debugapi: Debug API User Guide and Reference Version 1

CML00060-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

August 16, 2016

Contents

1 Introduction

The Code Magus Limited `debugapi` Library is used in script engines to implement a debugging interface for that script engine. The actual debugger is implemented as a combination of the functions of the script engine and the `debugapi` library.

The `debugapi` library provides an environment for interacting with a script being debugged by providing a TCP/IP command interface to the script engine running a script in debug mode. The commands passed through to the command interface are parsed by the `debugapi` library and any interaction with the script execution environment is accomplished by the `debugapi` library calling functions supplied by the script engine.

Figure 1 on page 2 illustrates the relationship of the `debugapi` library and the script engine into which the `debugapi` has been integrated. The integration of the `debugapi` into a script engine makes a socket access point available for debug commands.

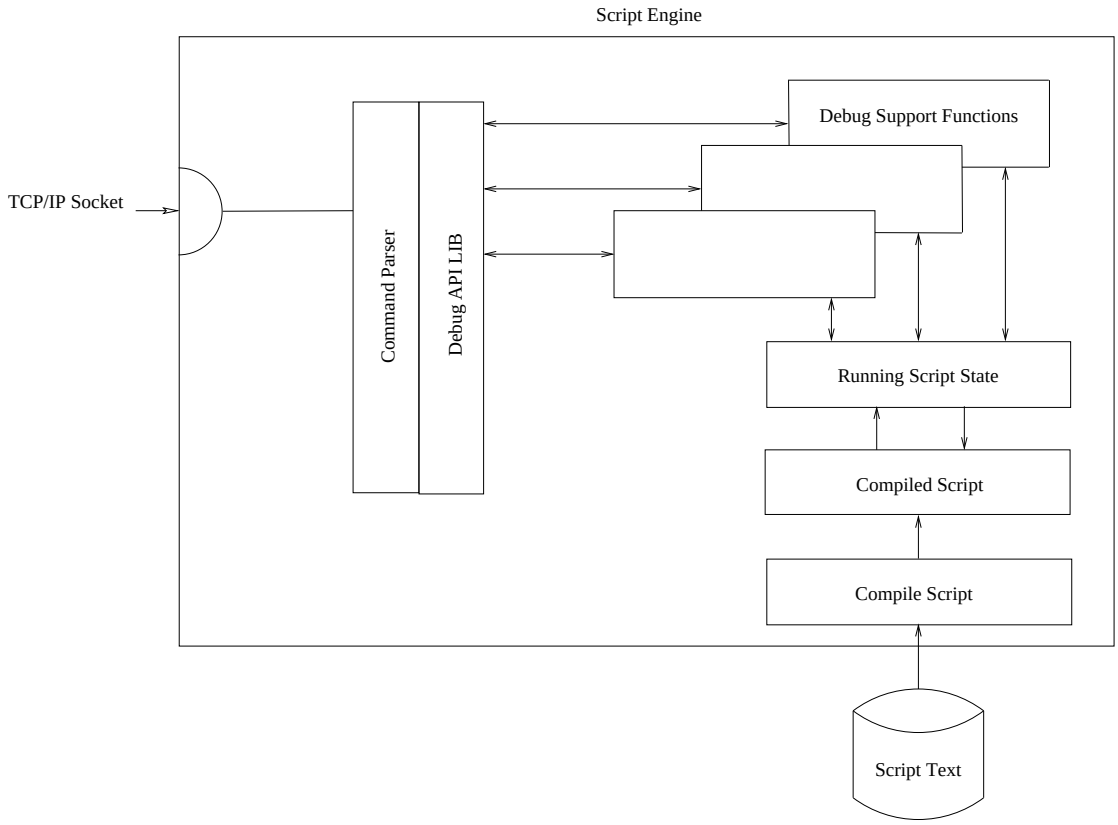


Figure 1: Relationship of `debugapi` library and script engine and exposed socket interface for debug commands

The `debugapi` library is written in a portable style, and because user interaction with the `debugapi` library is through TCP/IP, debugging is supported across many system types, including Linux/Unix, z/OS, Windows, and Stratus VOS.

