modifying Longs

CRISP Longs are declared using the LONG; keyword. In the CRISP default radix, these are displayed as decimal integer values, ranging from -2147483648, to 2147483647. To modify a Long, supply a desired value within this range.

```
CSPDBG> D LNG1 = 12345678
Long; LNG1 =     12345678
CSPDBG> _
```

Modifying Numerics

CRISP Numerics are declared using the NUMERIC; keyword. In the CRISP default radix, these are displayed as decimal integer values, ranging from -32768 to 32767. To modify a Long, supply a desired value within this range.

```
CSPDBG> D NUM1 = 2000
Numeric; NUM1 =   2000
CSPDBG> _
```

Modifying Strings

CRISP Strings are declared using the STRING; keyword. To modify a String, specify a desired value using the same rules that apply for supplying initial values for Strings when they are declared.

- The text is bounded by quote (") characters.
- Embedded quote characters are specified as double quotes ("").
- Control characters are specified using the @C construct.

Refer to the CRISP/32 Language reference manual for details.

Modifying Strings (cont)

In the following example, STR1 will be modified to contain trailing carriage-return and line-feed control characters, and the word "test" will be enclosed in quotes.

```
CSPDBG> d str1 = "This is a ""test""@M@J"
String; STR1 = "This is a ""test""@M@J"
CSPDBG> _
```

Modifying Timers

CRISP Timers are declared using the TIMER; keyword. The debugger treats Timers as structures consisting of three elements: Reset, Tickdown, and Status. In the CRISP default radix, Reset and Tickdown are displayed as unsigned decimal, and Status is displayed as a named state.

You must specify which element you wish to modify using this format.

DEPOSIT Timer.element = value

In the following example, a Timer "reset" value is modified.

```
CSPDBG> E T001
Timer; T001 .Reset =      1; .Tickdown =      1; .Status = reset
CSPDBG> D T001.RES = 2
Timer; T001 .Reset =      2; .Tickdown =      1; .Status = stopped
CSPDBG> _
```

Note that as a side effect the Timer status changed from "reset" to "stopped". See the "Examining Timers" section for the definition of the Timer status values.

The element names can be abbreviated. If you do not specify an element name, or specify an element that is not part of the Counter structure, the debugger will print an error message.

The Status element cannot be changed by the debugger.

Overriding the Conditional Expression

Whenever the debugger interrupts the flow of the logic, (such as when a breakpoint is reached, or when you are stepping through the logic), the logic always stops after the conditional expression has been evaluated, but before the Action statement is executed. Information about the current line and conditional expression result is automatically printed at that time.

You can examine this information any time using any one of three commands: The SHOW CONDITIONAL, SHOW HISTORY, and SHOW %LINE commands will cause the debugger to reprint the current line and pass count. If the current line contains a conditional expression, the conditional expression result and the history will also be printed.

Overriding the Conditional Expression (cont)

You can use the SET CONDITIONAL and SET HISTORY commands to modify these at any time. This allows you to override the normal flow of the logic. These commands have the following format.

SET { CONDITIONAL | HISTORY } boolean_value

Where:

boolean_value is one of the following.

- NON-ZERO. Any non-zero number will set the conditional expression or history to TRUE. Instead of a non-zero number, you can supply the boolean state <True>.
- ZERO. Zero will set the conditional expression or history to FALSE. Instead of a non-zero number, you can supply the boolean state <False>.

In response to these commands, the debugger will reprint the current line information.

These commands will only affect the conditional expression result or the result history. They will NOT affect any of the CRISP symbols that were used in the conditional expression. If there is no conditional expression or history on the current line, these commands will have no effect.

In the following example, the conditional expression result is modified to force a statement to execute.

```
Break at TEST\%line 117;        pass 1
Conditional is FALSE - Statement will NOT execute
CSPDBG> set cond 1
At TEST\%line 117;      pass 1
Conditional is TRUE - Statement WILL execute
CSPDBG> _
```

In the following example, the conditional expression result history is modified to force a statement to execute.

```
Break at TEST\%line 136;        pass 2
Conditional is TRUE, History is TRUE - Statement will NOT execute
CSPDBG> set hist 0
At TEST\%line 136;      pass 2
Conditional is TRUE, History is FALSE - Statement WILL execute
CSPDBG> _
```

Modifying Debugger Behavior

Commands are available which are used to modify the Debugger behavior. These are present to allow you to tailor the Debugger to your preferences.

Using the Keypad

Some of the most commonly used commands are assigned by default to keys on the numeric keypad on a DEC VT terminal. This keypad mode can be disabled if you would rather use the numeric keypad for the entry of numerical data.

The SET MODE KEYPAD command enables the use of keypad keys as commands. Appendix A contains a map of the keypad showing the supported keys. The SET MODE NOKEYPAD command sets the keypad in numeric mode.

Input and Output Radix

The SET RADIX command allows you to change the default display method to binary, octal, decimal, unsigned decimal, or hexadecimal. This allows you to examine and modify any CRISP variable in any of these formats. The radix for subscripts is not affected by this command; it is always decimal. The display of String values is not affected by this command; it is always ASCII text.

