CHAPTER

9

Routine Debugging

This chapter describes the Caché techniques for testing and debugging Caché applications. Topics discussed here include:

- Overview of Debugging page 9-2
- Debugging With BREAK page 9-3
  - Using Argumentless BREAK to Suspend Routine Execution
  - Using Argumented BREAK to Suspend Routine Execution
  - Understanding the Programmer Mode Prompt Information
  - Resuming Execution after a BREAK or an Error
  - The NEW Command in Programmer Mode
  - The QUIT Command in Programmer Mode
  - Caché Error Messages
  - Error Trap Utilities
- Debugging with the Caché Debugger page 9-15
  - Using Breakpoints and Watchpoints
  - Establishing Breakpoints and Watchpoints
  - Disabling Breakpoints and Watchpoints
  - Delaying Execution of Breakpoints and Watchpoints
  - Deleting Breakpoints and Watchpoints
  - Tracing Execution
  - INTERRUPT Keypress and Break
  - Displaying Information About the Current Debug Environment
  - Using the Debug Device
  - Caché Debugger Example

- Understanding Caché Debugger Errors
- Using %STACK to Display the Stack page 9-32
  - Running %STACK
  - Seeing Stack Display Actions
  - Displaying the Process Execution Stack
  - Understanding the Stack Display

Overview of Debugging

An important part of application development is routine debugging: the testing and correcting of program code. Caché gives you two ways to debug your routines:

- Using the BREAK command in routine code to suspend execution and allow you examine what is happening.
- Using the ZBREAK command to invoke the Caché *Debugger* to interrupt execution and allow you to examine both code and variables.

Debugging With BREAK

Caché includes three forms of the BREAK command:

- The BREAK command without an argument inserted into routine code suspends a routine and returns a job to programmer mode.
- The BREAK command with an argument of 1 or 0 to enable and disable interrupts from terminals.
- The BREAK command with special debugging arguments to suspend an Caché routine at a designated location so you can later resume execution at the same (or another) location.

When you sign on in programmer mode, your job begins with an implicit BREAK 1 (interrupt enabled) by default. BREAK functionality is not available in application mode. You establish this default as part of the configuration process. See Chapter 4, *System Configuration* of the Caché System Manager's Guide for more information on configuration.

Using Argumentless BREAK to Suspend Routine Execution

To suspend a running routine and return the job to programmer mode, enter an argumentless BREAK into your routine at points where you want execution to temporarily stop. (Caché accepts only the abbreviation B for the BREAK command.)

When Caché encounters a BREAK, it take the following steps:

1. Suspends the running routine
2. Returns the job to programmer mode,

You can now issue Caché ObjectScript commands, modify data, edit the current (or any other) routine, and execute further routines or subroutines, even those with errors or additional BREAKs.

To resume execution at the point at which the routine was suspended, issue an argumentless GOTO command.

Using Argumentless BREAK with a Condition

You may find it useful to specify a condition on an argumentless BREAK command in code so that you can rerun the same code simply by setting a variable rather than having to change the routine. For example, you may have the following line in a routine:

```
CHECK B:$D(DEBUG)
```

You can then set the variable DEBUG to suspend the routine and return the job to programmer mode or clear the variable DEBUG to continue running the routine.

Using Argumented BREAK to Enable or Disable Interrupts

You can use BREAK with an argument of 1 or 0 to enable or disable interrupts (<CTRL/C>) from the terminal. Entering B 1 at the programmer prompt or including it in source code enables user interrupts with a <CTRL/C>. Entering B 0 at the programmer prompt or including it in source code disables user interrupts with a <CTRL/C>.

Using Argumented BREAK to Suspend Routine Execution

You do not have to place argumentless BREAK commands at every location where you want to suspend your routine. Caché provides several argument variations of the BREAK command that can periodically suspend a routine as if argumentless BREAKs are scattered throughout the code. Variations of the BREAK command arguments are listed in Table 9-1.

Table 9-1: Variations of the BREAK Command

Variation Syntax	Function
B "S"	Use B "S" (Single Step) to step through your code a single command at a time, breaking on every Caché ObjectScript command. The system stops breaking when a DO command, an XECUTE command, a FOR loop, or an extrinsic function is encountered, and resumes single-step breaking when the command, function or loop is done.
B "S+"	B "S+" acts like the B "S" variation except that Caché continues to break on every command when a DO command, XECUTE command, FOR loop, or extrinsic function is encountered.
B “S-”	B “S-” disables single stepping at the current level and enables command stepping at the previous level (acts like B "C" at the current level and B "S" at the previous level)
B "L"	Use B "L" (Line Break) to step through your code a single routine line at a time, breaking at the beginning of every line. The system stops breaking when a DO command, an XECUTE command, or extrinsic function is encountered, and resumes when the command or function is done.
B "L+"	B "L+" acts like B "L", except that Caché continues to break at the beginning of every routine line when a DO command, XECUTE command, or extrinsic function is encountered.

Table 9-1: Variations of the BREAK Command (Continued)

Variation Syntax	Function
B “L-”	B “--” disables single stepping at the current level and enables line stepping at the previous level (acts like B "C" at the current level and B "L" at the previous level)
B "C"	Use B "C" (Clear Break) to stop breaking. Breaking will resume at a higher routine level after the job executes a QUIT if a BREAK state is in effect at that higher level.

Caché stacks the BREAK state whenever a DO, XECUTE, FOR, or extrinsic function is entered. If you choose B "C" to turn off breaking, the system restores the BREAK state at the end of the DO, XECUTE, FOR, or extrinsic function. Otherwise, Caché ignores the stacked state.

Thus if you enable breaking at a low subroutine level, breaking continues after the routine returns to a higher subroutine level. In contrast, if you disable breaking at a low subroutine level that was in effect at a higher level, breaking resumes when you return to that higher level.

When you enter programmer mode, the BREAK state is not stacked. Thus you can change the BREAK state and the new state remains in effect when you issue an argumentless GOTO to return to the executing routine.

Periodic breaking does not occur for lines of code executed in programmer mode or for XECUTE lines started from programmer mode. When you enter programmer mode after a BREAK, you can enter Caché ObjectScript commands and use the line editor without encountering periodic breaking. However, after you issue a DO command or extrinsic function, breaking resumes if B "L+" or B "S+" is in effect.