The style of communication used by the `debugapi` library when interacting through the user interface is intended to allow the item being debugged, and hence the debug session, to execute within an Integrated Development Environment (or IDE). To support this, the initialisation of the `debugapi` within a script engine needs to include some indication of the means by which the user or environment (if it is an IDE) communicates with the debugger. This is illustrated in Figure 2 on page 3. The `debugapi` library will connect to a given host and port number, or will bind to a given or non-specific host and/or port number and then listen on the socket for an incoming debug session. Using this scheme the `debugapi` library also supports remote debugging as illustrated in figure 3 on page 4.

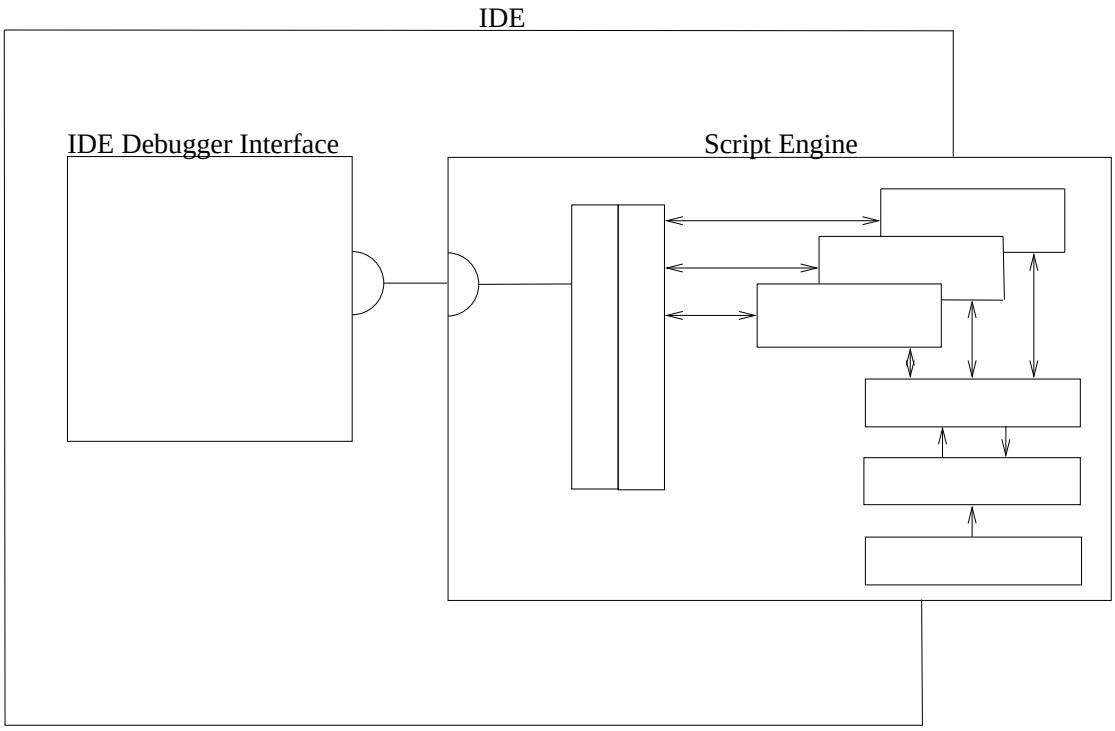


Figure 2: Integration of the script engine into an IDE supporting a GUI for debugging using the `debugapi`

The commands to the debugger, which are entered as clear text by the user or by an IDE through the TCP/IP interface, are parsed and interpreted by the `debugapi` interface and or the script engine which provides appropriate call-back functions to support this.

The commands are designed to be simple and their responses are intended to be parsed in turn by the IDE issuing them, as well as to be interpreted by a user of the interface.