When the debug session begins, the radix is set to the CRISP default radix.

The SET RADIX command has the following format.

SET RADIX value

Where:

value is one of the following.

- BINARY. Causes the debugger to display the current values of all symbols in unsigned binary. Also, the value supplied in the DEPOSIT command must be in unsigned binary.
- OCTAL. Causes the debugger to display the current values of all symbols in unsigned octal. Also, the value supplied in the DEPOSIT command must be in unsigned octal.
- DECIMAL. Causes the debugger to display the current values of all symbols in signed decimal. Also, the value supplied in the DEPOSIT command must be in signed decimal.
- UNSIGNED. Causes the debugger to display the current values of all symbols in unsigned decimal. Also, the value supplied in the DEPOSIT command must be in unsigned decimal.
- HEXADECIMAL. Causes the debugger to display the current values of all symbols in unsigned hexadecimal. Also, the value supplied in the DEPOSIT command must be in unsigned hexadecimal. Furthermore, the first character in the desired value must be a numeric (0-9) character. If the desired value begins with the letters A - F, you must precede it with a leading zero.

Input and Output Radix (cont)

To return to the CRISP default radix, enter either the SET RADIX DEFAULT or the CANCEL RADIX command. In the CRISP default radix, symbols are treated in the following ways.

- Constants are displayed as Floats, or signed integers
- Counter .Reset and .Count elements are displayed as unsigned decimal integers.
- Floats are displayed in single-precision standard floating-point format. Exponential notation is used to display very large or very small values.
- Logicals are displayed as boolean <True> or <False> states.
- Longs are displayed as signed decimal integers ranging from -2147483648 to 2147483647.
- Numerics are displayed as signed decimal integers ranging from -32768 to 32767.
- Strings are displayed as ASCII text, following the rules of String initialization described in the Examining Strings section.
- Timer .Reset and .Tickdown elements are displayed as unsigned integers. The .Status element is displayed as one of four named states. These are defined in the Examining Timers section.

The current radix can be shown using the SHOW RADIX command.

Presentation

At any time, you can press Ctrl/W to repaint the debugger display. This is useful if the display should become disturbed for any reason.

Cancel

The CANCEL command removes the effect of a specified debugger attribute.

Cancel Break

The CANCEL BREAK command removes a breakpoint from a specified line. To remove a single breakpoint, the format is the following.

CANCEL BREAK %LINE n

To remove all existing breakpoints, the format is the following.

CANCEL BREAK /ALL

Cancel Radix

The CANCEL RADIX command returns the debugger radix for EXAMINEs and DEPOSITs to the CRISP default radix. This command is identical to the SET RADIX DEFAULT command.

Ctrl/C

Ctrl/C is used to interrupt the logic, causing it to break at the next CRISP statement. It is also used to abort a lengthy debugger command currently in progress.

Ctrl/W

Ctrl/W is used to repaint the debugger display if it should become disturbed.

Ctrl/Z

Ctrl/Z causes the debugger to terminate the logic, and exit. It is identical to the EXIT command, and to the F10 key.

Deposit

The DEPOSIT command allows you to modify the current value of the named variable. It has the following format.

DEPOSIT variable[.element] = value

Counters and Timers require a ".element" specification. Arrays require a subscript, which must be specified as a numeric integer constant value. You cannot specify a symbol name, either variable or constant, for a subscript.

Examine

The EXAMINE command causes the debugger to display the type, name, and current value of the named symbol. Any symbol defined in the declarations section of your logic can be examined. Symbols can be variables or constants.

The EXAMINE command has the following format.

EXAMINE symbol

The format of the current value is dependent upon the symbol type and the debugger radix.

Exit

The EXIT command causes the debugger to terminate the logic, and exit. It is identical to Ctrl/Z, and to the F10 key.

Go

The GO command causes the logic to resume execution to the next breakpoint. If no further breakpoints are encountered, the logic will run without interruption. The GO command is bound to the comma (,) key on the numeric keypad on a DEC VT terminal.

The logic can be manually interrupted by pressing Ctrl/C.

Set

The SET command is used to enable the effect of a debugger attribute.

Set Break

The SET BREAK command sets a breakpoint at a specified location in your logic. This causes the debugger to interrupt the execution of the logic at that location, and prompt you for input. It has the following format.

SET BREAK %LINE n

The line number n is the physical line number of the CRISP statement in the source (.C32) program. These line numbers can be seen in the listing file (.LIS) produced by the CRISP compiler if the /LIST qualifier is used.

Set Conditional

The SET CONDITIONAL command allows you to override the normal flow of your CRISP application program by modifying the conditional expression result of the current line. It has the following format.

SET CONDITIONAL boolean_value

Where:

boolean_value can be 0 (zero), <False>, non-zero, or <True>

Set History

The SET HISTORY command allows you to override the normal flow of your CRISP application program by modifying the conditional expression result history of the current line. It has the following format.

SET HISTORY boolean_value

Where:

boolean_value can be 0 (zero), <False>, non-zero, or <True>

Set Mode

The SET MODE command modifies the specified debugger attribute or function. It has the following format.