When B "L" or B "S" is in effect in programmer mode, a DO from programmer mode breaks on the first line or command in the routine, although a subsequent DO does not. Therefore, you can begin debugging by entering a B "L" or B "S" in programmer mode then issuing a DO without specifying B "L+" or B "S+".

Enabling Single Stepping at the Previous Execution Level

Use the BREAK command with “L-” or “S-” to end single stepping at the current execution and enable it at the previous execution level.

These are very similar to the "C" BREAK argument except that the "C" argument doesn't enable stepping at the previous level where single stepping may not have been activated yet.

Understanding the Programmer Mode Prompt Information

When a BREAK command suspends execution of a routine or when an error occurs, the program stack retains some stacked information. When this occurs in programmer mode, a brief summary of this information is displayed before the programmer mode prompt (>).

Such messages take the form:

```
5d3>
```

where:

5	Indicates there are five DO, FOR, EXTRINSIC FUNCTIONS, XECUTE, ERROR, and BREAK states stored on the program stack.
d	Indicates that the last item stacked is a DO.
3	Indicates how many QUITS need to be performed in order to unstack the most recent NEW command, parameter passing, or extrinsic function. This value is a zero if no NEW commands, parameter passing, or extrinsic functions are stacked.

Prompts you can see in such messages are listed in Table 9-2.

Table 9-2: Error Prompts in Programmer Mode

Prompt	Definition
d	DO
e	Extrinsic Function
f	FOR Loop
x	XECUTE
B	BREAK state
E	Error state
S	Sign on state

Resuming Execution after a BREAK or an Error

When entering programmer mode after a BREAK or an error, Caché keeps track of the location of the command that caused the BREAK or

error. Later, you can resume execution at the next command simply by entering an argumentless GOTO in programmer mode:

```
4f0>G
```

By typing a GOTO with an argument, you can resume execution at the beginning of another line in the same routine with the break or error, as follows:

```
4f0>G TAG
```

You can also resume execution at the beginning of a line in a different routine:

```
4f0>G TAB^ROU
```

Alternatively, you may clear the program stack with an argumentless QUIT command:

```
4f0>Q
%SYS>
```

Sample Dialogs

The following routines are used in the examples below.

```
MAIN     ;   03 Jan 97 11:40 AM
         S X=1,Y=6,Z=8
         D SUB1 W !,"SUM=",SUM
         Q
SUB1     ;
         S SUM=X+Y+Z
         Q
```

Examples **Example with BREAK "L"**

With BREAK "L", breaking does not occur in the routine SUB1.

Example with BREAK "L"

```
%SYS>B "L"
%SYS>D ^MAIN
S X=1,Y=6,Z=8
^
<BREAK>MAIN+1^MAIN
2d0>G
D SUB1 W !,"SUM=",SUM
^
<BREAK>MAIN+2^MAIN
2d0>G
SUM=15
Q
^
<BREAK>MAIN+3^MAIN
2d0>G
%SYS>
```

With BREAK "L+", breaking also occurs in the routine SUB1.

```
%SYS>B "L+"
%SYS>D ^MAIN
S X=1,Y=6,Z=8
^
<BREAK>MAIN+1^MAIN
2d0>G
D SUB1 W !,"SUM=",SUM
^
<BREAK>MAIN+2^MAIN
2d0>G
S SUM=X+Y+Z
^
<BREAK>SUB1+1^SUB1
3d0>G
Q
^
<BREAK>SUB1+1^SUB1
3d0>G
SUM=15
Q
^
<BREAK>MAIN+3^MAIN
2d0>G
%SYS>
```

The NEW Command in Programmer Mode

The argumentless NEW command effectively saves all symbols in the symbol table so you can proceed with an empty symbol table. You may find this command particularly valuable when you are in programmer mode after an error or BREAK.

To run other routines without disturbing the symbol table, issue an argumentless NEW command in programmer mode. The system then:

- Stacks the programmer mode frame on the program stack
- Reenters programmer mode.

Example

```
4d0>N
5B1>D ^%T
3:49 PM
5B1>Q 1
4d0>G
```

The 5B1> prompt indicates that the system has stacked the programmer mode entered through a BREAK. The 1 indicates that a NEW command has stacked variable information, which you can remove by issuing a QUIT 1. When you wish to resume execution, issue a QUIT 1 to restore the old symbol table, and a GOTO to resume execution.

Whenever you use a NEW command, parameter passing, or extrinsic function, the system places information on the stack indicating that later an explicit or implicit QUIT at the current subroutine or XECUTE level should delete certain variables and restore the value of others.

In programmer mode, you may find it useful to know if any NEW commands, parameter passing, or extrinsic functions have been executed (thus stacking some variables), and if so, how far back on the stack this information resides.

The QUIT Command in Programmer Mode

In programmer mode you can easily remove all items from the program stack. Simply enter an argumentless QUIT command in programmer mode:

```
4f0>Q
%SYS>
```

If you want to remove only a couple of items from the program stack (for example, to leave a currently executing subroutine and return to a previous DO level), use a QUIT with arguments. QUIT 1 removes the last item on the program stack, QUIT 3 removes the last three items, and so forth, as illustrated below:

```
9f0>Q 3
6d0>
```

Caché Error Messages

Caché displays error messages within angle brackets, as in <ERROR>, followed by a reference to the line that was executing at the time of the error and by the routine. (A caret (^) separates the line reference and routine.) Also displayed is the intermediate code line with a caret

character under the first character of the command executing when the error occurred.

Example

```
S X=Y+3 D ^ABC
^
<UNDEFINED>TAG+3^ROUT
```

This error message indicates an <UNDEFINED> error (that refers to the variable Y) in line TAG+3 of routine ROU. At this point, this message is also, the value of the special variable $ZE.

Error messages that can occur in response to either a programming error in Caché ObjectScript code or a system error are listed in Table B-1 in Appendix B, *Error Messages*.

Error Trap Utilities

The error trap utilities, %ETN and %ERN, help in error analysis by storing variables and recording other pertinent information about an error.

%ETN Application Error Trap

When you set the reserved variable $ZT to an entry reference (such as TAG^ROU), errors adjust the stack and GOTO to that location. You may find it convenient to set the error trap to execute the utility %ETN on an application error. This utility saves valuable information about the job at the time of the error. You can later call the %ERN error report utility to examine the information. Use the following code to set the error trap to this utility:

```
SET $ZT="^%ETN"
```

When an error occurs and you call the %ETN utility, you see a message similar to the following message:

```
Error has occurred: <SYNTAX> at 10:30 AM
```

You may find it useful to set an error trap in an application routine only if it is used in application mode (rather than in programmer mode). The following code sets an error trap only if Caché is in application mode:

```
SET $ZT=$S($ZJ#2:"",1:"^%ETN")
```

%ERN Application Error Report

The %ERN utility examines application errors recorded by the %ETN error trap utility.

Procedure: **Take the following steps to use the %ERN utility:**

1. When prompted, enter the date on which the errors occurred or enter a question mark (?). If you enter “?” in response to the “Enter date:” prompt, you get a list of dates and the number of errors on each date

   When entering a date, use any date format that is accepted by the %DATE utility.

2. When prompted for the error you wish to examine, supply the number of the error you want (1 for the first error, 2 for the second, and so on) or enter a question mark (?).

   If you enter "?" in response to the "Error #:" prompt, %ERN will display a list of further available responses. These responses are shown in the following table:

Available Responses to the "Error #:" Prompt
Select one of the errors for this date.
Enter ?L to list all the errors which are defined for the current date.
Enter * to enter a comment relating to all the errors which exist for this date (e.g. 'all fixed')
Enter tag^routine to list this date's errors which occurred in a specific routine.
Enter [text] to list this date's errors which had 'text' in either the error, line of code, or comment.
Enter <error> to list the errors with the specified error.

3. The utility displays information about the error, including the line of code executed at the time of the error.

4. You are then prompted for a variable. You can now enter either a question mark or one of the responses shown in the following table:

variable name	If you enter the name of a variable, the value of the variable will be displayed. You can examine only nonsubscripted variables this way.
*L	If you enter the *L command, all the old variables will be loaded into your job's partition.
*C	If you enter the *C command, you may then add a comment to the error log.
*	If you enter an asterisk (*), the values of all non-subscripted variables will be displayed.

If you enter a question mark (?) at the variable prompt, you see the list of further options in the following table:

Further Options to Examine Variables after a "?" Response to the "Variable:" Prompt
Enter the name of the variable you wish to view.
Enter the stack level you wish to view.
Enter ?# to view the variables defined for stack level #.
Enter ?var to list levels where variable 'var' is defined.
Enter *S to view all the Process State Variables ($S, etc.).
Enter *F to view the execution Frame Stack.
Enter *C to enter a Comment for this error.
Enter *L to Load the variables into the current partition.
Enter *P to Print the Stack & Symbol information to a device.
Enter *A to print ALL information, state variables, Stack Frames, and Local Variables to a device.
Enter *V to trace selected variables through the frame stack.
Enter *? to redisplay the error information.

5. If you press <RETURN> in response to the Variable: prompt, you return to the Error #: prompt.
6. If you press <RETURN> again, you return to the For Date: prompt.
7. Press <RETURN> a third time to leave the utility.
8. Caché may first ask you:

   ```
   Delete errors older than 30 days? Yes=>
   ```

   Answer **Y** or press <RETURN> to delete old errors.

Example: In the following code, a ZLOAD of the routine REPORT is issued to illustrate that by loading all of the variables with "*LOAD" and then loading the routine, you can recreate the state of the job when the error occurred except that the program stack, which records information about DOs, etc., is empty.

```
%SYS>D ^%ERN

Enter date: ?

30 Dec 95 3 errors
03 Jan 96 2 errors.

For date:12/30/95 3 errors.

Error #: ?
```

(List of available responses appears.)

```
             ?L

1) <DIVIDE>CALC+4^CALC at 01:35PM. Device=70, TRM #70.
   $ZA=0, $ZB="^M", $ZS=20
   S C=R/(F+D-T)

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16PM. Device=70, TRM #70.
   ZA=0, $ZB="^M", $ZS=20
   S ^REPORT(%DAT,TYPE)=I

3) <SYNTAX>ZSET+5^ZSET at 10:34AM. Device=70,
   TRM #70. $ZA=0, $ZB="^M", $ZS=20
   X XSET

Error #: 2

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16PM. Device=70, TRM #70. ZA=0, $ZB="^M",
$ZS=20
   S ^REPORT(%DAT,TYPE)=I

Variable: %DAT="Dec 30 95"

Variable: TYPE=""

Variable: *
%DAT="Dec 30 95"
%DS=""
%TG="REPORT+1"
I="88"
TYPE=""
XY="S $X=250 W *27,*91,DY+1,*59,DX+1,*72 S $X=DX,$Y=DY"

Variable: *LOAD
%SYS>ZL REPORT

%SYS>W

%DAT="Dec 30 95"
%DS=""
%TG="REPORT+1"
I=88
TYPE=""
XY="S $X=250 W *27,*91,DY+1,*59,DX+1,*72 S $X=DX,$Y=DY"
%SYS>
```

Debugging with the Caché Debugger

The Caché Debugger lets you test routines by inserting debugging commands directly into your routine code. Then, when you run the code, you can issue commands to test the conditions and the flow of processing within your application. The Caché Debugger's major capabilities are:

- The ability to set breakpoints with the ZBREAK command at code locations and take specified actions when those points are reached.
- The ability to set watchpoints on local variables and take specified actions when the values of those variables change.
- The ability to interact with Caché during a breakpoint/watchpoint in a separate window.
- The ability to trace execution and output a trace record (to a terminal or other device) whenever the path of execution changes.
- The ability to display the execution stack.
- The ability to run an application on one device while debugging I/O goes to a second device. This enables full screen Caché applications to be debugged without disturbing the application's terminal I/O.

Using Breakpoints and Watchpoints

The Caché Debugger provides two ways to interrupt program execution:

- *Breakpoints*
- *Watchpoints*

A breakpoint is a location in a Caché routine that you specify with the ZBREAK command. When routine execution reaches that line, Caché suspends execution of the routine and, optionally, executes debugging actions you define. You can set breakpoints in up to 20 routines. You can set a maximum of 20 breakpoints within a particular routine.

A watchpoint is a variable you identify in a ZBREAK command. When its value is changed with a SET or KILL command, you can cause the interruption of routine execution and/or the execution of debugging actions you define within the ZBREAK command. You can set a maximum of 20 watchpoints.

Breakpoints and watchpoints you define are not maintained from one session to another. Therefore, you may find it useful to store breakpoint/watchpoint definitions in a routine or XECUTE string so it is easy to reinstate them between sessions.

Establishing Breakpoints and Watchpoints

You use the ZBREAK command to establish breakpoints and watchpoints.

Syntax ***ZB****REAK location[:action:condition:execute_code]*

location	Required. Specifies a code location (that sets a breakpoint) or local or system variable (which sets a watchpoint). If the *location* specified already has a breakpoint/watchpoint defined, the new specification completely replaces the old one.
action	Optional. Specifies the action to take when the breakpoint/watchpoint is triggered. For breakpoints, the action occurs before the line of code is executed. For watchpoints, the action occurs after the command that modifies the local variable. Actions must be enclosed in quotation marks. They may be upper- or lower-case. See "Action Argument Values" on page 9-17.
condition	Specifies an expression that will be evaluated when the breakpoint/watchpoint is triggered. The expression must be surrounded by quotation marks. If the *condition* is false, the *action* will not be carried out and the *execute_code* will not be executed. If a *condition* is not specified, the default is true.
execute_code	Specifies Caché ObjectScript code to be executed if the *condition* is true. The code must be surrounded by quotation marks if it is a literal. This code is executed prior to the *action* being carried out. Before the code is executed, the value of $TEST is saved. After the code has executed, the value of $TEST as it existed in the program being debugged is restored.

Setting Breakpoints with Code Locations

You specify code locations as a routine line reference that you can use in a call to the $TEXT function. A breakpoint occurs whenever execution reaches this point in the code, before the execution of the line of code. If you do not specify a routine name, Caché assumes the reference is to the current routine.

Argumentless GOTO in Breakpoint Execution Code

An argumentless GOTO is allowed in breakpoint execution code. Its effect is equivalent to executing an argumentless GOTO at the debugger BREAK prompt and execution proceeds until the next breakpoint.

Examples If the routine you are testing is in the current namespace, you can enter location values such as these:

tag^*rou*	Break before the line at tag in the routine *rou*.
tag+3^*rou*	Break before the third line after tag in routine *rou*.
+3^*rou*	Break before the third line in routine *rou*.

If the routine you are testing is currently loaded in memory (that is, an implicit or explicit ZLOAD was performed), you can use location values such as these:

tag	Break before the line at tag.
tag+3	Break before the third line after tag.
+3	Break before the third line.

Setting Watchpoints with Local and System Variable Names

Local variable names cause a watchpoint to occur in the following situations:

- When the local variable is created
- When a SET statement changes the local variable's value
- When a KILL statement deletes the local variable

Variable names are preceded by an asterisk, as in *A.

If you specify an array-variable name, the Caché Debugger watches all descendent nodes. For instance, if you establish a watchpoint for array A, a change to A(5) or A(5,1) triggers the watchpoint.

The variable need not exist when you establish the watchpoint.

You can also use the following system variables:

$ZERROR	Triggered whenever an error occurs, before invoking the error trap.
$ZTRAP	Triggered whenever an error trap is set or cleared.
$I	Triggered whenever explicitly SET.

Action Argument Values

Table 9-3 describes the values you can use for the ZBREAK action argument

Table 9-3: Action Argument Values

Argument	Description
"B"	Default, except if you include the "T" action, then you must also explicitly include the "B" action, as in ZB *a:"TB", to actually cause a break Suspends execution and displays the line at which the break occurred along with a caret (^) indicating the point in the line. Then displays the programmer prompt and allows interaction. Execution resumes with an argumentless GO command. For a watchpoint, if the command that initiated the break is at the end of a routine line, the next line in the routine is displayed with the ^ mark at the beginning of the line.
"L"	Same as "B", except GO initiates single-step execution, stopping at the beginning of each line. When a DO command, extrinsic function, or XECUTE command is encountered, single-step mode is suspended until that command or function completes.
"L+"	Same as "B", except GO initiates single-step execution, stopping at the beginning of each line. DO commands, extrinsic functions, and XECUTE commands do not suspend single-step mode.
"S"	Same as "B", except GO initiates single-step execution, stopping at the beginning of each command. When a DO command, extrinsic function, FOR command, or XECUTE command is encountered, single-step mode is suspended until that command or function completes.
"S+"	Same as "B", except GO initiates single-step execution, stopping at the beginning of each command. DO commands, extrinsic functions, FOR commands, and XECUTE commands do not suspend single-step mode.
"T"	Can be used together with any other argument. Outputs a trace message to the trace device. This argument works only after you have set tracing to be ON with the ZBREAK /TRACE:ON command, described later. The trace device is the principal device unless you define it differently in the ZB /TRACE command. If you use this argument with a breakpoint, you see the following message: `TRACE: ZBREAK at tag2^rou2` If you use this argument with a watchpoint, you see a trace message that names the variable being watched and the command being acted upon. In the example below, the variable **a** was being watched. It changed at the line test+1in the routine test. TRACE: ZBREAK SET a=2 at test+1^test If you include the T action, you must also explicitly include the B action as in ZB *A:"TB", to have an actual break occur.
"N"	Take no action at this breakpoint/watchpoint.

ZBREAK Examples

The following example establishes a watchpoint that suspends execution whenever the local variable *A* is killed. No action is specified, so "B" is assumed.

```
ZBREAK *A::"'$D(A)"
```

The following example illustrates the above watchpoint acting on a direct mode Caché command (rather than on a command issued from within a routine). The caret (^) points to the location in the line where execution was suspended.

```
%SYS>K A <Return>
K A
   ^^
<BREAK>0^%rde
%SYS>
```

The following example establishes a breakpoint that initiates single-step execution at the beginning of line TAG^ROU.

```
ZBREAK TAG^ROU:"L"
```

The following example shows how the break would appear when the routine is run. The caret (^) indicates where execution suspended, precedes the line defined in the ZBREAK.

```
TAG     SET X=1
   ^
<BREAK>TAG^ROU
%SYS>
```

In the following example, a breakpoint at line TAG^ROU does not suspend execution, because of the "N" action. However, if X<1 when the line TAG^ROU is reached, then FLAG is SET to X.

```
ZBREAK TAG^ROU:"N":"X<1":"S FLAG=X"
```

The following example establishes a watchpoint that executes the code in ^GLO whenever the value of *A* changes. Note the double colon, indicating no condition argument.

```
ZBREAK *A:"N"::"X ^GLO"
```

The following example establishes a watchpoint that causes a trace message to display whenever the value of *B* changes. The trace message

will display only if trace mode has been turned on with the ZBREAK /TRACE:ON command.

```
ZBREAK *B:"T"
```

The following example establishes a watchpoint that suspends execution in single-step mode when variable *a* is set to 5.

```
ZBREAK *a:"L":"a=5"
```

Note in the next example that when the break occurs, a caret (^) symbol is below the exact location in the line where the break occurred.

```
%SYS> DO ^test< Return>

FOR I=1:1:6 S a=a+1
      ^
<BREAK>test+3^test
3f0> W a <Return>
5
```

Disabling Breakpoints and Watchpoints

You can disable either:

- Specific breakpoints and watchpoints
- All breakpoints or watchpoints

Disabling Specific Breakpoints and Watchpoints

You can disable a breakpoint / watchpoint by preceding the *location* with a minus sign. The following command disables a breakpoint previously specified for location TAG^ROU:

```
ZBREAK -TAG^ROU
```

A disabled breakpoint is "turned off" but Caché remembers its definition. You can enable the disabled breakpoint by preceding the *location* with a plus sign. The following command enables the previously disabled breakpoint:

```
ZBREAK +TAG^ROU
```

Disabling All Breakpoints and Watchpoints

You can disable all breakpoints / watchpoints by using the plus or minus signs without a *location*.

`ZBREAK -`	Disable all defined breakpoints and watchpoints.
`ZBREAK +`	Enable all defined breakpoints and watchpoint.

Delaying Execution of Breakpoints and Watchpoints

You can also delay the execution of a break/watch point for a specified number of iterations. You might have a line of code that appears within a loop that you want to break on periodically, rather than every time it is executed. To do so, follow the *location* argument with a count.

The following ZBREAK command causes the breakpoint at TAG^ROU to be disabled for 100 iterations. On the 101st time this line is executed, the specified breakpoint action occurs.

```
ZBREAK TAG^ROU#100
```

Caution A delayed breakpoint does not work if the line you specify in the location argument of the ZBREAK command is repeated as the first line of a loop.

Deleting Breakpoints and Watchpoints

You can delete individual break/watchpoints by preceding the *location* with a double minus sign.

Example

```
ZBREAK --TAG^ROU
```

After you have deleted a breakpoint/watchpoint, you can only reset it by defining it again.

To delete all points, issue the command:

```
ZBREAK /CLEAR
```

This command is performed automatically when an Caché process halts.

Single-Step Breakpoint Actions

You can use single step execution to stop execution at the beginning of each line or of each command in your code. You can establish a single step breakpoint to specify actions and execution code to be executed at each step. Use the following syntax to define a single step breakpoint:

```
ZBREAK $:action[:condition:execute_code]
```

Unlike other breakpoints, ZBREAK $ does not cause a break, because breaks occur automatically as you single-step. ZBREAK $ allows you to specify actions and execute code at each point where the debugger breaks as you step through the routine. It is especially useful in tracing executed

lines or commands. For example, to trace executed lines in the application ^TEST:

```
%SYS>ZBREAK /TRACE:ON
%SYS>BREAK "L+"
%SYS>ZBREAK $:"T"
```

The "T" action specified alone suppresses the single step break that normally occurs automatically. The "N" action code also suppresses the single step break that normally occurs. Establish the following single step breakpoint definition if both tracing and breaking should occur:

```
%SYS>ZBREAK $:"TB"
```

Tracing Execution

You can control whether or not the "T" action of the ZBREAK command is enabled by using the following form of ZBREAK:

```
ZBREAK /TRACE:state[:device]
```

where *state* can be:

ON — to enable tracing

OFF — to disable tracing

ALL — to enable tracing of all application lines by performing the equivalent of:

```
ZBREAK /TRACE:ON[:filename]
BREAK "L+"
ZBREAK $:"T"
```

When *device* is used with the ALL or ON state keywords, trace messages are redirected to the specified device rather than to the principal device. If the device is not already open, Caché attempts to open it as a sequential file with WRITE and APPEND options.

When *device* is specified with the OFF state keyword, Caché closes the file if it is currently open.

Note ZBREAK /TRACE:OFF does not delete or disable the single step breakpoint definition set up by ZBREAK /TRACE:ALL, nor does it clear the "L+" single stepping set up by ZBREAK /TRACE:ALL. If you want to start debugging after switching off tracing, delete or disable the single step breakpoint definition and change the single step setting.

Tracing messages are generated at breakpoints associated with a "T" action. With one exception, the trace message format is as follows for all breakpoints:

```
Trace: ZBREAK at <line_reference> TRACE
```

where <*line_reference*> is the line reference of the breakpoint

The trace message format is slightly different for single step breakpoints when stepping is done by command:

`Trace: ZBREAK at` <*line_reference*> <*source_offset*> `TRACE`

<*line_reference*> line reference of the breakpoint

<*source_offset*> 0-based offset to the location in the source line where the break has occurred

OpenVMS Procedure **To send TRACE messages to another device on OpenVMS platforms:**

1. Log in on TTA1:.
2. Verify the device name by entering the following command at the DCL prompt:

```
$ WRITE sys$output f$getjpi("","TERMINAL")
TA1:
```

3. Issue the PROTECTION command so you have write privileges to the terminal:

```
$ SET PROT=W:rwlp TTA1:
```

4. Issue the following command to avoid contention for the device:

```
$ WAIT 1
```

5. Return to your working window or to a terminal where you are logged in on your principal device.
6. Issue the following command to set your process privileges to share.

```
$ SET PROC/PRIV=share
```

7. Start M.
8. Issue your TRACE command:

```
ZB /T:ON:"TTA1:"
```

9. Run your program.

   If you have set breakpoints or watchpoints with the "T" action, you will see trace messages appear on the window connected to TTA1:.

UNIX Procedure **To send TRACE messages to another device on UNIX platforms:**

1. Log into /dev/tty01.
2. Verify the device name by entering the following command at the UNIX prompt.

   $ ***tty***

   /dev/tty01
3. Issue the following command to avoid contention for the device:

   $ ***exec sleep 50000***
4. Return to your working window.
5. Start and enter M.
6. Issue your TRACE command:

   > ***ZB /T:ON:"/dev/tty01"***
7. Run your program.

If you have set breakpoints or watchpoints with the "T" action, you see trace messages appear in the window connected to /dev/tty01.

NT/95 TRACE messages to another device are supported on Windows 95 and NT platforms only for terminal devices connected to a COM port, such as COM1:. You cannot use the console or a terminal window. You can specify a sequential file for the trace device.

Trace Message Format

If you set a code breakpoint, the following message appears

```
Trace:  ZBREAK at tag2^rou2
```

If you set a variable breakpoint, one of the following messages appears:

```
Trace:  ZBREAK SET var=val at tag2^rou2
Trace:  ZBREAK SET var=array val at tag2^rou2
Trace:  ZBREAK KILL var at tag2^rou2
```

- `var` is the variable being watched
- `val` is the new value being set for that variable.

If you issue a NEW command, you receive no TRACE message. However, the Trace on the variable is triggered the next time you issue a SET or KILL on the variable at the NEW level. If a variable is passed by

reference to a routine, then that variable is still traced, even though the name has effectively changed.

INTERRUPT Keypress and Break

Normally, pressing the interrupt key sequence (typically <CTRL/C>) generates a trappable (<INTERRUPT>) error. To set interrupt processing to cause a break instead of an <INTERRUPT> error, use the following ZBREAK command:

```
%SYS> ZBREAK /INTERRUPT:BREAK
```

This causes a break to occur when you press the INTERRUPT key even if you have disabled breaks at the application level for the device.

If you press the INTERRUPT key during a read from the terminal, you may have to press <ENTER> to display the break-mode prompt. To reset interrupt processing to generate an error rather than cause a break, issue the following command:

```
%SYS> ZBREAK /INTERRUPT:NORMAL
```

Displaying Information About the Current Debug Environment

To display information about the current debug environment, including all currently defined break/watchpoints, issue the ZBREAK command with no arguments:

```
%SYS> ZBREAK
```

The argumentless ZBREAK command describes the following aspects of the debug environment:

- Whether <CTRL-C> causes a break
- Whether trace output specified with the "T" action in the ZBREAK command displays
- The location of all defined breakpoints, with flags describing their enabled/disabled status, action, condition and executable code
- All variables for which there are watchpoints, with flags describing their enabled/disabled status, action, condition and executable code

Output from this command is displayed on the device you have defined as your debug device, which is your principal device unless you have defined the debug device differently with the ZBREAK /DEBUG command (see “Using the Debug Device” on page 9-28).

Table 9-4 describes the flags provided for each breakpoint and watchpoint.

Table 9-4: Information in Display of Breakpoints and Watchpoints

Display Section	Meaning
Identification of break/watch point	Line in routine for breakpoint. Local variable for watchpoint.
F:	Flag providing information about the type of action defined in the ZBREAK command.
S:	The number of iterations to delay execution of a breakpoint/watchpoint defined in a ZBREAK - command.
C:	Condition argument set in ZBREAK command.
E:	Execute_code argument set in ZBREAK command.

Table 9-5 describes how to interpret the F: value in a breakpoint/watchpoint display. The F: value is a list of the applicable values in the first column.

Table 9-5: Flag Values

Value	Meaning
E	Breakpoint or watchpoint enabled
D	Breakpoint or watchpoint disabled
B	Perform a break
L	Perform an "L"
L+	Perform an "L+"
S	Perform an "S"
S+	Perform an "S+"
T	Output a Trace Message

Default Display

Figure 9-1 shows the output when you first enter M:

- Trace execution is OFF
- There is no break if <CTRL-C> is pressed
- No break/watchpoints are defined

```
%SYS> ZB
BREAK:
No breakpoints
No watchpoints
```

Figure 9-1: Display When No Breakpoints or Watchpoints Exist

Display When Breakpoints and Watchpoints Exist

Figure 9-2 shows two breakpoints and one watchpoint being defined. The first two ZBREAK commands define a delayed breakpoint; the second two ZBREAK commands define a disabled breakpoint; the fifth ZBREAK command defines a watchpoint. The sixth ZBREAK command enables trace execution. The final ZBREAK command, with no arguments, displays information about current debug settings.

```
%SYS> ZB +3^test :"WRITE ""IN test"""
%SYS> ZB -+3^test#5
%SYS> ZB +5^test:"L"
%SYS> ZB -+5^test
%SYS> ZB *a:"T":"a=5"
%SYS> ZB /TRACE:ON
%SYS> ZB
BREAK:TRACE ON
+3^test F:EB S:5 C: “E:”WRITE ""IN test""“
+5^test F:DL S:0 C: E:
a F:ET S:0 C:”a=5” E:
```

Figure 9-2: Display When Breakpoints and Watchpoints Exist

In Figure 9-2, the ZBREAK display shows that:

- Tracing is ON
- There is no break if <CTRL-C> is pressed.

The output in Figure 9-2 then describes the two breakpoints and one watchpoint:

- The F flag for the first breakpoint equals "EB" and the S flag equals 5, which means that a breakpoint will occur the fifth time the line is encountered. The E flag displays executable code, which will run before the Caché programmer prompt for the break is displayed.
- The F flag for the second breakpoint equals "DL", which means it is disabled, but if enabled will break and then single-step through each line of code following the breakpoint location.
- The F flag for the watchpoint is "ET", which means the watchpoint is enabled. Since trace execution is ON, trace messages will appear on the trace device. Since no trace device was defined, the trace device will be the principal device.
- The C flag means that trace is displayed only when the condition is true.

Using the Debug Device

The debug device is the device where:

- The ZBREAK command displays information about the debug environment.
- The Caché programmer prompt appears if a break occurs.

95/NT

- Windows 95 and NT platforms: TRACE messages to another device are supported on only for terminal devices connected to a COM port, such as COM1:

Examples

When you enter M, the debug device will automatically be set to your principal device. At any time, debugging I/O can be sent to an alternate device with the command:

```
> ZB /DEBUG:"device"
```

Alpha

To cause the break to occur in an X window linked to the device TTA1:, issue the following command on an OpenVMS system:

```
> ZB /D:"TTA1:"
```

UNIX

To accomplish the same end, issue the following command on a UNIX platform:

```
> ZB /D:"/dev/tty01/"
```

When a break occurs, because of a <CTRL-C> or to a breakpoint or watchpoint being triggered, it appears in the window connected to the device. That window becomes the active window.

If the device is not already open, an automatic OPEN is performed. If the device is already open, any existing OPEN parameters are respected.

Caution If the device you specify is not an interactive device, such as a terminal, you are not be able to return from a break. However, the system does not enforce this restriction.

Caché Debugger Example

First, suppose you are debugging the simple program named **test** shown in Figure 9-3 below. The goal is to put 1 in variable a, 2 in variable b and 3 in variable c.

```
test;Assign the values 1,2 and 3 to the variables a,b, and c, respectively
    S a=1
    S b=2
    S c=3 K a WRITE "in test, at end"
    QUIT
```

Figure 9-3: Routine *test*

However, when you run **test**, only variables b and c hold the correct values.

```
%SYS> DO ^test
in test, at end
%SYS>W
b=2
c=3
%SYS>
```

The problem in the program is obvious: variable a is KILLed on line 5. However, assume you need to use the debugger to determine this.

Example You can use the ZBREAK command to set single-stepping through each line of code ("L" action) in the routine **test**. By a combination of stepping and writing the value of a, you determine that the problem lies in line 5.

```
%SYS> NEW

1S1> ZB
BREAK:

No breakpoints
No watchpoints

1S1>ZB ^test:"L"

1S1> DO ^test

 S a=1
 ^
<BREAK>test+1^test
3d3> W a
```

```
1
3d3> G

 S b=2
 ^
<BREAK>test+1^test
3d3> W a

<UNDEFINED>

3d3> G
 S b=2
 ^
<BREAK>test+1^test
3d3> W a

1
3d3> G
 S c=3 K a WRITE "in test, at end"
 ^
<BREAK>test+5^test
3d3> W a

1
3d3> G
 QUIT
 ^
<BREAK>test+6^test
3d3> W a

<UNDEFINED>
3d3> G
1S1>
```

You can now examine that line and notice the **KILL a** command. In more complex code, you might now want to single-step by command ("S" action) through that line.

If the problem occurred within a DO, FOR or XECUTE command or extrinsic function, you would use the "L+" or "S+" actions to single-step through lines or commands within the lower level of code.

Understanding Caché Debugger Errors

The Caché Debugger flags an error in a condition or execute argument with an appropriate Caché error message.

If the error is in the execute code parameter, then the condition surrounds the execute code when the execute code is displayed prior to the error message. The condition ($TEST) is always set back to one at the end of the execution code so that the rest of the debugger processing code works properly. When control returns to the routine, the value of $TEST within the routine is restored.

Suppose you issue the following ZBREAK command for the example program **test** discussed in “Caché Debugger Example” on page 9-29:

```
%SYS> ZBREAK test+1^test:"B":"a=5":"WRITE b"
```

In the program **test**, variable *b* is not defined at line test+1, so there is an error. The error display appears as in Figure 9-4.

```
i a=5 X "WRITE b" i 1
^
<UNDEFINED>test+1^test
```

Figure 9-4: Error in ZBREAK execution_code Argument

If you had not defined a condition, then an artificial true condition would be defined prior to and after the execution code, as below:

```
%SYS> i 1 WRITE b i 1
```

Using %STACK to Display the Stack

You can use the %STACK utility to:

- Display the contents of the process execution stack
- Display the values of local variables, including values that have been "hidden" with the NEW command or through parameter passing.
- Display the values of process state variables, such as $IO and $JOB

Running %STACK

You execute %STACK by entering the following command:

```
%SYS> DO ^%STACK
```

As shown in Figure 9-5, the %STACK utility displays the current process stack without variables. You can redisplay the current execution stack without variables at any time by entering **F* at the "Stack Display Action" prompt.

Under the current execution stack display, %STACK prompts you for a stack display action.

```
Level  Type          Line                    Source
 1 SIGN ON
  2   DO            LoadStk+13^%STACK    ~DO TEST^%STACK
      NEW ALL/EXCL                       NEW (E)
  3   DO            TEST^%STACKD  TEST   K  N (E) S A=1 ~D TEST1 Q  ;level=2
NEW                                   NEW A
  4   DO            TEST1^%STACKD  TEST1 N A S (B,A)=2 ~DO  ;level = 3
      NEW                                NEW A,B
  5   DO           TEST1+1^%STACKD      . N A,B S (C,B,A)=3 ~DO  ;level= 4
NEW                                     NEW A,B,C
      ERROR TRAP                         S $ZTRAP="TESTQ^%STACKD"
 6 DO     TEST1+3^%STACKD  . . ~S (D,C,B,A)=4 DO ;level= 5
      NEW                                NEW A,B,C,D
  7   DO            TEST1+4^%STACKD      . . . N A,B,C,D ~S (E,D,C,B,A)
=5 D TEST2 ;level 6
      NEW                                NEW XEC
  8   XECUTE        TEST2^%STACKD  TEST2 N XEC S XEC="D TEST3^"_$ZN ~X XEC Q
;level 7
  9   DO            ^%STACKD             ~D TEST3^%STACKD
 10   DO            TEST3^%STACKD  TEST3 ~D TESTD(A,,C,.D) Q
```

Figure 9-5: Initial %STACK Display

Seeing Stack Display Actions

You can see the possible stack display actions by entering *?* at the "Stack Display Action" prompt, as shown in Figure 9-6.

```
Stack Display Action: ?

    Enter ?# to view the variables defined for stack level #
    Enter ?? to view all the stack levels with variables.
    Enter *S to view all the Process State Variables ($S,etc)
    Enter *F to view the Process Execution Frame Stack
    Enter *V to trace selected Variables through the Execution
             stack.
    Enter *P to Print the Stack & Symbol information to a device
    Enter *A to Print ALL information, State variables, Stack
     Frames, and local variables to a device.
Stack Display Action:
```

Figure 9-6: Stack Display Actions

Displaying the Process Execution Stack

Depending on what you enter at the “Stack Display Action” prompt, you can display the current process execution stack in four forms:

- Without variables, by entering **F*
- With a specific variable, by entering **V*
- With all variables, by entering **P*
- With all variables, preceded by a list of process state variables, by entering **A*

Displaying the Stack without Variables

Figure 9-5 shows a sample display of the process execution stack without variables as it appears when you first enter the %STACK utility or when you select the stack action **F*.

Displaying the Stack with a Specific Variable

Enter **V* at the "Stack Display Action" prompt, followed by the name of the variable you want to track through the stack. In Figure 9-7, the variable E is being tracked and the display is sent to the screen by pressing <RETURN> at the "Device:" prompt.

```
Stack Display Action: *V

Variable(s): E

Display on
Device: <RETURN>
```

Figure 9-7: Track Specific Variable in Stack

Figure 9-8 shows the screen stack display.

```
Level    Type        Line                  Source
  1      SIGN ON
                        E         =                5
    2    DO   LoadStk+13^%STACK     ~DO TEST^%STACK

         NEW ALL/EXCL                       NEW (E)

    3    DO   TEST^%STACKD  TEST                K  N (E) S A=1 ~D TEST1 Q
;level=2
      NEW                                 NEW A
    4    DO   TEST1^%STACKD  TEST1  N A S (B,A)=2 ~DO  ;level = 3

      NEW                                 NEW A,B

    5    DO   TEST1+1^%STACKD                   NA,BS(C,B,A)=3~DO;level=4
--more--
```

Figure 9-8: Sample Display When Tracking Variable E

Displaying the Stack with All Defined Variables

Enter **P* to see the process execution stack together with the current values of all defined variables.

Displaying the Stack with All Variables, including State Variables

You can print all possible reports to screen, file or printer by entering **A* at the "Stack Display Action" prompt. This report prints the following:

- Process state variables
- Process execution stack with all variables

Understanding the Stack Display

Each item on the stack is called a *frame*. Table 9-6 describes the information provided for each frame.

Table 9-6: %STACK Utility Information

Heading	Description
Level	Identifies the level within the stack. The oldest item on the stack is number 1. Frames without an associated level number share the level that first appears above them.
Type	Identifies the type of frame on the stack, which can be: DIRECT BREAK – A BREAK command was encountered that caused a return to direct mode. DIRECT CALLIN – A Caché process was initiated from an application outside of Caché, using the Caché call-in interface. DIRECT ERROR – An error was encountered that caused a return to direct mode. DO – A DO command was executed. ERROR TRAP – If a routine sets $ZTRAP, this frame identifies the location where an error will cause execution to continue. FOR – A FOR command was executed. NEW – A NEW command was executed. If the NEW command had arguments, they are shown. SIGN ON – Execution of the Caché process was initiated. XECUTE – An XECUTE command was executed. $$EXTFUNC – An extrinsic function was executed.
Line	Identifies the Caché ObjectScript source line associated with the frame, if available, in the format *tag+offset^routine*.
Source	Shows the source code for the line, if it is available. If the source is too long to display in the area provided, horizontal scrolling is available. If the device is line-oriented, the source wraps around and continued lines are preceded with "...".

Table 9-7 shows whether level, line, and source values are available for each frame type. A "No" under Level indicates that the level number is not incremented and no level number appears in the display.

Table 9-7: Frame Types and Values Available

Frame Type	Level	Line	Source
DIRECT BREAK	Yes	Yes	Yes
DIRECT CALL IN	Yes	No	No
DIRECT ERROR	Yes	Yes	Yes
DO	Yes	Yes*	Yes
ERROR TRAP	No	No	No, but the new $ZT value is shown.
FOR	No	Yes	Yes

Table 9-7: Frame Types and Values Available (Continued)

Frame Type	Level	Line	Source
NEW	No	No	Shows the form of the NEW (inclusive or exclusive) and the variables affected.
PARAMETER	No	No	Shows the formal parameter list. If a parameter is passed by reference, shows what other variables point to the same memory location.
SIGN ON	Yes	No	No
XECUTE	Yes	Yes*	Yes
$$EXTFUNC	Yes	Yes*	Yes
* The LINE value is blank if these are invoked from programmer mode.			

Moving through %STACK Display

If a %STACK display fills more than one screen, then you see the prompt "-- more --" in the bottom left corner of the screen. At the last page, you see the prompt "-- fini --". Type *?* to see key presses you use to maneuver through the %STACK display.

```
- - - Filter Help - - -

    <space>  Display next page.
    <return> Display one more line.
       T     Return to the beginning of the output.
       B     Back up one page (or many if arg>1).
       R     Redraw the current page.

     /text   Search for 'text' after the current page.
       A     View all the remaining text.
       Q     Quit.
       ?     Display this screen

       #     specify an argument for B, L, or W actions.
       L     set the page length to the current argument.
       W     set the page width to the current argument.
```

Figure 9-9: - more - Help Screen

You enter any of the commands listed above whenever you see the "--more--" or "--fini--" prompts.

For the B, L and W commands, you enter a numeric argument prior to the command letter. For instance, enter 2B to move back two pages, or enter 20L to set the page length to 20 lines.

Be sure to set your page length to the number of lines which are actually displayed; otherwise, when you do a page up or down, some lines may not be visible. The default page length is 23.

Displaying Variables at Specific Stack Level

To see the variables that exist at a given stack frame level, enter ?# at the "Stack Display Action" prompt, where # is the stack frame level. For example, Figure 9-10 shows the display if you request the variables at level 1.

```
Stack Display Action: ?1

The following Variables are defined for Stack Level: 1
E

Stack Display Action:
```

Figure 9-10: Variable Display by Level in %STACK Utility

Displaying Stack Levels with Variables

You can display the variables defined at all stack levels by entering ?? at the "Stack Display Action" prompt. Figure 9-11 shows a sample display if you select this action.

```
Stack Display Action: ??
Now loading variable information ...  19

Base Stack Level: 5
A

Base Stack Level: 3
A    B    C    D

Base Stack Level: 1
E

Stack Display Action:
```

Figure 9-11: Select Level/Variable to Display in %STACK

Displaying Process State Variables

To display the process state variables, such as $IO, enter *S at the "Stack Display Action" prompt. You will see the defined variables as shown in Figure 9-12. This display may appear on two screens on your terminal.

```
Process State Intrinsics:
        $D =
       $EL =
       $ES =
        $H = 55574,43548
        $I = TTA0:
        $J = 20592
        $K =
        $P =
        $R =
        $S = 236016
       $SY =
        $T = 0
       $TL =
       $TR =
        $X = 0
        $Y = 14
       $ZA = 0
       $ZB = $c(13)
      $ZE = <UNDEFINED>RestST+3^%STACK
      $ZR = ^mtemp(33)
        $ZT = TESTQ^%STACKD
  $ZU(100) =
   $ZU(12) = DUA0:["sys1"]
   $ZU(18) = 0
   $ZU(20) = DUA0:["sys1"]
   $ZU(39) = DUA0:["sys1"]
    $ZU(5) = DUA0:["sys1"]
   $ZU(55) = 0
 $ZU(68,1) = 0
 $ZU(68,5) = 1
 $ZU(68,6) = 0
 $ZU(68,7) = 0

--fini--
```

Figure 9-12: State Variables in %STACK Utility

Printing the Stack and/or Variables

When you select the following actions, you can choose the output device:

- *P
- *A
- *V after selecting the variables you want to display.