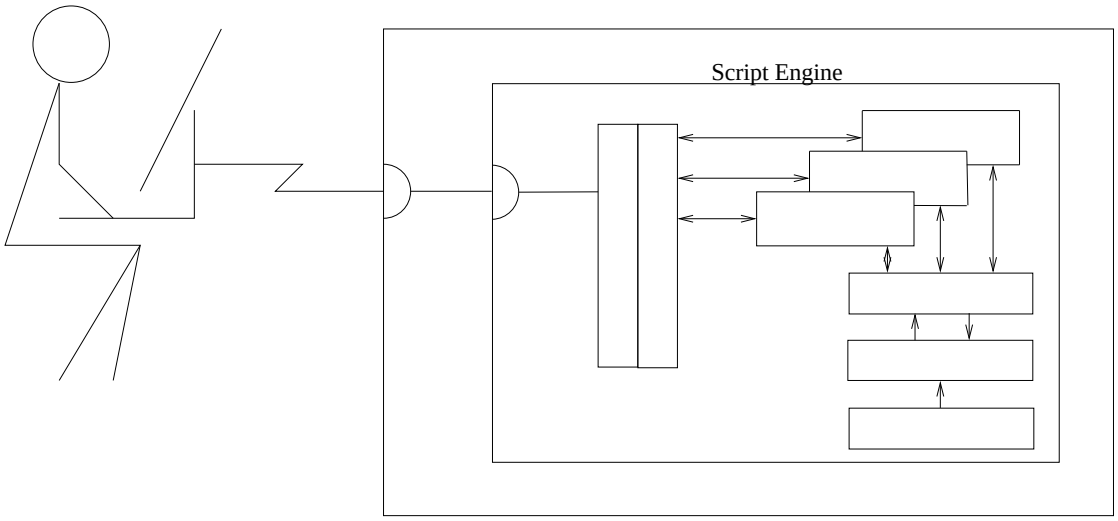


Figure 3: Instance of the `debugapi` into a script engine provides support for remote debugging

2 Debug API Command Interface

This section describes the debug commands that can be used once the call interface and the socket interface have been established.

The commands that a user (either a person or an integrated IDE) would issue over the `debugapi` debug socket are all clear text commands and all responses are clear text responses. The functionality available, and hence the types of commands available, is similar to what one would expect from any debugger. Commands are provided for setting and clearing break points; showing and setting the values of variables; stepping into, over and out of the sequences of statements; showing the current location; etc.

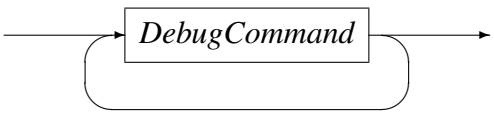
Each command starts with a verb indicating the required function. A command may be entered in either upper or lower case, or in a mixed case. The case of the verb has no significance, but the case of the operands may have some significance and, in those cases, the context will determine this. For example, the name of a file will have some significance depending on whether this has some significance to the local machine. Most command verbs also have abbreviations simplifying usage of the `debugapi` command interface.

The `debugapi` always prompts the user of the interface, indicating that the script in the script engine being debugged is in the stopped state and that a debug command may be entered. This prompt is indicated by the string "`DEBUG>`" as illustrated below:

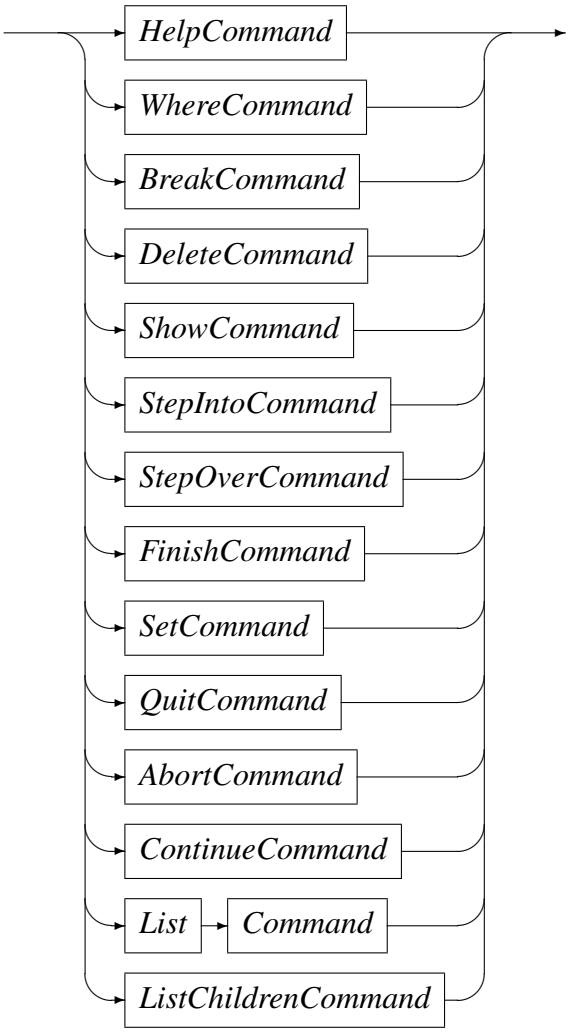
```
[stephen@nomad ~]$ telnet localhost 59212
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
VERIFY: Code Magus Limited Verify Rule Check System V1.0
build 2010-02-23-16.19.46
CASATEST.vcf:1278
DEBUG>
```

A debug session is made up of a series of commands that interact with the script engine during the execution of a script. There are two special cases of input to the `debugapi` command interface. Comments may be entered at the debug command prompt, and are introduced using the # character, and continue until the end of the line. An empty command string indicates that the last successful command (if such a command exists) is to be re-executed.

DebugSession



DebugCommand



2.1 The `HELP`-Command

The `HELP`-command provides basic help on available commands. Entering `HELP` on `debugapi` command line on its own displays a list of available commands for which there is available help:

```
DEBUG> help
Enter the following on a line by itself:
Try HELP {WHERE|BREAK|DELETE|SHOW|STEPINTO|STEPOVER|STEPOUT|SET|QUIT|CONTINUE}
This will give help about the specific command.
DEBUG>
```

The `HELP`-command may be abbreviated as "`H`":

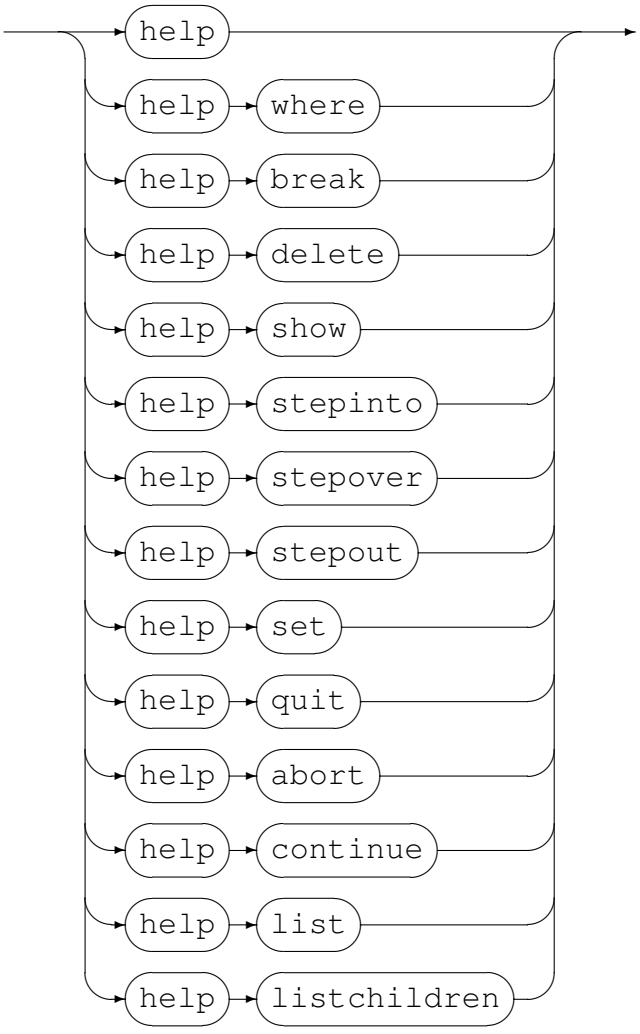
```
DEBUG> h where
Enter the following on a line by itself:
```

```
WHERE
Show the location of the next executable statement.
DEBUG>
```

The command for which help is required can also be abbreviated:

```
DEBUG> h w
Enter the following on a line by itself:
WHERE
Show the location of the next executable statement.
DEBUG>
```

HelpCommand



2.2 The `WHERE`-Command

The `WHERE`-command displays the file name and line number in that file of the current statement execution position. This is the position of the statement that would ordinarily

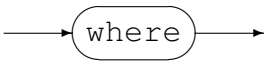
be the next statement to execute. For example, the result of the following WHERE-command

```
DEBUG> where
CASATEST.vcf:1278
DEBUG>
```

indicates that the next statement in the script currently in debug mode within the script engine is in file CASATEST.vcf and line number 1278.

The WHERE-command may be abbreviated as "W".

WhereCommand



2.3 The BREAK-Command

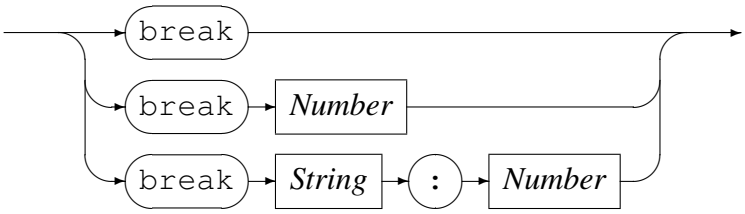
The BREAK-command is used to set a break point. It should be possible to set a break point on any executable statement. The BREAK-command responds with the file name and line number of the executable statement that the actual break point was set on. This may be something slightly different from the break point requested. For example, it is acceptable for a break point requested on a non-executable line, or a line on which an executable statement does not start, to be set on the next executable statement at which a break point can be set or on the position at which the executable statement does start.

```
DEBUG> break
BREAKPOINT SET AT "CASATEST.vcf":1278
DEBUG>
```

```
DEBUG> break "CASATEST.vcf":1291
BREAKPOINT SET AT CASATEST.vcf:1291
DEBUG>
```

The BREAK-command may be abbreviated as "B".

BreakCommand



If the file name is not supplied on the BREAK-command then the name of the current file is used. The current file is the file containing the next statement that will be executed. If the line number is not supplied on the BREAK-command then the line number of the next statement in the current file will be used.

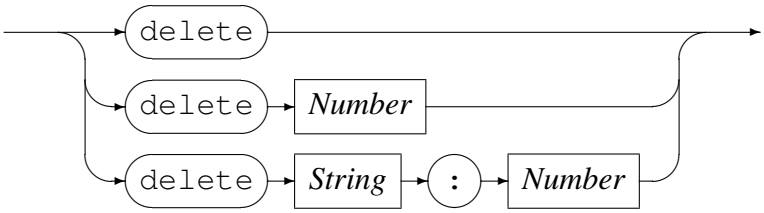
2.4 The DELETE-Command

The DELETE-command is used to remove a previously set break point. The command responds with the position of the executable statement (file name and line number) at which the actual break point was removed. This might be different from the position requested on the DELETE-command.

```
DEBUG> delete "CASATEST.vcf":1278
BREAKPOINT DELETED AT "CASATEST.vcf":1278
DEBUG>
```

The DELETE-command may be abbreviated as "DEL".

DeleteCommand



If the file name is not supplied on the DELETE-command then the name of the current file is used. The current file is the file containing the next statement that will be executed. If the line number is not supplied on the DELETE-command then the line number of the next statement in the current file will be used.

2.5 The SHOW-Command

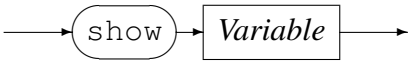
The SHOW-command takes as an argument a variable name. If this variable exists and is currently within context in terms of the script being debugged, then the command responds with the value of that variable.

```
DEBUG> show total_system_fee
VALUE OF "total_system_fee" IS 17810.00
DEBUG>
```

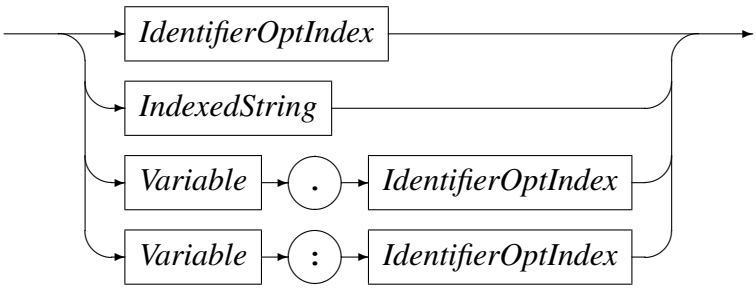
```
DEBUG> show test_case.JOIN_RECORD.JOIN_CHARGE
VALUE OF "test_case.JOIN_RECORD.JOIN_CHARGE" IS 80.00
DEBUG>
```

The SHOW-command may be abbreviated as "S".

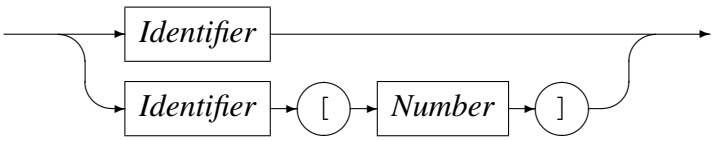
ShowCommand



Variable



IdentifierOptIndex



IndexedString

2.6 The `STEPINTO`-Command

The `STEPINTO`-command causes the next statement to be executed when the debugger is in the stopped state. Additionally, if the execution of the next statement being executed causes control to appear elsewhere within the script being debugged (for example, if the execution of an assignment statement included a script function call requiring evaluation), then a break point would be taken at the start of that function. Typically, this break point should be a *soft* break point in that it should be automatically removed once it has been triggered.

In the following example, assume the following script fragment:

```
CASATEST.vcf:1276:   function sq(n : number) : number;
CASATEST.vcf:1277:      begin
CASATEST.vcf:1278:         sq := n*n
CASATEST.vcf:1279:      end;
CASATEST.vcf:1280:
CASATEST.vcf:1281:   procedure inittest();
CASATEST.vcf:1282:      local square : number;
CASATEST.vcf:1283:      begin
CASATEST.vcf:1284:         square := sq(4);
CASATEST.vcf:1285:         calculated_fee := 0
CASATEST.vcf:1286:      end;
```

Where the debugger has stopped at `CASATEST.vcf:1284` (possibly because of a break point):

```
STOPPED AT "CASATEST.vcf":1284
```

```
DEBUG>
```

Entering the STEPINTO-command here should result in a break point being taken at CASATEST.vcf:1278:

```
DEBUG> stepinto
BREAKPOINT STOPPED AT "CASATEST.vcf":1278
DEBUG>
```

The STEPINTO-command may be abbreviated as "SI".

StepIntoCommand

2.7 The STEPOVER-Command

The STEPINTO-command causes the next statement to be executed when the debugger is in the stopped state. If the execution of the next statement being executed causes control to appear elsewhere within the script being debugged (for example, if the execution of an assignment statement included a script function call requiring evaluation), then these portions of the script will be performed while the debugger is in the executing mode and the debugger API will not be presented with a break point unless a break point had been previously set using the break command.

In the following example, assume the following script fragment:

```
CASATEST.vcf:1276:   function sq(n : number) : number;
CASATEST.vcf:1277:      begin
CASATEST.vcf:1278:         sq := n*n
CASATEST.vcf:1279:      end;
CASATEST.vcf:1280:
CASATEST.vcf:1281:   procedure inittest();
CASATEST.vcf:1282:      local square : number;
CASATEST.vcf:1283:      begin
CASATEST.vcf:1284:         square := sq(4);
CASATEST.vcf:1285:         calculated_fee := 0
CASATEST.vcf:1286:      end;
```

Where the debugger has stopped at CASATEST.vcf:1284 (possibly because of a break point):

```
STOPPED AT "CASATEST.vcf":1284
DEBUG>
```

Entering the STEPOVER-command here should result in a break point being taken at CASATEST.vcf:1285:

```
STOPPED AT "CASATEST.vcf":1284
DEBUG> stepover
BREAKPOINT STOPPED AT "CASATEST.vcf":1285
```

```
DEBUG>
```

The STEPOVER-command may be abbreviated as "SO".

StepOverCommand

2.8 The STEPOUT-Command

The STEPOUT-command causes execution to continue to the end of the current chain of sequential instructions. The effect of the STEPOUT-command is to cause a soft break point (this is a break point that once triggered, is automatically deleted) to be set at the first statement following a loop (if the script is currently executing within a loop); the first statement following the then-part or else-part of an if-statement (if the script is currently executing the then-part or else-part of an if-statement); or the first statement following the statement that caused a function to be called (if the script is currently executing within a called function). If none of these are the case, then the STEPOUT-command has no effect.

In the following example, assume the following script fragment:

```
CASATEST.vcf:1276:   function sq(n : number) : number;
CASATEST.vcf:1277:      begin
CASATEST.vcf:1278:         sq := n*n
CASATEST.vcf:1279:      end;
CASATEST.vcf:1280:
CASATEST.vcf:1281:   procedure inittest();
CASATEST.vcf:1282:      local square : number;
CASATEST.vcf:1283:      begin
CASATEST.vcf:1284:         square := sq(4);
CASATEST.vcf:1285:         calculated_fee := 0
CASATEST.vcf:1286:      end;
```

Where the debugger has stopped at CASATEST.vcf:1284 (possibly because of a break point):

```
STOPPED AT "CASATEST.vcf":1284
DEBUG>
```

A break point is then set at the (first and only) executable statement of the function sq, and execution allowed to continue:

```
DEBUG> b 1278
BREAKPOINT SET AT "CASATEST.vcf":1278
DEBUG> c
```

The break point inside sq is then triggered, and execution is allowed to continue again by issuing the STEPOUT-command. This time execution stops at the soft, and implied, break point, following the assignment-statement containing the call to the sq function:

```
BREAKPOINT STOPPED AT "CASATEST.vcf":1278
DEBUG> stepout
DEBUG>
BREAKPOINT STOPPED AT "CASATEST.vcf":1285
DEBUG>
```

The STEPOUT-command may be abbreviated as "F" or "O".

StepoutCommand

2.9 The SET-Command

The SET-command is used to assign values to script variables at points in the execution of scripts being debugged. The debugger should be in the stopped-state when issuing the SET-command. The value to be assigned to the named variable is determined by an expression. This expression is evaluated within the stack of currently open scopes and contexts of the script being executed. For example, if the script is stopped in a function being evaluated, then a variable name in the expression which matches a local variable or formal parameter name of that function refers to that local variable or formal parameter, and not to any global variables which might have the same name.

The example below assumes the following script fragment where the debugger has initially stopped at CASATEST.vcf:1287, possibly because of a break point:

```
CASATEST.vcf:1276   function sq(n : number) : number;
CASATEST.vcf:1277      local res : number;
CASATEST.vcf:1278      begin
CASATEST.vcf:1279         res := n*n;
CASATEST.vcf:1280         print("sq(",n,") = ",res);
CASATEST.vcf:1281         sq := res
CASATEST.vcf:1282      end;
CASATEST.vcf:1283
CASATEST.vcf:1284   procedure inittest();
CASATEST.vcf:1285      local square : number;
CASATEST.vcf:1286      begin
CASATEST.vcf:1287         square := sq(4);
CASATEST.vcf:1288         print("sq(4) = ",square);
CASATEST.vcf:1289         calculated_fee := 0
CASATEST.vcf:1290      end;
```

The call to the function sq is stepped into and the value of the formal parameter n can be verified to have the expected value as an argument when used on the function invocation:

```
STOPPED AT "CASATEST.vcf":1287
DEBUG> stepinto
DEBUG>
```

```
BREAKPOINT STOPPED AT "CASATEST.vcf":1279
DEBUG> show n
VALUE OF "n" IS 4
DEBUG>
```

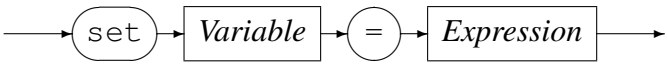
The SET-command can be used to change the value of the parameter within the called function, and the function evaluation allowed to continue using the STEPOUT-command. This causes the executing script to stop at the implied soft break point at CASATEST.vcf:1288.

```
DEBUG> set n = 16
Variable n set to 16
DEBUG> stepout
DEBUG>
BREAKPOINT STOPPED AT "CASATEST.vcf":1288
```

Since the square of the number was assigned to the local variable square, we can demonstrate that the function sq was applied to the debugger manipulated value of 16 rather than the script supplied argument value of 4:

```
DEBUG> show square
VALUE OF square" IS 256
DEBUG>
```

SetCommand

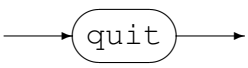


In the above, *Expression* is the expression text that will be evaluated and assigned to the given *Variable*. The required grammar of the *Expression* is documented in expeval: Expression Evaluation API Reference [1].

2.10 The QUIT-Command

The quit-command signals the end of the debugging session by turning off debugging mode and allowing the script to continue executing without further intervention of the debugger. No further commands on the debug command interface will be interpreted.

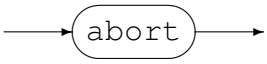
QuitCommand



2.11 The ABORT-Command

The ABORT-command calls the library function abort(3) to terminate the process running the script engine that is interpreting the script. Depending on environment and/or shell settings this produces a core file which may be useful in diagnosing script engine or environment problems.

AbortCommand

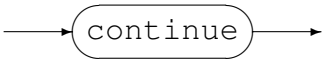


2.12 The CONTINUE-Command

The CONTINUE-command is used to switch the debugger into the executing state from the stopped state. This allows normal execution of the script to continue until the next break point is reached, or if no further break points are encountered, until the end of the script is reached.

The CONTINUE-command may be abbreviated as "C".

ContinueCommand



2.13 The LISTCHILDREN-Command

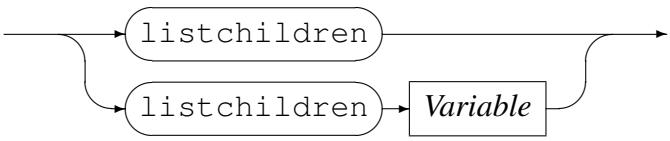
The LISTCHILDREN-command is used to navigate the variable tree of a scripting language which has support for this operation. Entering the LISTCHILDREN command without a variable name results in the names of the root variable's children being returned. Each child can then be listed by specifying the childnode's name as the operand of the LISTCHILDREN command. In this way, each leaf node in the variable tree can be visited.

The LISTCHILDREN-command may be abbreviated as "LISTC" or "LC".

```
DEBUG> listc
CHILD "A" TYPE "NODE"
CHILD "B" TYPE "NODE"
DEBUG>

DEBUG> listc A
CHILD "C" TYPE "PROPERTY"
DEBUG>
```

ListChildrenCommand



2.14 The LIST-Command

The LIST-command is used to display the source of the script. Entering the LIST command without operands results in the source code around the current point of interest being displayed. The point of interest is set the line number at which the script is currently stopped. After executing the LIST command the point of interest is changed to the last line displayed. The point of interest can be changed by the list command if either the line number or source file and line number are given as operands to the command.

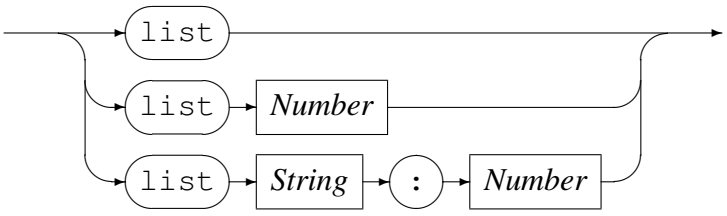
The LIST-command may be abbreviated as "L".

```
STOPPED AT "../configs/UNITTEST.vfy":1701
DEBUG> list
01696       end;
01697
01698    procedure inittest();
01699       local square : number;
01700       begin
01701          square := sq(4);
01702       --  print("sq(4) = ",square);
01703          calculated_fee := 0
01704       end;
01705
DEBUG> l 1703
01698    procedure inittest();
01699       local square : number;
01700       begin
01701          square := sq(4);
01702       --  print("sq(4) = ",square);
01703          calculated_fee := 0
01704       end;
01705
01706    procedure validate();
01707       begin
DEBUG> l
01707       begin
01708          total_system_fee := total_system_fee+test_case.JOIN_RECORD.JOIN_CHARGE
01709          total_tested_fee := total_tested_fee+calculated_fee;
01710          if calculated_fee = test_case.JOIN_RECORD.JOIN_CHARGE then pass
01711          else fail
01712       end;
01713
01714    procedure set_fee(fee_amount : number);
01715       local maxamt : number;
01716       begin

DEBUG> l "../configs/UNITTEST.vfy":1600
01595    title "DCAR_4_7"
01596    where (test_case.JOIN_RECORD.JOIN_CLUSTER = 4)
01597    or   (test_case.JOIN_RECORD.JOIN_CLUSTER = 7)
01598
```

```
01599 ;
01600
01601 -- 1,"Retail"
01602 -- 2,"SME"
01603 -- 3,"Business Bank"
01604 -- 4,"Corporate"
DEBUG>
```

ListCommand



3 Debug API Caller Interface

The caller of the `debugapi` library is responsible for making the appropriate calls to instantiate a debug environment and to populate the corresponding debug structure with the functions that interface to the script run-time engine. It is also the responsibility of the caller of the `debugapi` functions (in other words the responsibility of the particular script engine) to supply to the library the required flags and information for establishing the socket interface over which the user of the debug interface issues debug commands and expects to see responses to those commands.

References

[1] expeval: Expression Evaluation API Reference. CML Document CML00052-01, Code Magus Limited, November 2009. PDF.