SET MODE item

The following items are supported.

- KEYPAD. SET MODE KEYPAD enables the use of the keypad for command input. Some of the most commonly used commands are bound to keys on the numeric keypad on a DEC VT terminal.
- NOKEYPAD. SET MODE NOKEYPAD disables the use of the keypad for command input, allowing it to be used for numeric input instead.

Set Radix

The SET RADIX command can be used to modify the radix used for the current value of symbols in the EXAMINE and DEPOSIT commands. It has the format.

SET RADIX value

Where:

value can be BINARY, OCTAL, DECIMAL, HEXADECIMAL, UNSIGNED, or DEFAULT.

The SET RADIX DEFAULT command is identical to the CANCEL RADIX command.

Show

The SHOW command is used to view the current state of the specified debugger or logic attribute.

Show Break

The SHOW BREAK command causes the debugger to list all breakpoints currently set.

Show Conditional

The SHOW CONDITIONAL command will cause the debugger to reprint the current line and pass count. If the current line contains a conditional expression, the conditional expression result and the history will also be printed.

This is identical to the SHOW HISTORY and SHOW %LINE commands.

Show History

The SHOW HISTORY command will cause the debugger to reprint the current line and pass count. If the current line contains a conditional expression, the conditional expression result and the history will also be printed.

This is identical to the SHOW CONDITIONAL and SHOW %LINE commands.

Show %Line

The SHOW %LINE command will cause the debugger to reprint the current line and pass count. If the current line contains a conditional expression, the conditional expression result and the history will also be printed.

This is identical to the SHOW CONDITIONAL and SHOW HISTORY commands.

Show Radix

The SHOW RADIX command causes the debugger to identify the radix in which the current values of symbols will be displayed.

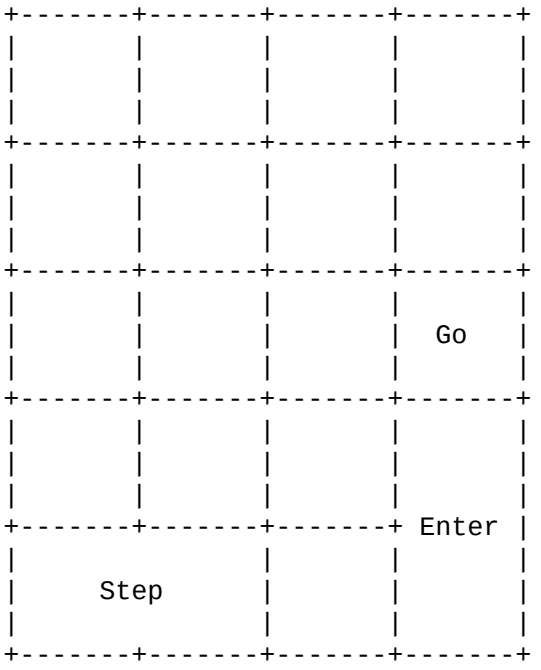
Step

The STEP command causes the logic to resume execution of the current line, and break at the next line. Using the STEP command, it is possible to single step through your entire logic. This command accepts a positive integer as an optional argument, which causes the logic to take multiple steps.

The STEP command is bound to the zero (0) key on the numeric keypad on a DEC VT terminal.

General

The following diagram identifies the commands or functions that are bound to keys on the numeric keypad of DEC VT terminals. These key bindings are activated by the SET MODE KEYPAD command, and disabled by the SET MODE NOKEYPAD command. When the debug session begins, the keypad mode is enabled.



Notes:

CANCEL BREAK %LINE n	Cancel the breakpoint on line n.
CANCEL BREAK /ALL	Cancel all breakpoints.
CANCEL RADIX	Set radix to CRISP default radix.
Ctrl/C	Interrupt logic and break to the debugger. Abort current debugger function in progress.
Ctrl/W	Repaint the debugger screen on a DEC VT terminal.
Ctrl/Z	Terminate logic and exit the debugger.
EXAMINE symbol	Examine the named symbolic variable or constant.
EXIT	Terminate logic and exit the debugger.
DEPOSIT variable[.element] = value	Modify the named symbolic variable to the specified desired value.
GO	Continue logic execution to the next breakpoint.
SET BREAK %LINE n	Set a breakpoint on line n.
SET CONDITIONAL value	Override the conditional expression result.
SET HISTORY value	Override the conditional expression result history.
SET MODE [NO]KEYPAD	Enable or disable use of the numeric keypad.
SET RADIX value	Set display radix to binary, octal, decimal, unsigned decimal, hexadecimal, or default.
SHOW BREAK	Show all breakpoints.
SHOW CONDITIONAL	Show current line, pass count, and conditional expression result and history.
SHOW HISTORY	Show current line, pass count, and conditional expression result and history.
SHOW %LINE	Show current line, pass count, and conditional expression result and history.
SHOW RADIX	Show current radix.
STEP [n]	Complete execution of current line, advance to the next line, and break. The optional argument n is a positive integer which instructs the logic to take n steps.

Notes: